# A Compositional Semantic Theory for Service Composition

Simon D. Foster

January, 2010

This Thesis is submitted for the degree of *Doctor of Philosophy*

Verification and Testing Group

**Department of Computer Science**

**The University of Sheffield**

Supervisor: Dr. Mike Stannett

WISDOM IS VINDICATED BY HER ACTIONS

# Abstract

SERVICE COMPOSITION refers to a popular modern software paradigm for building applications by combining distributed reactive components using the ***World-Wide-Web*** as the medium. The approach is characterised by the use of standardised protocols, languages and representations such as ***XML***, which ensures that services are platform agnostic. Whilst the architecture is well understood for simple single-interaction services built in the style of classical procedures from the traditional programming paradigm, much discussion is still taking place with regard to more complicated, fully reactive services, where the consumer interacts in a stateful manner.

The aim of this Thesis is to investigate the way in which composite Web services are built, and provide a useful semantic theory for service composition. Inspiration for this comes from several sources, including existing technologies such as ***WS-BPEL*** and ***WSMO***, and the ***workflow patterns*** research, which defines a diverse collection of control-flow patterns which should be provided by a service composition language.

My approach to service composition is based on ***Abstract Timed Process Calculus***. Core to this approach is ***compositionality*** – a Web service model must be semantically decomposable to allow component manipulation. An Abstract Timed Process Calculus allows elegant modelling of component systems through a variety of ***synchronisation patterns*** such as ***isochronic broadcast***. In this work I will seek to advance this area by constructing a novel timed process calculus which seeks to surpass previous calculi. This calculus will form the underlying meta-model for a semantic theory for a service composition language called *Cashew-A* . The semantic theory will be useful for both verification and execution of composite Web services.

The Thesis is in three parts:

- **PART I** explores the background to Web services and Process Calculi with a theory overview, literature review and introduction to Haskell;
- **PART II** constructs the semantic theory for service composition in terms of the overall architecture, the language and the underlying process calculus;
- **PART III** explores implementation by providing a framework for the process calculus in Haskell.

# Contents

# Acronym Definitions

| | |
|---|---|
| **ACP** | Algebra of Communicating Processes |
| **BPA** | Basic Process Algebra |
| **BPMN** | Business Process Modelling Notation |
| **CaSE** | Calculus for Synchrony and Encapsulation |
| **CaSheW** | Composition and Semantic Enhancement of Web services |
| **CBS** | Calculus of Broadcasting Systems |
| **CCS** | Calculus of Communicating Systems |
| **cCSP** | Compensating CSP |
| **CSA** | Calculus of Synchrony and Asynchrony |
| **CSP** | Communicating Sequential Processes |
| **DAML-S** | DARPA Agent Markup Language for Services (now OWL-S) |
| **GHC** | Glasgow Haskell Compiler |
| **HTTP** | Hyper-Text Transfer Protocol |
| **LTS** | Labelled Transition System |
| **OWL** | Ontology Web Language |
| **OWL-S** | Ontology Web Language for Services |
| **PMC** | Processes with Multiple Clocks |
| **SOA** | Service Oriented Architecture |
| **SOAP** | Simple Object Access Protocol (deprecated meaning) |
| **SOS** | Structural Operational Semantics |
| **StAC** | Structured Activity Compensation |
| **SWS** | Semantic Web Service |
| **TPL** | Temporal Process Language |
| **UDDI** | Universal Description, Discovery and Integration |
| **URL** | Uniform Resource Locator |
| **USF** | Unified Semantics Framework |
| **WSA** | Web Service Architecture |
| **WS-BPEL** | Web Service Business Process Execution Language (a.k.a. BPEL) |
| **WS-CDL** | Web Service Choreography Description Language |
| **WSDL** | Web Service Description Language |
| **WSMO** | Web Service Modelling Ontology |
| **WSML** | Web Service Modelling Language |
| **WSMX** | Web Service Modelling eXecution environment |
| **XML** | eXtensible Markup Language |

# Part I

# Background

# Chapter 1

# Introduction

> *"Web services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes ... Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service."*
> – IBM Web service tutorial (Tidwell, 2000)

## 1.1 Motivation

THE SCOPE OF THE WORLD-WIDE-WEB IS EXPANDING, with new services being continually provided to make different parts of our lives easier. It is now possible to perform a multitude of tasks using the Web, ranging from booking a holiday, to playing the stock-market, to buying groceries – the Web is becoming more and more a part of everyday life. Likewise, the Web is becoming a global platform for distributed computing, on which heterogeneous, distributed applications can be built.

The *Service-Oriented Architecture* (SOA) is a paradigm for distributed computation. SOA is essentially a component paradigm, where an application is built by invoking the services of others via the Web. These components are called *Web services*, and use the standards such as XML and HTTP to act as a medium for accessing their functionality. Many Web based companies such as *Amazon*[1] and *eBay*[2] now provide Web service based application programming interfaces (APIs), so that third-parties can make use of these services in their applications.

The primary reason for this architecture's success is its platform agnosticism and the ubiquity of the medium it uses. Unlike predecessors like Microsoft's *Component Object Model*[3] (COM), SOA does not rely on vendor specific protocols, implementations or

[1] `http://aws.amazon.com`
[2] `http://developer.ebay.com/developercenter/soap`
[3] `http://www.microsoft.com/com`

languages. Instead it uses the technologies already provided for the distribution of hypermedia, such as XML and HTTP, to make distributed reusable components available. As a result, any platform which is capable of accessing the Web is also capable of using Web services. Perhaps the most important aspect of this is *control* – a Web service may be implemented in any way the vendor desires, and its resources remain fully under its control without the need for third parties. Additionally, Web servers are themselves so well-used that it takes little extra effort to reuse the same protocols to implement application frontends alongside existing Web sites.

As a further result, the Service-Oriented Architecture is finding increasing importance in the Business Process Modelling (BPM) world, with Martin et al. (2003) outlining the many advantages of this new paradigm. Firstly, it aids inter-enterprise distributed computing by allowing the formation of weakly-coupled links between companies, preserving absolute ownership of the Web resources, and power over which services are exposed. Secondly, it is also finding use within single enterprises, where individual business processes are being formalised in terms of services. In this way Web services may be thought of as a possible replacement for all existing component-based programming paradigms, certainly where any form of distribution exists.

It is therefore clear that one of the main future component architecures is going to be the Web service architecture. However, like previous component paradigms this new Web-oriented paradigm needs a form of *programming language* which will allow different components to be glued together in different ways to fulfil different requirements, whilst dealing with Web service nuances. These languages are called *service composition languages*, and they allow execution of Web services to be *orchestrated* using constructs similar to those found in traditional imperative programming languages. There are already several such languages available commercially, the prime example being *WS-BPEL* (Jordan and Evdemon, 2007) (Web Services Business Process Execution Language). WS-BPEL is well supported by industry, and has several associated graphical CASE (Computer Aided Software Engineering) tools available. It also supports several advanced error-correction mechanisms such as *compensation* where the effects of a faulty transaction can be mitigated.

However, many of these languages currently lack an adequate formal grounding, particularly in deciding whether two orchestrations are equivalent according to some definition of behavioural equivalence. WS-BPEL in particular is computationally complete, and whilst there has been much progress in formalising it using advanced theories like the $\pi$-calculus and Petri-nets, any complete formal theory will likely be undecidable. Without a consistent notion of equivalence it is impossible to ensure that a language is *compositional* – that the overall meaning can be determined purely as a function of the parts. This property is particularly imporant for Web services, as the Web is by nature highly dynamic, and thus orchestrations need to change constantly as old services disappear and new ones appear. Without compositionality, it is impossible to decompose a Web service and hence allow incremental evolution of an orchestration. From a practical

perspective this means a recompilation of the execution model is needed every time a change occurs.

It is therefore my contention that a rich calculus with less power than $\pi$-calculus is better suited to describing composition of Web services. In this Thesis I focus on a particular class of calculi known as *abstract time process calculi*. These calculi are well known for their modelling of different *synchronisation patterns* such as broadcast. Indeed "time" is modelled as a form of synchronisation, rather than a notion of physical time. My aim is to construct a semantic meta-model for service composition which is both *compositional* and *extensible*. In the first place it will allow an orchestration to be decomposed into its parts, whilst in the second place it will allow new patterns of composition to be described, so that associated languages can evolve to meet new requirements. The language will be a *dataflow* oriented language and linked to precondition negotiation, which will allow resolution of execution order separate from control flow. I will also provide an implementation of this language in the functional programming language *Haskell* to exemplify what can be achieved with this process calculus paradigm.

## 1.2 Contributions

The main contributions of this Thesis are as follows:

- **An investigation of the key features necessary to represent service compositions and an experimental language called Cashew-A (CHAPTER 5)**. This first step it to establish what behaviour needs to be modelled so the calculus can be extended accordingly. My language considers a composite Web service in terms of its *control flow*, *data flow* and *message flow*. Message flow in particular motivates an extension of how abstract clocks should behave in my timed process calculus. Additionally I consider *compensable transactions*.

- **A novel abstract time process calculus called CaSE$^{ip}$ with a bisimulation semantics (CHAPTER 6)**, which I have developed to allow the features of Cashew-A to be represented through various *synchronisation patterns*. The calculus's main novelty is that, in contrast to previous calculi such as CaSE(Norton et al., 2003), clocks may be in *three* (as opposed to two) distinct states within a process: *stalled*, *patient* and *active*. This is achieved by a tripartite clock sort which underpins the structural operational semantics. This novelty allows a greater degree of flexibility in forming compositional process behaviour and hence is well-suited to specifying component system model.

- **An extensible semantic framework using CaSE$^{ip}$ to give a formal behavioural semantics to Cashew-A (CHAPTER 7)**. The main feature of this framework is a synchronous *protocol* which allows all components to be abstracted to a common communication interface. This semantic framework is both *compositional*, meaning

that a model can be manipulated without full reconstruction, and *extensible*, meaning constructs additional to those in Cashew-A can be specified with the protocol.

- **An implementation of CaSE$^{ip}$ in Haskell (CHAPTER 8)**. This allows both finite-state verification and execution processes, and includes a supporting command-line process experimentation tool called *ConCalc*. The use of Haskell is motivated by the ease in which the formal theory of CaSE$^{ip}$ may be converted into inductive functions. Haskell also has a sophisticated type-system which allows fine-grained specification of processes and their underlying behaviour. Haskell's *Monads* also allow an elegant way of binding real world behaviour to synchronisations.

- **A partial implementation of the Cashew-A semantic framework (CHAPTER 8)**. This uses the above CaSE$^{ip}$ implementation, through which the viability of the semantic framework is demonstrated. I provide a simple *Calculator* workflow as an example Web service.

## 1.3 Reading Pathways

There are several pathways through the chapters of this Thesis.

- For readers wishing to assess the **main theoretical contribution** of this work, but not the implementation aspects, I recommend $2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 9$.

- For readers interested in **Web service languages**, but not process calculus theory I recommend 2 (first half) $\rightarrow 4 \rightarrow 5 \rightarrow 9$.

- For readers interested in **Process Calculus** theory, but not my application, I recommend $2 \rightarrow 6$ (some motivating work from Chapter 5 may also be required).

- For readers interested in **Functional Programming** and **Process Calculus**, I recommend $2 \rightarrow 3 \rightarrow 8$ (and possibly also Chapter 6).

# Chapter 2

# Literature Review

*This chapter provides a basis for the rest of the thesis. There are two primary areas which must be covered – **Web services** and **Process Algebra**. My aim in this thesis is to model the composition of the former using the latter. I therefore first demonstrate the core concepts associated with Web services such as orchestration, and then describe the key technologies on which I am basing my work. I then describe the link between Web services and Process Algebra, and look at some of the existing theory. Finally I look at **Timed Process Algebra** and argue why it is a natural theory for modelling service composition.*

## 2.1 Web services

WEB SERVICES ARE PROGRAMMABLE COMPONENTS which use the World-Wide-Web as a medium for describing the functionality of real world services in a computer manipulable way (Kuropka and Nern, 2006). According to the *Oxford English Dictionary* a service is *"the action or process of serving"* or alternatively *"an action of assistance"* thus a Web service is any sort of Web accessible tool which is in some context useful. A service can be anything from ordering a shipment of steel to providing a particular digital image. In the context of computer science, a Web service can be viewed as a distributed component which can be integrated into applications and which describes itself using some standardised method.

Like all components in a programming environment, a Web service has two facets:

- An interface or *type* (the service's *observable behaviour*);
- An implementation (the service's *internal behaviour*).

A Web service's interface represents the *protocol* by which a party may interact with it. An interaction is usually represented as a message exchange between two or more parties, using HTTP and an XML-based message language like SOAP as an underlying

medium. The implementation, in contrast, represents how the Web service fulfils the task which a client requests it to perform – how it implements the interface. This will usually involve subcontracting parts of the task to other Web services by communication with their respective interfaces. The internal part of a Web service is called the *orchestration*, which is defined by the *Web Consortium* as:

> *"... the sequence and conditions in which **one** Web service invokes other Web services in order to realise some useful function."* (Haas and Brown, 2004)

An orchestration can thus be seen as a kind of program, which calls other Web services as its methods. It describes how multiple services can be composed in order to achieve some combined service. In general this is done in terms of *control flow*, *dataflow* and *message flow*.

**Control flow** governs the order in which communications to the participants of an orchestration are made. Service composition languages such as WS-BPEL (Web Services Business Process Execution Language) and OWL-S (Ontology Web Language for Services) use several control flow constructs which are familiar from the imperative programming and process algebra world, such as *sequence*, *choice* and *parallel*. In addition constructs are included which specifically reflect the nuances of Web services, such as sending and receiving messages. In the context of *business process modelling* much research has focused on determining what the fundamental *patterns* of control flow are. In particular van der Aalst et al. (2003) have made a detailed study into this by defining a total of 43 control-flow patterns using Petri-net semantics and contrasting the expressivity of different service composition languages in terms of how many patterns they implement.

**Dataflow** by contrast defines how data is directed between activities within an orchestration, and is usually orthogonal to control flow (though usually respecting the structure dictated). WS-BPEL and OWL-S handle dataflow fundamentally differently. The former contains mutable variables, which can be populated by service invocations and copied between control flow scopes. The latter allows the specification of inputs and outputs to *processes*, which can be connected together. The main issue for dataflow is deciding when a process has received sufficient inputs to allow execution. Both WS-BPEL and OWL-S have constructs which allow the order of execution to be determined purely by when a process is ready to execute, rather than a control-flow construct like sequence. Therefore the definition of process "readiness" is an important factor in dataflow semantics.

**Message flow** describes the exchange of messages between a Web service and its clients. A Web service's interface at the simplest level describes the Web service's *observable behaviour* in the form of the messages it can send and receive and at which point. WS-BPEL has several constructs which allow messages to be sent across what it calls *partner links* (see Section 2.2.2), which represent links to a single partner. Each link has an associated message flow which is determined by the order in which messages are

sent and received over the link. A fully specified WS-BPEL process therefore also has an associated *abstract process* which includes only constructs which induce observable behaviour and abstracts away implementation details. A fully specified WS-BPEL process is called an *executable process*. Similarly the *Web Service Modelling Ontology* (WSMO) splits the description of a Web service into observable and unobservable perspectives, which it calls **choreography** and **orchestration**.

Message flow is strongly linked to the "choreography" concept, though the definition of the term varies. Roughly, a choreography defines the order in which these message exchanges are made between partners. The Web Consortium defines a choreography like this:

> *"the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state."* (Haas and Brown, 2004)

Choreographies are used to concretely specify transactions between multiple parties. A choreography effectively describes a *cross-cut* over several Web services and therefore each Web service's interface must conform to the protocol which the choreography predetermines. In WSMO a choreography is seen as simply the Web service's observable interface, and this is the definition I adopt for the remainder of this work.

Having given an overview of the basic Web service concepts, I now proceed to examine the various technologies in more detail.

## 2.2 Technologies

The main three contemporary Web service technologies I focus on for this thesis are WS-BPEL, the *Web Ontology Language for Services* (OWL-S) and the *Web Service Modelling Ontology* (WSMO), the latter two of which are heavily influenced by the development of the *Semantic Web service* paradigm. A "Semantic" Web service is a Web service which provides a logical description of its purpose using a *Description Logic* and several logical knowledge bases called *ontologies*.

WS-BPEL is a well-developed and supported Web service language for describing executable business processes. OWL-S (McIlraith et al., 2001) is an attempt to leverage service composition into the Semantic Web world. It provides both a description of *processes* which give an execution semantics for the Web service and also an underlying logical framework, though I am only concerned with the process language. WSMO is newer, and is still under active development, whereas the OWL-S project has long since come to an end, with most of its work being integrated into its successor project FLOWS, part of the *Semantic Web Services Ontology* (SWSO) (Battle et al., 2005) (though only OWL-S is directly relevant to this work[1]). I will deal with WS-BPEL, OWL-S and WSMO in Sections 2.2.2, 2.2.3 and 2.2.5 respectively.

[1]SWSO has seen little activity since 2005. See `http://www.daml.org/services/swsf/1.1/swso/`

However, before considering these it is necessary to go over the basic languages for describing individual Web services.

### 2.2.1 Basic Web service technologies

Arguably two of the most important and certainly well used technologies in the field of basic Web services are SOAP (Box et al., 2000), which formerly stood for *Simple Object Access Protocol*, and the *Web Service Description Language* (WSDL , pronounced by many as "Wuz Dull") (Christensen et al., 2001).

SOAP is a protocol for conveying messages, and is frequently combined with HTTP (the backbone protocol for delivering Web content) to simulate remote procedure call style invocations over the Web. SOAP provides so-called *Envelopes*, which consist of one or more *headers* for conveying meta-data, and a *body*, which carries the payload (i.e. the actual message). In this way a Web service's operations can be conveyed as a series of SOAP *endpoints*, to which messages can be sent, detailing the method to be executed and the associated parameters. A response SOAP Envelope will then be sent back with the answer. SOAP Envelopes are almost always encoded using XML.

WSDL on the other hand is a language for specifying the interface of a Web service, in terms of **types**, schemata for the **messages** it uses, abstract **interfaces** (port types) for each of the available operations, and finally **bindings** which attach a concrete execution medium to an operation. The only operations specifiable by WSDL are a composite of up to one send and one receive, the most common being a request/response. A Web service would specify, for example, an input and output message, an operation type in terms of these messages, and information on how this operation can be executed using SOAP over HTTP. Web services described by WSDL and SOAP alone can only be very basic in nature, and cannot provide a complicated stateful protocol. Thus there have been a number of efforts to add statefulness, such as WS-CDL (Kavantzas et al., 2005) (Web Service Choreography Description Language) and WS-BPEL (Andrews et al., 2003), both of which provide some degree of interactional interface specification.

A third technology mentioned frequently in the Web service community is UDDI, *Universal Description, Discovery and Integration*. UDDI allows registries to be created which describe Web services so as to aid their discovery by third parties. Web services must be advertised somewhere so that consumers can make use of them, and UDDI provides such advertisement features. However, I don't consider it here in any great detail, as advertisement does not constitute a significant part of my research.

Having covered the basic Web service technologies, I now proceed to look at three important service composition languages which build on these technologies.

### 2.2.2 WS-BPEL

The first major service composition language I consider is *Web Services Business Process Execution Language WS-BPEL)*. WS-BPEL[2] (Andrews et al., 2003; Jordan and Evdemon, 2007) is a Web service oriented Business Process Modelling language, which was originally built as a conjunction of the features in *XLANG* by Microsoft, and *WSFL* by IBM. The resulting language was originally called BPEL4WS 1.1 (Business Process Execution Language for Web Services) but then entered a standardisation process through OASIS, and in 2007 WS-BPEL 2.0 was completed. BPEL extends WSDL, which specifies primitive Web service operations, with *processes* allowing the specification of composite functionality. Core to BPEL is the *partner link*, which links the Web service being described with another Web service, allowing access to its interface, as described in WSDL (cf. $\pi$-calculus channels in Section 2.3.5). WS-BPEL provides communication primitives for sending and receiving messages via these partner links, as well as executing operations.

The basic behavioural unit in WS-BPEL is the *activity*, which can be anything from sending a message to a partner, to a complex structured activity consisting of many other activities. WS-BPEL, like OWL-S, defines a number of constructs for composing communications in different ways, which it calls *structured activities*. There are a total of seven structured activities which describe different ways in which the enclosed activities can be ordered.

- **sequence**, which runs an enclosed list of activities one after the other;
- **if**, analogous to the classic *if-then-else* construct which chooses the next activity to execute based on one or more logical expressions;
- **while**, which repeats the execution of an activity while an expression evaluates to true;
- **repeatUntil**, a variant of the above;
- **pick**, which chooses the next activity to execute based on event guards. Example events include reception of a message (**onMessage**) or a timeout (**onAlarm**). The branch is chosen once the first of the given events occurs;
- **flow**, a complicated graph based pattern. At its most basic, a **flow** activity runs all of enclosed activities in parallel. In addition links between the activities may be specified, which leads to a partial order between the activities executed;
- **forEach**, at its most basic an iterator which executes the enclosed activity once for each value of a counter, like a normal *for* loop. In addition, **forEach** supports parallel execution, which spawns $N \in \mathbb{N}$ (for some specified $N$) copies of the activity to run in parallel (this is equivalent to $\pi$-calculus ! operator – see Section 2.3.5).

[2]See `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html` for the specification.

WS-BPEL also has an advanced fault handling system, which includes both the standard exception system present in languages like Java, and a *compensation* system, which allows the specification of *transactions*. A transaction is an interaction between two or more parties with a specific shared goal, certain protocols of execution, and restrictions on either party's capabilities (Gray, 1981). A transaction assumes the presence of a *contract* which lays out the rules by which a transaction may proceed. The failure to complete the transaction, such as by a party making an illegal move, will lead to some sort of redress being performed, in which the transaction will be "rolled back" to its initial state, if possible.

Web services are inherently unreliable media, not merely in terms of the protocols, whose issues can be overcome somewhat by the service bus, but also at a higher level. Since Web services cannot directly keep track of all the distributed resources needed to complete a transaction, it is not always possible to ensure prior to execution that a transaction will succeed. Extenuating circumstances can and frequently do arise in real-world transactions which prevent a goal from succeeding, perhaps due to an unforeseen circumstance such as lack of stock, or something as simple as the client wishing to cancel. When this occurs, a Web service needs to know exactly where it is in the transaction, and how it can revert as much as possible to the state before it started.

In the world of relational databases and similar media it is assumed that a transaction will either complete successfully and commit, or fail for some reason and thus abort. It is necessary that the system only makes legal moves in the transaction and that no outside force can otherwise alter its course. Integrity is maintained by the presence of so-called *ACID* properties (Haerder and Reuter, 1983), namely:

- *Atomicity* – either all the actions of transactions are performed, or none of them are (all or nothing);
- *Consistency* – the state of the database remains consistent with the transaction;
- *Isolation* – no outside agent can observe the partial state changes being made, a change is only observed upon commit;
- *Durability* – the state of the transaction survives system failure (e.g. via a transaction log).

A relational database maintains these properties by locking the resources it requires to make the transaction, and making them available once the transaction either commits or aborts. However, such properties are impossible to maintain in the Web service world. A Web service cannot lock resources as they are distributed and required by other clients. For this reason state change is always transparent and atomic rollback is not possible. Thus, the Web service architecture advocates the use of a weaker form of rollback based on *compensation*. Compensation is a concept developed to cope with rollback of Long Running Transactions (LRTs) which face similar problems to Web services

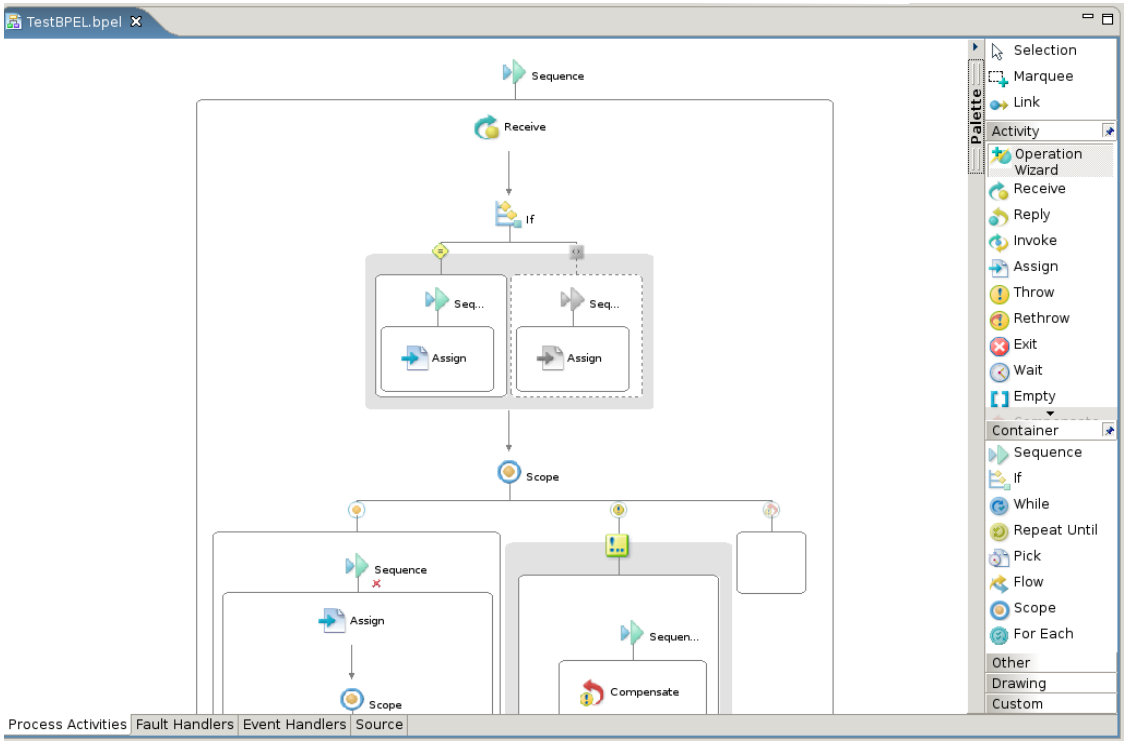



Figure 2.1: Active BPEL WYSIWYG interface screenshot

(Garcia-Molina and Salem, 1987). Instead of having every action being reversible, a Web service transaction allows every activity to have an associated compensation activity, which applies redress.

To enable the implementation of compensable transctions WS-BPEL has a **scope** operator, which encloses an activity workflow. Each **scope** has associated **compensation**, **fault** and **event** handlers. Event and fault handlers react to relevant events in their associated scope by executing other activities (cf. Java's **catch** mechanism which reacts to an exception), with the latter also terminating the enclosed activity. Compensation handlers facilitate rollback for failing non-atomic transactions, and are triggered by the **compensate** activity which can be used in fault handlers. If a compensation handler is not explicitly specified for a scope, the default compensation handler is installed which compensates the scope in reverse order of its execution.

WS-BPEL is primarily of interest to this thesis due to its popularity, particularly in the business world and because existing attempts at formalisation, such as by Lucchi and Mazzara (2007) and Butler, Ferreira and Ng (2005) have met limited success, and in general only formalise a fragment. In particular, verification is difficult due to the language being computationally complete and providing a large set of features. One thing is certain – there is a strong link between WS-BPEL and $\pi$-calculus (see Section 2.3.5) and thus any theoretical issues with $\pi$ are carried over.

Despite these limitations WS-BPEL remains the definitive language for business process modelling. This is clearly represented by the number of commercial implementa-

tions it has, including Oracle's *BPEL Process Manager*[3] (which also provides a WYSIWYG tool for designing Web services) and the open source *ActiveBPEL Engine*[4] (see Figure 2.1 for the interface). Thus it represents an important development in the Business Process Modelling World, and its most useful features should be integrated into any new language aimed at Business Process Modelling.

### 2.2.3 OWL-S

The second language I consider is the *Web-Ontology Language for Services (OWL-S)*. OWL-S is a language for describing the behaviour of Web services and is described in the technical report of Martin et al. (2004), which gives the structure of the three part *ontology* for services:

- The **Profile**, which gives a logical description of a Web service's purpose;
- The **Grounding**, describing how a client can communicate with the Web service, in terms of the protocol used, the location of the Web service etc. (specifically relating to communication level protocols like SOAP);
- The **Model**, describing how the Web service invokes other services to achieve the overall goal.

The **Profile** is an ontological description of the services provided by a Web service being described, and the requirements of these services. OWL-S places no restrictions on how these should be represented, allowing any constraints which can be represented by OWL. Specifically, a profile will normally describe the provider of the service, the functional properties of the service (i.e. what it does) and an indeterminate selection of other data about the service, which can be broadly called its *non-functional properties*. The report suggests that response time and quality of service ratings could be included here.

The **Grounding** describes the method by which a client can communicate with the service, in terms of the protocol used, with the *Web Service Description Language* as a basis. WSDL only allows the definition of Web services which receive a single request message, and send a response message back (or a one-way message in either direction), and thus it is assumed that all Web services in OWL-S will be of this form (a limitation).

The majority of Martin et al. (2004) is devoted to the main contribution of OWL-S: the **Service Model** (also called the *Process Model*), which describes the Web service's internal behaviour. At the core of the service model is the *process* concept, which acts as the main building block for compositions. All Web services are abstracted to processes, representing their control flow and dataflow. Processes are split into two distinct classes:

[3] `http://www.oracle.com/technology/products/ias/bpel/index.html`
[4] `http://www.activebpel.org/`

- **Atomic** processes, which act as frontends for outsourced Web services. They are inherently *one-shot*, i.e. taking a set of inputs, and producing a set of outputs (possibly because of the limitations set by WSDL);
- **Composite** processes, which are built by composing together other processes. They are further split into six types; namely
  - **sequence**;
  - **split** (runs a collection of processes in parallel asynchronously);
  - **split-join** (split with synchronisation after completion of all processes);
  - **any-order** (defer resolving the order of execution to the processes themselves);
  - **choice**; and
  - **if-then-else**.

  Processes which are composed by these patterns are further abstracted behind **performances**, which instantiate the process within a particular behavioural context. Processes are reusable, but performances are not. This is an important distinction between the prototype and an instance of that prototype.

Each process is assigned a set of inputs, which can be manipulated and passed to its inner behaviour, and a set of outputs. Depending on the type of process, not all inputs may be required, and not all outputs may be produced, particularly where choice of some sort is present inside. Thus, the **any-order** construct reflects not simply an arbitrary order, but one where the preconditions of the enclosed processes determine the order. However, the method of handling dataflow is left mainly to the implementor. OWL-S has at least three behavioural semantics, one based on Petri nets by Narayanan and McIlraith (2002), one based on Concurrent Haskell and Erlang (Ankolekar et al., 2002) and one using Timed Process Calculus (Norton et al., 2005).

The aim of the latter is to create a semantics for OWL-S which is both complete (the Petri-net semantics only implements an earlier OWL-S fragment) and more importantly *compositional*, which neither of the former two semantics claim to be. Work on this semantics also spawned our project, called *Cashew* (Foster et al., 2005; Norton, 2005*b*), with the aim of investigating service composition language semantics with compositionality as a fundamental property. This work uses a *timed process calculus* called Cashew-Nuts, a variant of CaSE(see Section 2.5.3), to give a compositional semantics to a language called Cashew-S. Cashew-S is essentially the same as the OWL-S service model language, but removes a "magic" variable called theParentPerform which allowed a workflow to access the variables of its parent context (thus breaking compositionality). Instead data is passed around purely on a basis of input and output parameters, connected via *dataflow connections*, which helps ensure compositionality. Dataflow was the main driving force behind this effort, which showed that CaSE could soundly represent the precondition

system of OWL-S, particularly in the **any-order** pattern. The Cashew-S semantics will be discussed in detail in Section 2.5.3.

The OWL-S service model, and particularly its pre/postcondition system, provides one of the main inspirations for my work. The composition language I describe in Chapter 5 directly draws on it, owing to its sensible subset of language constructs. The *performance* concept also proves vital for giving a compositional semantics to Web service composition.

### 2.2.4 Workflow Patterns

So far in this review I have considered two service composition languages which have broadly comparable control flow constructs, but with different expressivity. In fact there is a wide variety of such workflow languages, all with very different constructs. In response to the many and varied workflow languages which have appeared with different levels of expressivity (WS-BPEL is one, but there are over 13 others), Wil van der Aalst's research team at Eindhoven in conjunction with Arthur ter Hofstede's team at Queensland have created the *Workflow Patterns* initiative[5]. They describe themselves this way:

> *The* **Workflow Patterns Initiative** *was established with the aim of delineating the fundamental requirements that arise during business process modelling on a recurring basis and describe them in an imperative way. A patterns-based approach was taken to describing these requirements as it offered both a language-independent and technology-independent means of expressing their core characteristics in a form that was sufficiently generic to allow for its application to a wide variety of offerings.* (van der Aalst, ter Hofstede, Kiepuszewski and Barros, 2006)

This group has examined in detail the different perspectives of workflow languages, in an attempt to identify the key "patterns" which occur. Such patterns should provide the fundamental building blocks needed for constructing all the different types of workflow, thus providing a useful theory. The workflow pattern catalogue can then be used for contrasting different workflow systems in terms of their *expressive power*, and thus the range of concepts a language supports.

The first line of work has been in *control-flow patterns* (van der Aalst et al., 2003), which includes an initial selection of twenty patterns ranging from obvious patterns like sequence and parallelism, to more complex patterns, such as arbitrary cycles. The selection of control patterns was later revised and the number increased to forty-three (van der Aalst et al., 2006). The revised patterns are split into a total of 8 categories:

- **Basic Control Flow Patterns**, which include patterns like Sequence, Parallel Split, Exclusive Choice and Synchronisation;

[5] `http://www.workflowpatterns.com`

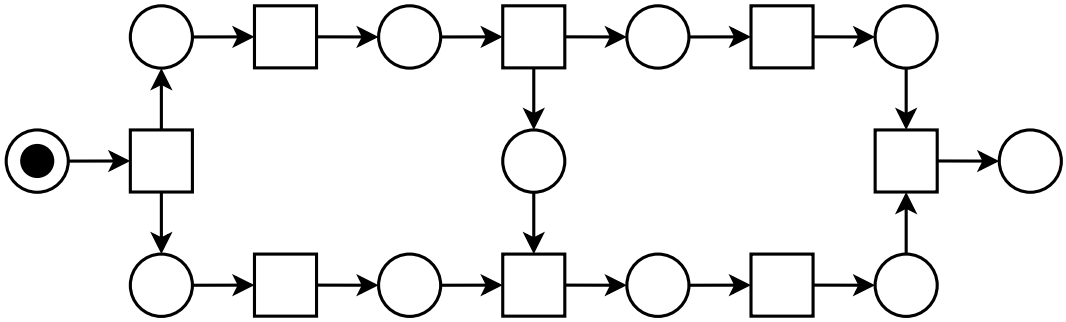


Figure 2.2: Difficult to model Petri-net workflow from van der Aalst (2005)

- **Advanced Branching and Synchronisation Patterns**, which include various complex forms of choice, split, synchronisation and merge. For instance *Multi-Choice* allows multiple parallel paths to be chosen from a bag;
- **Multiple Instance Patterns**, which describe patterns where multiple copies of the same workflow can be spawned and executed concurrently;
- **State-based Patterns**, which describe patterns linked to the state of the system. For instance *Deferred Choice* makes a decision based on an interaction with the operating environment (receiving a message, for example);
- **Cancellation and Force Completion Patterns**, which include patterns where an activity's execution can be halted and either cancelled or force completed (exception handling, for example);
- **Iteration Patterns**, which include the different forms of iteration and recursion;
- **Termination Patterns**, which describe the circumstances under which a workflow completes;
- **Trigger Patterns**, which include events based on triggering of tasks.

Key to this work has been the use of *Petri-nets* as the underlying theory. Rather than being bound to a more algebraic setting, along with the restrictions of block structuring, the workflow patterns allow very flexible flow representation. This is perhaps best illustrated by the *Arbitrary Cycles* pattern (in Iteration Patterns) which allows loops with multiple entry and exit points (in a similar manner to the classic GOTO statement), thus allowing, for instance, a "figure of eight" shape example. It is not possible to build such a pattern by composition in a purely process algebraic, or indeed any block structured setting, as it isn't possible to form a jump from one point in a block-structure to any other point. Nevertheless, the claim of van der Aalst et al. (2003) is that Arbitrary Cycles should not be dismissed as GOTO is, as the graphical nature of workflows makes them easier to interpret, and necessary for solving complex problem elegantly. The Petri-net shown in Figure 2.2 was set as a challenge for elegant modelling in a Process Calculus

such as CCS or $\pi$-calculus. This pattern is difficult because the top branch must progress to the middle before the bottom section can progress to the end.

The use of Petri-nets also means that parallel split and synchronisation or merge can be split into separate patterns, whereas in a block-structured language they would be merged. This allows the different types of synchronisation to be studied independently. For instance it is possible to distinguish a synchronisation where all concurrent threads must synchronise, from one where only a subset must synchronise. A further claim of van der Aalst (2005) is that Petri-nets are both graphical and formal, and thus they provide a simple way to understand the workflow semantics in a theoretically sound setting.

Although the work in this thesis focuses on *Process Calculus* and not Petri-nets, I utilise the Workflow Patterns in designing my orchestration language in Chapter 5. I aim to demonstrate that my approach provides the best of both worlds between block-structuring and graph structuring.

### 2.2.5 WSMO

The *Web Service Modelling Ontology* (WSMO) described by Roman et al. (2005) is a group of languages and technologies which follow a *Goal-driven* approach to describing Semantic Web services. This approach enables decoupling of intent and provision, of description and of implementation – ultimately of pragmatics and semantics. Instead of representing a particular Web service which can satisfy the user's needs, the goal specifies what these requirements are, deferring selection of an actual concrete Web service to the *service broker*, a server which dynamically selects suitable services based on requirements.

When seeking a component to solve a particular problem within an application, instead of specifying something concrete, WSMO allows a semantic description of the problem (the "Goal") to be inserted, with the service broker seeking suitable concrete services. Components in WSMO are thus abstracted from the context in which they will be used. This style of component modelling is not new, and there are many parallels with separation of type and data, as found in Object Oriented Programming, although the types in WSMO are based more on the semantics of the problem they are describing, than simply syntax.

WSMO considers Web service in terms of four abstract concepts, which together allow Semantic Web service descriptions to be formed:

**Ontologies** are logical knowledge bases which provide the vocabulary for the Web service. The ontology describes the concepts which exist, their attributes and the relations between them.

**Web services** are the raw materials of the Web. They provide services on behalf of particular parties. A Web service is described in terms of its **capability**, which provides a logical description of what it does, and its **interface** which describes how we

can communicate with it and how it communicates with others. The latter is split into a **choreography**, which describes the service interface and an **orchestration**, which gives the implementation.

**Goals** are milestones in a computation which the client wishes to achieve. They describe what the intent of the client is, for example the specification of a holiday they wish to book. At the simplest level, a goal is specified in the same way as a Web service, having a capability and an interface. A goal represents the *intent* of the client, whereas the Web service specifies the actual behaviour, which may be used (via mediation) to fulfil a goal.

**Mediators** link heterogeneous parts of the system together by providing mappings. For instance, two Web services may use different currencies for a financial exchange and an appropriate Web service to Web service mediator would map between them.

Unlike WS-BPEL and OWL-S which include a workflow language as part of their specification, WSMO does not include such a language for representing a Web service's **orchestration**. Instead, it has a lower level behavioural model based on Börger's *Abstract State Machine* (ASM) model (Börger, 1999). An abstract state machine is an automaton whose behaviour is described in terms of a *state signature* and a collection of *transition rules*. The state signature is effectively a data-type or collection of data-types, which describes the possible states which the machine can evolve into. The evolutions are defined by the transition rules, which take the form of "test $\leftarrow$ update", where the test checks if some condition is true in the state signature and allows update to act if so. In a WSMO ASM each attribute in the state signature can be further grounded to a particular Web service function. For instance, some attributes may be declared as having the *IN* mode, which means they can only be read by the machine but never written to – only the machine's environment can populate them. Such attributes may be used to represent an incoming message from a partner. Likewise, some attributes may have the *OUT* mode, representing a message which the machine sends. WSMO ASMs provide a powerful model for describing Web service orchestrations; nevertheless, they are also very cumbersome to use and are better suited to act as a *meta-model* for an actual workflow language.

To conclude, WSMO is a very capable model and due to its ASM basis provides a more flexible basis for describing orchestrations. Because of this, and the fact that WSMO is the most contemporary of the service technologies, I will be adopting its terminology for my work.

## 2.3 Process Algebra

Having discussed all the relevant aspects of Web services I now move onto the second part of this literature review. Some of the most important models studied in the description and simulation of Web services are the group of formalisms known as *process*

*algebras*. In their introduction to the *Handbook of Process Algebra*, Bergstra et al. (2001) describe a process algebra this way:

> *"A process algebra is a formal description technique for complex computer systems, especially those with communicating, concurrently executing components."*

Process algebras enable computer scientists to build models of real-world concurrent systems, for the purpose of predicting their behaviour. Since it is cumbersome to reason directly about physical systems, it is sometimes useful to draw out the relevant attributes to reason about, whilst abstracting away the irrelevant details. Bergstra et al. further explain that all process algebras share three key constituents: **Compositional modelling**, **Operational semantics** and **Behavioural reasoning**.

**Compositional modelling** involves building a system up from smaller parts, using a small number of primitive operators. Given two processes $P$ and $Q$, a process algebra provides several binary operators for joining them together into a larger system. We could for instance run them in parallel, represented, for example, as $P \mid Q$, or we may build a system which makes a choice between them, $P + Q$. The idea is to discover the most fundamental set of operators, which don't overlap in any way, so as to lead to a suitably expressive language which is cleaner to reason over. The *Calculus of Communicating Systems* (Milner, 1989*a*) (CCS) for instance, a very important and well-studied process algebra, contains only 6 or 7 operators (depending on the variant) and yet allows the specification of a wide variety of concurrent systems (see Section 2.3.3).

**Operational semantics** is an approach to describing the behaviour of a formal system (many other forms of expressing behavioural semantics are detailed by van Glabbeek (2001)). An operational semantics describes the individual steps which a system can take to evolve into another process. For example, a choice is usually resolved by the execution of a single action, and the operational semantics must describe how this occurs. Such a semantics may for instance contain a rule of the form *if a process $P$ can evolve into $P'$ by performing an $\alpha$ then a choice between $P$ and another process $Q$ can be resolved to $P'$ if an $\alpha$ action is permitted*.

The most common approach to describing operational semantics is called *Structural Operational Semantics* (SOS) created by Plotkin (1981). A structural operational semantics consists of a collection of inductive logical rules of the the form $\frac{antecedent}{consequent}$, which describe the conditions under which the system can evolve. These rules induce a *labelled transition system* which describes the complete behavioural transition structure of the given process. A labelled transition system consists of three parts :

1. A state label alphabet $\mathcal{P}$;
2. A transition label alphabet $\Lambda$;
3. A relation giving the transitions $\rightarrow \subseteq \mathcal{P} \times \Lambda \times \mathcal{P}$.

A collection of SOS rules entails the definition of the transition relation $\rightarrow$. Additionally, when describing a process algebra, it is usual to adopt the shorthand $P \xrightarrow{a} P'$ for $(P, a, P') \in \rightarrow$, read as "$P$ can do an $a$ and evolve into $P'$". For instance, I might give the rule:

$$\textsf{Sum1} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

This rule, labelled Sum1, states that, if some process $P$ (drawn from $\mathcal{P}$) can perform an action $\alpha$ (drawn from $\Lambda$), resulting in process $P'$ (i.e. $(P, \alpha, P') \in \rightarrow$), then the process formed by composing $P$ with another process $Q$ using operator $+$ can also perform $\alpha$ and evolve into $P'$ (i.e. $(P + Q, \alpha, P') \in \rightarrow$). Structural operational semantics is the primary method for giving operational semantics in this thesis.

In contrast, an alternative type of semantics to SOS is *Algebraic Semantics*, where a process algebra is primarily described in terms of the *equational axioms* of its operators. For instance, the *Algebra of Communicating Processes* (Bergstra and Klop, 1984) is described entirely in terms of equational axioms which identify its operators (although a corresponding operational semantics is also provided). Such axioms may state, for instance, that the parallel operator is commutative, or that choice distributes over parallel composition. Different sets of axioms allow different types of algebras to be distinguished. For instance it is unusual in process algebra for sequential composition ; to distribute over $+$, as this would be unsound with respect to *bisimulation*, an important behavioural equivalence which I shall describe in the next paragraph. If a process algebra does not have some form of underlying algebraic theory a better reference term is *process calculus*, which I also use in this thesis.

Finally, **behavioural reasoning** is the actual end objective of a process algebra. Once we have built a system model, it is necessary to be able to draw some conclusions from it, and without this ability the modelling language is of little use. Usually behavioural reasoning involves the comparison of different processes through behavioural relations, such as behavioural equivalences and preorders. A behavioural equivalence defines when two processes' behaviours are in some way *indistinguishable*, and thus the one process can be substituted for the other. A well known and understood behavioural equivalence is *bisimulation*, where two processes are considered equivalent if they can match each other's moves in each state they find themselves in. Bisimulation and associated equivalences will be described in more detail in subsection 2.3.1. Another, much coarser equivalence is *trace equivalence*, where two processes are considered equivalent if they possess the same sets of possible atomic move sequences. Between these two equivalences there is a whole range of relations with different characteristics as described by van Glabbeek (2001).

A behavioural preorder on the other hand describes when one process (or the behaviour thereof) is in some way *"less-than-or-equal-to"* another process. For instance, *simulation* requires that one process be able to match another process's move in each step

(but not vice-versa). Preorders tend to be types of *refinement relations*, stating when a given implementation of a system satisfies a specification process, having at least the specified features.

A behavioural relation can either be *semantic* or *axiomatic*. A semantic relation is a relation defined in terms of the transition systems which two processes represent – such relations associate directed transition graphs. An axiomatic relation on the other hand is defined directly on the process syntax, and shows the algebraic properties of various process operators. For instance, it may be desirable that choice is associative and commutative. These types of relation are in no way mutually exclusive, for instance it is possible to take a semantic equivalence such as bisimulation and axiomatise it over a given calculus. Once axiomatised it must be shown that the axioms are consistent with the semantic equivalence (soundness) and that every equivalence pair is represented (completeness). Some process calculi combine the two approaches, for example the $\pi$-calculus (Milner, 1999) is first given a small axiomatic equivalence called a *structural congruence* which is then used as a basis for the operational semantics. A possible advantage of axiomatisation over a semantic equivalence is that it overcomes the need to generate a transition graph, which can be very costly.

I will now outline bisimulation in more detail.

### 2.3.1 Simulation and Bisimulation

The *simulation* preorder is one of the simplest semantic relations which can be formed on labelled transition systems. Simulation states, in essence, that for one system to *simulate* another, it must be capable in every state of performing all the same transitions, and each pair of resulting states must also be *similar*. Due to the nature of Labelled Transition Systems, relations defined on them are invariably (co)inductively defined, as is the case with simulation. We can therefore describe the condition for similarity between two processes $P$ and $Q$ semi-formally thus:

> *A process $P$ simulates a process $Q$ provided for every $Q'$ into which $Q$ can evolve by doing $\alpha$ there is a corresponding process $P'$ into which $P$ can evolve by doing $\alpha$ and $P'$ simulates $Q'$.*

That is, $P$ can match every single action of $Q$ (but may also have additional actions), and the resulting states $P'$ and $Q'$ have the same property. If we think of it as a game in which to "win" means that $P$ simulates $Q$, then any move which $Q$ can make must be countered by an identical move from $P$.

If this relation is also made symmetric, so that the processes much match each other's moves, the equivalence formed is called a *bisimulation*. Bisimulation is almost tantamount to process algebra (particularly CCS), in that many theories are based on it and a great deal of work has been done in trying to explore it. Bisimulation can be defined inductively:

**Definition 2.3.1** ***Bisimulation***
*A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a bisimulation provided that for all $\langle P, Q \rangle \in \mathcal{R}$:*

- *If $P \xrightarrow{\alpha} P'$ then $\exists Q'.Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$*
- *If $Q \xrightarrow{\alpha} Q'$ then $\exists P'.P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$*

The finite relations induced by this definition can be combined to create a coinductive relation $\sim$ called *bisimilarity*, the set of all bisimilar process pairs:

$$\sim \; = \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\}$$

Then $P \sim Q$ means that there exists a bisimulation $\mathcal{R}$ with $\langle P, Q \rangle \in \mathcal{R}$. Bisimulation is usually used indirectly as an equivalence relation for a process calculus. One of the most frequently used variants of bisimulation is *weak bisimulation*, also known as observation equivalence, which allows a more component-oriented view of processes. Process languages frequently contain one or more special, distinguished actions called *silent actions* sometimes written as $\tau$. This action stands for activity which occurs but is not directly observable. For instance, if two agents can communicate, when a transition system is formed for the combined process, this communication must have a transition associated with it so that both agents can move forward. However, since this is effectively a private communication, no other party can take part, and thus a silent action is substituted. Weak bisimulation takes advantage of the fact that, since silent actions are effectively irrelevant to any observer, they can be ignored when checking two processes for equivalence. This idea is formalised through the creation of a derived weak transition relation $\Rightarrow$. This transition relation references only observable actions which can be seen by another process.

**Definition 2.3.2** ***(Weak Transition Relation)***
*Given a labelled transition system $(\mathcal{P}, \Lambda, \rightarrow)$ where $\rightarrow \subseteq \mathcal{P} \times \Lambda \times \mathcal{P}$. The associated weak transition relation $\Rightarrow \subseteq \mathcal{P} \times \Lambda \times \mathcal{P}$ is the smallest relation closed under the following rules:*

$$\frac{-}{P \overset{\epsilon}{\Rightarrow} P} \qquad \frac{P \xrightarrow{\alpha} P}{P \overset{\hat{\alpha}}{\Rightarrow} P'} \qquad \frac{P \overset{\epsilon}{\Rightarrow} \overset{\hat{\alpha}}{\Rightarrow} P'}{P \overset{\hat{\alpha}}{\Rightarrow} P'} \qquad \frac{P \overset{\hat{\alpha}}{\Rightarrow} \overset{\epsilon}{\Rightarrow} P'}{P \overset{\hat{\alpha}}{\Rightarrow} P'}$$

(where $P, P' \in \mathcal{P}$, $\alpha \in \Lambda$, and $\hat{\alpha} = \epsilon$ if $\alpha \equiv \tau$ and $\alpha$ otherwise)

The special label $\epsilon$ refers to an empty sequence of actions. It can either represent a sequence of silent actions, or a null transition, since the relation is reflexive. Therefore the weak transition relation identifies a sequence of $\tau$ transitions with doing no transition at all. The relation characterises the states into which an action can lead when surrounded by silent actions. The hat is used to map the $\tau$ action onto the empty sequence. A weak bisimulation is then simply a bisimulation over a weak transition system, where all transitions are replaced by weak ones. Weak bisimulation can therefore be interpreted as

the same relation as bisimulation, but on a different transition system, i.e. $(\mathcal{P}, \Lambda, \Rightarrow)$. Nevertheless, it is usually specified in the following (equivalent) way:

**Definition 2.3.3** ***Weak Bisimulation***
*A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation provided that for all $\langle P, Q \rangle \in \mathcal{R}$:*

- *If $P \xrightarrow{\alpha} P'$ then $\exists Q'.Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$*
- *If $Q \xrightarrow{\alpha} Q'$ then $\exists P'.P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$*

As for normal (strong) bisimulation, we can then define the relation encompassing all weak bisimulations, called *weak bisimilarity*:

$$\approx \; = \; \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation}\}$$

Having described the fundamental concepts of process algebra, I now proceed to review the most important and fundamental calculi.

### 2.3.2 Communicating Sequential Processes

Process Calculus development largely began in the late seventies, building on much previous work in classical automata theory (von Neumann, 1951; Rabin and Scott, 1959) and programming language theory (Dijkstra, 1975). It was primarily an effort to consider concurrency as a general mathematical concept to be modelled, rather than the existing ad-hoc methods found in the contemporary concurrent programming languages.

Hoare (1978) begins from a traditional concurrent programming paradigm. Existing methods of modelling concurrency, such as shared variables and semaphores, whilst suitable for their domain of problems did not provide a general theory of how a concurrent programming language should be formed. Furthermore, at a physical hardware level constructing such methods is fraught with difficulties.

Therefore, his new approach was to take an existing language model, namely Dijkstra's guarded command language (Dijkstra, 1975), and extend it with fundamental operators to enable concurrency. Dijkstra's basic language consists of all the operators one would expect to find in an imperative programming language, in particular sequencing, choice, repetition and variable manipulation. Hoare essentially adds two things, an input and an output command, which enable a process to send or receive a message, and a parallel construct which allows two processes to be executed concurrently. Two parallel processes may communicate when one nominates the other by name to send output to, while the other nominates the first to receive input from. Processes may not otherwise share data, and variables are scoped only over a sequential process. He then used this model to demonstrate how a number of problems may be solved, in particular the famous *"dining philosophers"* synchronisation problem.

This simple model further evolved and became the process calculus *CSP*, which is described in further detail in the CSP book (Hoare, 1985). CSP has been widely applied

in the field of formal methods for verification of concurrent systems. CSP theory is based on *trace semantics*, where each program is represented as a set of action sequences, representing all the possible paths a program may execute. CSP contains the follow basic operators:

- $e \rightarrow P$ – perform the event $e$ and then behave like process $P$ (prefixing);
- $P \,\Box\, Q$ – make a choice between $P$ and $Q$ based on their first action (external choice);
- $P \sqcap Q$ – make a choice between $P$ and $Q$ by allowing them to make the decision internally (internal/non-deterministic choice);
- $P \,|||\, Q$ – interleave the actions of $P$ and $Q$;
- $P \,|[\mathcal{A}]|\, Q$ – require $P$ and $Q$ to synchronise on the actions contained in set $\mathcal{A}$;
- $P/\mathcal{A}$ – hide the actions contained in set $\mathcal{A}$.

Actions in the latter version of CSP are not (in contrast to the earlier model described by Hoare (1978)) only uni-directional, that is to say inputs and outputs. Actions all have the same direction and multiple parallel processes may synchronise on the same action simultaneously, provided they are in the action set. This kind of synchronisation is called *multi-party synchronisation* and it is something that forms a key part of the work in this thesis.

### 2.3.3 Calculus of Communicating Systems

In contrast to Hoare's CSP, Robin Milner's *Calculus of Communicating Systems* (Milner, 1980, 1989*a*) did not start out as a programming language. The aim was always to completely remove state manipulating commands and switch entirely to processes as purely mathematical expressions. Thus the theory of CCS holds that the sole core concepts needed for a theory of distributed systems are *communication* and *concurrency*. With this in mind, CCS is built around two key concepts. Firstly **observation**, where parallel processes are said to observe the visible actions of their peers and cannot distinguish systems where the observations are identical. This follows even if the components are implemented very differently – to quote Milner *"two systems are indistinguishable if we cannot tell them apart without pulling them apart"*. Secondly **synchronised communication**, where a communication between two agents of a concurrent system is seen as indivisible and foundational to the calculus. Furthermore, the heart of the calculus is the *parallel composition* operator, which allows two systems to run in parallel. Indeed it is viewed as so central that it supersedes traditional programming combinators such as sequential composition. This is demonstrated by the small number of constructs in the basic finite calculus. If $E$ and $F$ are process expressions, then the following are also process expressions:

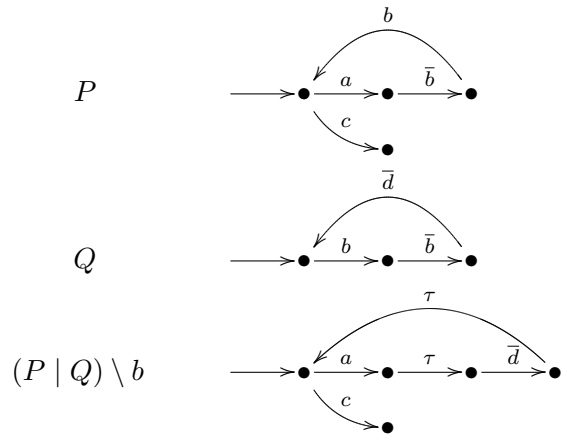
- **0** – the inactive process which can do nothing;
- $\alpha.E$ – action prefixing (as in CSP);
- $\mu X.E$ – recursive fixpoint with process variable $X$;[6]
- $E + F$ – choice (which can be internal if either $E$ or $F$ is prefixed by a silent action, or external otherwise);
- $X$ – a process variable for building recursive expressions;
- $E \setminus a$ – prevent observers from seeing $a$ actions originating from process $E$;
- $E \mid F$ – two processes running in parallel (parallel composition).

These operators are given in order of binding precedence (excepting nullary operators), from highest to lowest. Actions $\alpha$ $(\in \mathscr{A})$ are split into three classes in CCS, regular actions $a(\in \overline{\Lambda})$, coactions $\overline{a}(\in \Lambda)$ and silent actions $\tau$. Actions and coactions broadly represent inputs and outputs, though they are entirely abstract in nature. When two parallel processes can perform an action and coaction with the same underlying label, a synchronisation can occur and a silent action $\tau$ is produced. The semantics for the core operators of CCS is shown in Table 2.1. An example CCS process composition can be seen in Figure 2.3. It if made up of two agents $P$ and $Q$ which repeatedly communicate with each other until $P$ encounters a $c$ input, at which point the communication ends. The processes communicate on a restricted channel called $b$.

Action prefix, choice and recursion are called *dynamic* operators because they reduce after doing only one action. Processes built with only the dynamic operators are called *sequential*. In contrast parallel composition and restriction are *static* operators because they never reduce. The contrast is imortant because the use of static operators within recursion allows CCS to become infinite state and Turing complete (Milner, 1989*a*).

CCS is undoubtedly one of the most well known process calculi today, and almost synonymous with *bisimulation semantics*. Although not original in many ways, CCS was one of the first calculi to identify the fundamental concepts of concurrent systems and provide a concrete and useful theory behind them. CCS's equivalence theory is based on weak bisimulation, though its actual equivalence relation is a derivation called *observation congruence*. The reason for this is that weak bisimulation itself is not a *congruence relation* with respect to all the operators of CCS. An equivalence relation is a congruence whenever any two equivalent process $P$ and $Q$ remain equivalent when placed within an identical context (i.e. a process with a hole $\_$, such as $\_ + R$). For CCS, weak bisimulation is not a congruence with respect to summation – there are equivalent processes which can be distinguished by summation. For example although $a.b.P \approx \tau.a.b.P$, as expected since weak bisimulation ignores $\tau$ actions, summing the two processes with $c.d.Q$ leads

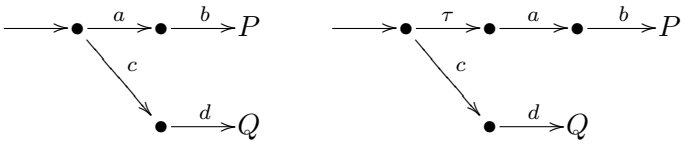
[6] Milner (1989*a*) mainly uses a variant of recursion based on process naming, but I adopt this one as it fits in better with the other calculi in this thesis. It is the equivalent of Milner's **fix**(X = E) notation.




$$P \triangleq \mu X.(a.\overline{b}.b.X + c.0)$$
$$Q \triangleq \mu X.b.\overline{b}.\overline{d}.X$$

Figure 2.3: An example of CCS process composition

to two inequivalent processes. The reason for this can be seen in the representations of the two processes below:




In the first state the second process can silently move into a state where $a$ is possible but $c$ is not possible. This move can only be matched by the first process doing an $\epsilon$ transition, i.e. an empty transition, and staying in the first state. However, since the first process in the first state can do both an $a$ and a $c$, this cannot be matched by the second process, hence the two are not weakly bisimilar.

To resolve this problem, Milner introduces *observation congruence* which fixes the relation so that empty steps are not allowed in the first state of each process. Specifically, any initial $\tau$ must be matched by at least one $\tau$, and not simply an $\epsilon$. After the first step normal weak-bisimulation is used, since only the first step can lead to a choice operator being resolved. Observation congruence is defined below:

**Definition 2.3.4** ***Observation Congruence***
*Two processes $P$ and $Q$ are **observation congruent**, written $P \cong Q$, provided:*

- *If $P \xrightarrow{\alpha} P'$ then $\exists Q'.Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \approx Q'$*
- *If $Q \xrightarrow{\alpha} Q'$ then $\exists P'.P \stackrel{\alpha}{\Rightarrow} P'$ and $P' \approx Q'$*

At first sight this looks identical to weak bisimulation, but there are two changes. Firstly $\hat{\alpha}$ has been replaced by $\alpha$ in the matching weak transition, reflecting the fact that

| | | | |
|---|---|---|---|
| Act | $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$ | Rec | $\dfrac{E\{\mu X.E/X\} \xrightarrow{\alpha} E'}{\mu X.E \xrightarrow{\alpha} E'}$ |
| Sum1 | $\dfrac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$ | Sum2 | $\dfrac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$ |
| Com1 | $\dfrac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$ | Com2 | $\dfrac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$ |
| Com3 | $\dfrac{E \xrightarrow{a} E' \quad F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$ | Res | $\dfrac{E \xrightarrow{\alpha} E'}{E \setminus a \xrightarrow{\alpha} E' \setminus a}\ \alpha \notin \{a, \overline{a}\}$ |

Table 2.1: CCS Operational Rules

empty sequences are no longer sufficient. Secondly it suffices only that the resulting processes are weakly bisimilar – we do not construct a new inductive relationship, just prepend the existing one. This relationship is indeed a congruence as proved in (Milner, 1989*a*). It therefore provides CCS with a useful behavioural equivalence which ensures that processes cannot be distinguished when placed in identical contexts, a result vital to any sort of component-based system.

Milner's later paper (Milner, 1989*b*) also gives a complete equational axiomatisation of the dynamic operators of CCS with respect to observation congruence. He also an *expansion law* – an axiom schema which can be used to aid in converting a process containing static operators into one with only dynamic operators, providing the process is finite state (but see next Section). For instance, a simple parallel composition such as $a.P \mid \overline{a}.Q$ can be rewritten as $a.(P \mid \overline{a}.Q) + \overline{a}.(a.P \mid Q) + \tau.(P \mid Q)$, and the expansion law allows every static operator to be rewritten in his way. Hence the axiomatisation of "Full CCS" with parallel composition is non-finite. It is therefore safe to say that CCS is the first *process algebra*, a phrase coined later to describe calculi which identify the fundamental properties of their operators. Indeed, the quest to understand parallel composition and the fix-point operators led directly to the next calculus I consider.

### 2.3.4 Algebra of Communicating Processes and Basic Process Algebra

Bergstra and Klop (1984) build on Milner's work by studying the operators from an algebraic perspective. ACP is described as an *alternative formulation of CCS* and this is clear from the operators chosen. Indeed, ACP is not really one calculus, but a whole range, each with different levels of expressivity. For instance the basic language ACP does not contain silent actions, which are instead found in the extension $\text{ACP}_\tau$. The equational rules of the basic form of ACP can be found in Table 2.2.

There are a number of key differences which distinguish ACP from CCS in an effort to aid axiomatisation. ACP does not have action prefix, instead opting for a general

$$
\begin{array}{rcl}
x + y & = & y + x \\
(x + y) + z & = & x + (y + z) \\
x + x & = & x \\
(x + y) \cdot z & = & (x \cdot z) + (y \cdot z) \\
(x \cdot y) \cdot z & = & x \cdot (y \cdot z) \\
\delta + x & = & x \\
\delta \cdot x & = & \delta
\end{array}
\qquad
\begin{array}{rcl}
x \parallel y & = & x \between y + y \between x + x \mid y \\
a \cdot x \between y & = & a \cdot (x \parallel y) \\
a \between y & = & a \cdot y \\
(x + y) \between z & = & (x \between z) + (y \between z) \\
a \cdot x \mid b & = & (a \mid b) \cdot x \\
a \mid b \cdot x & = & (a \mid b) \cdot x \\
a \cdot x \mid b \cdot y & = & (a \mid b) \cdot (x \parallel y) \\
(x + y) \mid z & = & x \mid z + y \mid z \\
x \mid (y + z) & = & x \mid y + x \mid z
\end{array}
$$

$$
\begin{array}{rcl}
a \mid b & = & b \mid a \\
(a \mid b) \mid c & = & a \mid (b \mid c) \\
a \mid \delta & = & \delta \\
\partial_H(a) & = & a \text{ if } a \notin H \\
\partial_H(a) & = & \delta \text{ if } a \in H \\
\partial_H(x + y) & = & \partial_H(x) + \partial_H(y) \\
\partial_H(x \cdot y) & = & \partial_H(x) \cdot \partial_H(y)
\end{array}
$$

Table 2.2: The axioms of ACP

sequential composition operator $p \cdot q$, taken from imperative programming. This leads to a simple axiomatisation, with $+$ being associative, commutative and idempotent, and $\cdot$ is associative and right-distributive over $+$. ACP also has a deadlock operator $\delta$, similar to the CCS **0** operator and representing a process with no possible transitions.

The single parallel composition operator from CCS is replaced by three separate operators, namely merge $\parallel$, left-merge $\between$ and communication merge $\mid$. Merge is essentially the same as CCS's parallel composition (though it varies based on the ACP version). Left-merge is the same except that the first action must come from the left hand-side (and thus involves no synchronisation) after which it behaves like merge. Finally communication merge requires that the first action be a synchronisation between both sides, otherwise it deadlocks.

The reason for the three merge operators is somewhat involved. CCS's algebraic theory has an awkward approach to dealing with axiomatisation of parallel composition. This operator is in general very difficult to finitely axiomatise and as a result CCS deals with the problem by the use of the expansion law. The expansion law rewrites the parallel composition operator in terms of summations by interleaving the actions and forming synchronisations from the processes. The problem is, the expansion law is not an axiom but an axiom *schema*, leading to an infinite equation system. Therefore, ACP provides merge, left-merge and communication merge, which allow a finite axiomatisation of the operator:

$$x \parallel y = x \between y + y \between x + x \mid y$$

That is, either $x$ can act first, $y$ can act first or the two can synchronise. It was later

shown by Aceto et al. (2005) that the axiomatisation (this equation plus 8 others) cannot be further simplified. Therefore direct finite axiomatisation of parallel composition seems to be impossible. Hence my work will focus on operational semantics instead of axiomatic semantics.

### 2.3.5 $\pi$-calculus

$$\pi \ ::= \ \overline{a}\langle x\rangle.\pi \mid a(x).\pi \mid \pi|\pi \mid (\nu x)\pi \mid !\pi \mid \mathbf{0}$$

Table 2.3: Syntax of $\pi$-calculus processes

$\pi$-calculus (Milner, 1999; Sangiorgi and Walker, 2001) is undoubtedly one of the most famous process calculi to date. Although not foundational to my approach, it is important particularly in the Web service world where it has been much applied. The $\pi$ calculus can be thought of as an extension to CCS, or alternatively as a relaxation of some of the restrictions on *value-passing CCS*, which in turn makes it Turing complete. $\pi$-calculus is based on the concept of *channel-passing*. Value-passing CCS is a variant of the CCS process calculus which attempts to make it more applicable to programming by allowing values to be passed by an output action of one process to the input action of another process (CCS actions are normally abstract and do not explicitly carry data). $\pi$-calculus takes this one step further by allowing actual channel names to be passed through channels. The syntax of $\pi$-calculus is shown is Table 2.3.

Some of the syntax is familiar from CCS, but there are a number of notable changes. The output and input operators are extended thus, $\overline{a}\langle x\rangle.P$ and $a(x).P$ respectively, such that the channel $a$ is used to communicate the separate channel $x$. This is formally described by the following reduction:

$$\overline{a}\langle x\rangle.P \mid a(y).Q \ \longrightarrow \ P \mid Q[x/y]$$

i.e. the channel $x$ is sent from the left process to the right process over channel $a$, and therefore the $x$ channel is substituted for the bound variable $y$ in Q. Since $Q$ now knows about the channel $x$ it can use this for communication. The semantics for $\pi$-calculus is given by a collection of reduction rules like the one above, and also a set of *structural congruence* rules – axioms which set a base equivalence class for $\pi$ processes.

The restriction operator of CCS is adapted in $\pi$ to become the $(\nu x)P$ operator (also known as "new"), which rather than simply preventing the action being observed beyond the boundary, declares that the given channel is distinct from any other $a$ channel in the process topology. This is important, since a restricted channel $x$ may now be passed beyond the restriction boundary via an unrestricted channel $a$, and the scope of channel $x$ must grow to encompass the agent on the receiving end – a concept called *scope extrusion*. It is the new restriction operator, and resulting scope extrusion, that gives $\pi$ its power (but also makes reasoning about processes difficult). To exemplify, I expand

the definition of the above process and show the reductions:

$$(\nu x)\overline{a}\langle x\rangle.x(z).P' \mid a(y).\overline{y}\langle c\rangle.Q' \longrightarrow (\nu x)(x(z).P' \mid \overline{x}\langle c\rangle.Q') \longrightarrow P'[c/z] \mid Q'$$

I assume two processes $P'$ and $Q'$ which don't contain $x$. In the first reduction the synchronisation happens as before. However, notice that initially the $x$ channel is restricted to the left process only (or rather it is a new variable of the left process). When the $x$ channel is communicated over $a$ the scope of $x$ grows to also encompass the right process – the scope has thus been "extruded". The right process can then communicate on $x$ (something impossible in CCS since restriction boundaries are fixed) and send another channel $c$ to the right process, which is thus substituted into $P'$. Since neither $P'$ nor $Q'$ contains $x$ the restriction boundary disappears at this point.

In addition $\pi$-calculus moves from *recursion* as a core concept to *replication*. The process $!P$ represents the spawning of process $P$ in parallel many times. It is described by the following structural congruence rule:

$$!P \equiv !P \mid P$$

Most work using $\pi$-calculus uses a slightly simplified variant called the *asynchronous $\pi$-calculus*. In CCS there was always an implicit notion that a synchronisation is a two-way action which allows both participants involved to progress, and this was inherited into $\pi$. However, in this asynchronous variant an output must be the last action an agent performs, meaning that all outputs are performed asynchronously and there is no way of accounting for when they have been performed. Only input actions may guard a process, e.g. $a(x).P$ is an input process and $\overline{a}x$ is an output process.

The work in this thesis does not involve $\pi$-calculus directly, but it is useful for comparison. Firstly because much of the existing work on Web services focuses on $\pi$ (see Section 2.4.2 for example), and secondly because $\pi$-calculus and related calculi are capable of describing a wide variety of systems. $\pi$-calculus has been shown to be Turing complete (Sangiorgi and Walker, 2001) and thus a great many problems can be specified. Nevertheless, the equivalence theory is also many times more complicated than that of CCS and it is far from clear if the channel passing paradigm is truly the ideal model for Web services.

## 2.4 Web services and Process Algebra

As we have seen the purpose of process algebra is to model the behaviour of entities and the communications which happen between them. It should therefore come as no surprise that process algebra, and particularly the $\pi$-calculus, has been an inspiration for the formal description of Web services. For instance the language WS-CDL (Kavantzas

et al., 2005), which allows description of the communication protocol engaged in by a group of services, draws much of its inspiration from CCS and $\pi$. Likewise the more recent versions of WS-BPEL (Jordan and Evdemon, 2007) include constructs which are directly adopted from $\pi$ such as the parallel **forAll** construct (the equivalent of $!P$). The communication model is also heavily based on link passing. It is therefore fair to say that there is a strong relationship between Web services and process algebra.

There are broadly two approaches to the application of Process Algebra to modelling Web services: a *high-level* and a *low-level* approach. The first approach involves creating a purpose-specific language for Web services and then giving a formal theory to it. Sometimes this language will have a basis in an existing process calculus, but will always contain operators specific only to their context. For instance *Orc* (Misra and Cook, 2007) is a Web scripting language with operators for sending messages to Web services along with various process algebra style combinators. Similarly *cCSP* (Butler, Hoare and Ferreira, 2005) is a language based on CSP for modelling compensable transactions. Both of these languages are purpose-built – although based on pre-existing process algebra ideas, they are tooled to a particular area.

The second approach is to use a general theory like CCS or $\pi$-calculus to model Web services in terms of well established process algebraic operators. For instance there have been a number of attempts at giving a semantics to WS-BPEL in terms of $\pi$-calculus, such as (Lucchi and Mazzara, 2007). Likewise our own work (Norton et al., 2005) giving a semantics to OWL-S directly uses a *timed process calculus* to model the various constructs of the language using a variety synchronisation patterns.

The advantage of the first approach is a language which will clearly represent the intent, whilst the advantage of the second is pre-existing formal theory and generality. In this section I will consider examples from both perspectives.

### 2.4.1 Transaction Calculi

There are a number of calculi which have been created expressly for the purpose of describing business processes. Most of these calculi are in the style of either CSP or ACP and are essentially about composing workflows and transactions with process algebraic constructs. They are therefore purpose-specific calculi, building on existing work but extending this with Web service operators.

The first offering is StAC (Chessell et al., 2002; Butler and Ferreira, 2004) (Structured Activity Compensation), a formal language for describing compensable transactions. A process in StAC can either succeed (accept), abort or trigger compensation. It includes the standard process algebra constructs such as sequence, choice, parallel and hiding, but in addition contains a number of operators for handling exceptional behaviour. Of most relevance is the compensation operator $P \div Q$, which states that $Q$ is a compensation for the process $P$. When process $P$ successfully completes, the compensation $Q$ is installed, so that if compensation is triggered it will be executed. To set the bounds of a compen-

sation StAC also has a transaction scoping operator $[P]$ which limits the compensations triggered to those inside $P$. If part of $P$ fails to execute all the compensations executed so far will be executed in the reverse order. For instance in $[P \div P' \fatsemi Q \div Q' \fatsemi \boxtimes \fatsemi R] \parallel S$, where $\boxtimes$ is a compensation trigger, $P$ then $Q$ will be executed, followed by their compensations $Q'$ then $P'$, with $S$ being executed independently since it is outside the scope.

StAC is a unique language in that, unlike the other languages we will examine, scopes do not implicitly dispose of the enclosed compensation actions upon completion. Rather, there are two explicit commands to do this, *accept* ($\boxdot$) which purges the current compensation context, and *reverse* ($\boxtimes$) which executes the current compensation context. Furthermore, in order to give StAC an operational semantics, it is extended to a more general version called StAC$_i$. Here, not only is the compensation context retained, but in addition there are multiple indexed compensation contexts. For instance $P \div_2 Q$ adds the compensation $Q$ to context 2, which will only be executed by explicitly calling it using $\boxtimes_2$. This facility makes the semantics of StAC$_i$ quite complex, and it depends on an index-to-compensation map being carried around in the operational rules. Nevertheless, this is an important feature as WS-BPEL allows compensation handlers to be called manually, and not necessarily only in reverse order (which WS-BPEL terms the "default" compensation). Thus StAC is an important development, as it is one of the few calculi which can faithfully represent the semantics of WS-BPEL (Butler, Ferreira and Ng, 2005), albeit with a complexity tradeoff.

Following the ideas presented in StAC, a further process calculus was developed called *Compensating CSP* (cCSP) (Butler, Hoare and Ferreira, 2005), based on CSP (Hoare, 1985). It includes the standard operators of CSP, and adds transaction blocks and interrupt handling, as in StAC. However, it is a much simpler language than StAC with fewer operators and greater emphasis on compositionality (it does not include multiple compensation contexts). The trace semantics of CSP is extended to completed traces with an exit status for the trace, which can either be success ($\checkmark$), failure (!) or yield (?). A compensable process is represented by a pair of traces, the first representing the behaviour and the second the accumulated compensation. When a compensable action successfully terminates it installs its compensation by prepending it onto the compensation trace, which will be run if failure occurs. Processes can be placed within a block, which defines the scope of a transaction. When a whole block completes successfully, the compensation trace is simply thrown away and the result is the successful trace. Otherwise the result is the trace with the compensation trace appended. An example of a cCSP process is shown below:

$$[A \div P; B \div Q; C \div R; THROW; D]; E; F$$

where $A, B, \cdots, P, Q \cdots \in \Sigma$ for an action alphabet $\sigma$, $A \div P$ represents $A$ with compensation $P$, ; represents sequential composition, $THROW$ throws an exception and $[P]$ represents $P$ enclosed as a transaction block. This process would have the completed

trace $(A, B, C, R, Q, P, E, F)$, since $A$, $B$ and $C$ all execute, $D$ can't because an exception is thrown first, the compensations are run in reverse order, and then $E$ and $F$ run as non-compensable processes.

Butler and Ripon (2005) further extend cCSP with a (small-step) operational semantics, for which correspondence with the trace semantics is claimed. This allows a broad framework in which cCSP can provide the basis for an executable language, given by implementing the operational rules in Prolog. Unlike StAC, cCSP represents a more conservative model of compensation than WS-BPEL allows, but one for which theoretical results come more easily.

The third and final group of compensation languages we consider are the *Sagas Calculi*, from Bruni, Melgratti and Montanari (2005). The calculus demonstrated is again quite similar to cCSP in style, but has a number of differences, other than the style of semantics. First is the method of exception handling, which relies on an execution context to define if an action is successful or not, rather than having explicit throws. Secondly the Sagas Calculi include the option of having compensations which fail, whereas in cCSP they always succeed. To indicate this the Sagas Calculi also have an abnormal termination action, along with cCSP's failure and success actions.

However, a more important difference between this and cCSP, as described by Bruni, Butler, Ferreira, Hoare, Melgratti and Montanari (2005), is the way that compensation is handled in parallel processes. In cCSP, if an exception is raised in a transaction block, all the parallel processes are interrupted and the compensations are started simultaneously. Bruni et al. identify a total of 4 compensation strategies for parallel flows, namely:

1. *No interruption and centralised compensation*, where all parallel processes execute to completion and only then does compensation occur;

2. *No interruption and distributed compensation*, where parallel processes perform compensation independently and when required;

3. *Coordinated interruption*, where parallel processes are interrupted synchronously when compensation is required, and then compensated centrally;

4. *Distributed interruption*, where parallel processes are interrupted asynchronously when compensation is required, and then compensated independently.

The form of compensation in cCSP is identified as *coordinated interruption*. The Sagas calculi has two semantics for parallel compensation; the *naïve* semantics and the *revised* semantics. The former simply allows all parallel processes to continue even after an exception is raised, triggering their compensations after completion, and corresponding to *no interruption and distributed compensation*. In contrast, the revised semantics though have extra signals to force interruption of parallel sagas, though unlike in cCSP the compensations are started when they are ready to run, rather than at the same time. This is identified as *distributed interruption*.

### 2.4.2 $\pi$-calculus based calculi

Several authors have created calculi for representing compensation within a $\pi$-calculus setting. These calculi usually have all the features of the asynchronous $\pi$-calculus, but in addition have some sort of transaction operators. Due to the powerful nature of $\pi$, authors usually provide a mapping from their extension back into the basic calculus. In some respects this work is close to my own because of the CCS link, but on the other hand this work is noticeably different due to $\pi$-calculus's very different way of handling concepts like recursion. Nevertheless, these calculi are much closer to being fully abstract calculi than the purpose-specific transaction languages described in the previous section.

Bocchi et al. (2003) present a calculus called $\pi$t for describing transactional Web services based on the $\pi$-calculus. Unlike cCSP and Sagas, $\pi$t is a fully fledged communicating process calculus with all the features of $\pi$-calculus, augmented with transaction and abort operators. A transaction is a four-tuple $\mathrm{t}(P, P, P, P)$ consisting of the forward flow process, a process to be executed after a failure occurs, a "failure bag" consisting of all the compensation processes accumulated so far, and a compensation process to be executed should the enclosing transaction fail. Aside from the additional complexities of $\pi$-calculus, a major problem of the compensation method in this calculus is that the order of compensations is not preserved. All compensations are simply executed in parallel when a failure occurs.

Laneve and Zavattaro (2005) improve on this idea with web$\pi$, an extension of $\pi$-calculus with a more flexible transaction operator $\langle\!|P; Q|\!\rangle_x^n$, but in a timed setting. $P$ is the forward flow of the transaction and $Q$ is the compensation flow. The transaction operator is parametrised over two variables: $x$, the channel on which exceptions can be raised to trigger compensation, and $n$, a natural number denoting the time-out remaining for the transaction. Every transition (or rather reduction) originating from inside the transaction causes this time remaining to decrease by 1. If the time remaining reaches 0 and all nested transactions have been exhausted, compensation can take place. This is effectively a highly bespoke form of maximal progress, where the specified number of transactions must complete before compensation can be triggered. Lucchi and Mazzara (2007) use a non-timed fragment of web$\pi$ called web$\pi_\infty$ to give a partial formal semantics to BPEL 1.1. However, as is the case with Bocchi et al. (2003), this calculus does not take any account of compensation ordering.

To conclude, the $\pi$-calculus is probably the most well used calculus for modelling service composition, particularly where WS-BPEL is involved. It clearly has a close relationship with Web-based systems because of its inherent dynamism. Furtheremore, with a small extension it can be used to describe compensable transactions. However, it is still questionable as to how much verification can be performed on the resulting model due to its complexity. Hence in the next section I introduce another branch of process calculi which also has a great deal of potential in this area.

## 2.5 Timed Process Calculi

*Timed Process Calculi* are increasingly used, particularly in the Web service world (see for instance Laneve and Zavattaro (2005); Bravetti and Zavattaro (2007); Wong and Gibbons (2008)), for reasoning about processes with a temporal dimension. A Timed Process Calculus augments the foundational concept of abstract processe with some way of measuring time. The best way to quantify time is subject to much debate, and different kinds of problems undoubtedly require different ways of approaching this question. Broadly, there are two paradigms for timed process calculi:

- **Real-time** process calculi, which add some sort of physical time passage to the transition system;
- **Abstract-time** process calculi, which take a more logical approach to time, trying to seek the high-level concepts which time can address.

The former were the first considered by researchers, and in many ways represent a more natural paradigm; their development also laid much of the groundwork which would later be used for the latter. The abstract time paradigm highlights the fundamental properties of discrete real-time and uses these to construct a process calculus which exhibits a more general form of timed action. As we shall see, this abstract action (called the *clock*) provides something very different from real-time modelling, in that it draws out some of the fundamental concepts needed for component modelling. Indeed, the approach I take in this thesis to Web service modelling is very much of a *low-level* variety of process calculus (See section 2.4). I will not be creating a Web service calculus as such; rather I will use a timed process calculus to model the fundamental synchronisation patterns of business processes.

I now consider both of the paradigms of timed process calculi in turn.

### 2.5.1 Temporal CCS and discrete real-time process calculi

$$P ::= \mathbf{0} \mid X \mid a.P \mid (t).P \mid \delta.P \mid P \oplus P \mid P + P \mid P|P \mid P \setminus a \mid P[a \mapsto a] \mid \mu X.P$$

Table 2.4: TCCS Syntax

We proceed with our brief study of real-time process calculi by looking at one of the foremost of these calculi, *Temporal CCS* (Moller and Tofts, 1990) (TCCS). Although not the first and by no means the only real-time process calculus it crystallises many of the key ideas of this field. Its syntax is shown in Table 2.4. TCCS is a conservative extension of CCS which adds the concept of a process waiting for time to pass before proceeding with its next action. It does this via a timed prefix operator $(t).P$, where $t$ is the period of time which must elapse in the environment before the remaining behaviour of $P$ can proceed. For instance:

$$(3).\overline{a}.P \overset{1}{\rightsquigarrow} (2).\overline{a}.P \overset{1}{\rightsquigarrow} (1).\overline{a}.P \overset{1}{\rightsquigarrow} \overline{a}.P \xrightarrow{\overline{a}} P$$

This process represents the passage of (discrete) time one unit at a time. Time transitions in TCCS are represented by wiggley arrows, whilst regular CCS actions are represented by straight arrows. The time transitions can in reality be any time period remaining, such as 2 or 3 in this case. Parallel processes all move forward at the same speed with respect to time, as the illustration of process $(3).P \mid (2).a.(1).Q$ below demonstrates:

$$(3).P \overset{2}{\rightsquigarrow} (1).P - - - (1).P \overset{1}{\rightsquigarrow} P$$

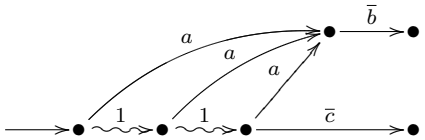
$$(2).a.(1).Q \overset{2}{\rightsquigarrow} a.(1).Q \xrightarrow{a} (1).Q \overset{1}{\rightsquigarrow} Q$$

Initially only 2 time units can elapse, and therefore the former process has to wait for the second process to perform its $a$ action before continuing. The reason for this is that the regular $\alpha$ actions of CCS are taken to be *insistent* (also called *urgent*) by default – an unguarded insistent action will not permit time to progress further before the given action is performed. Specifically, processes such as $a.R$ do not admit time transitions, the $a$ action always *pre-empts* the clock action.

Central to this axiom is the assumption that all actions in CCS are *instantaneous*, since their occurrence does not initiate a passage of time. Time may only pass between two observable events; this is central to interleaving semantics as a whole in that actions are all-or-nothing. Therefore, TCCS also includes a delay prefix operator $\delta.P$ which describes a process which will *patiently* (and non-deterministically) wait any period of time before $P$ can proceed. The actions of $P$ are all enabled but time passage is also permitted. Therefore a process such as $\delta.a.Q$ can either perform $a$ or allow time to elapse. A process is "patient" if it allows time to pass, and insistent if it will not. Time in TCCS is therefore *relative* – it passes at the rate that the processes will allow.

These principles are found in many areas of Computer Science, particularly including concurrent programming languages where it is known as the *two-phase functioning* approach, described by Nicollin et al. (1993). In the first phase elementary tasks which are assumed to be atomic and therefore instantaneous (relatively speaking) are required to execute. In the second phase when no more atomic tasks remain and all processes are in agreement that time may pass, the clock moves time forward one step. When only silent actions are considered instantaneous, and regular actions such as $a$ are thus *patient*, this is known as the *maximal progress* assumption (Yi, 1991), which I will further describe in Section 2.5.2.

Using the operators of TCCS we can build a basic *timeout* structure, $\delta.P + (t).Q$ which can proceed with $P$ during the period before $t$ has elapsed and can then timeout to $Q$. For instance with $P \triangleq a.\overline{b}.\mathbf{0}$ and $Q \triangleq \overline{c}.\mathbf{0}$ and $t = 2$ the following transition system is admitted:




This process waits for an $a$ until 2 time units have elapsed. At this point $\overline{c}$ becomes enabled as well. This may represent, for instance, a timeout on receiving a message. If a parallel process can provide an $a$ before 2 time units have elapsed the clock will be held up due to the maximal progress assumption and the synchronisation on $a$ will occur.

The actual sort of $t$ can vary depending on the application, though Moller and Tofts (1990) use a discrete sort for simplicity, resulting in *discrete time*. If by contrast the sort of $t$ were to be the positive rationals or reals, the result would be *dense time*, where any two events are separated by an infinite number of occurrences. Whilst it is generally possible to finitely axiomatise discrete time calculi, such is not always the case with dense time.

Time in TCCS is inherently *deterministic* – it is impossible for two different states to result from an identical time passage. This leads to an important axiom of timed process calculi:

**Definition 2.5.1** ***Time determinism***
*For any process $P$ and time period $\sigma$, if there exists processes $P'$ and $P''$ such that $P \xrightarrow{\sigma} P'$ and $P \xrightarrow{\sigma} P''$ then $P' \equiv P''$.*

Specifically, it is impossible that an identical passage of time could lead a process to two different states. This provides a particular challenge when dealing with the CCS choice operator, and thus there are two such operators in TCCS: $+$ and $\oplus$. The first operator $+$ requires that *both* sides proceed when time passes without reducing the operator. Only once one of the two processes can perform an action will the choice be resolved. The second operator $\oplus$ can resolve the choice if one side requires more time to pass that the other. In this instance the slower of the two processes will be chosen, for instance in $(1).P \oplus (2).Q$ the right-hand process is chosen. Neither of these operators violates time determinacy, and as we shall see this is one of the key assumptions behind most timed process algebras.

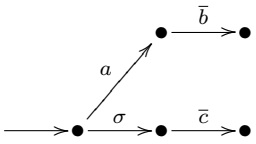
TCCS provides an excellent insight into the area of real-time process algebra, and although I will not be utilising it directly, many of the principles carry over into the area of abstract time.

### 2.5.2 Temporal Process Language and abstract time

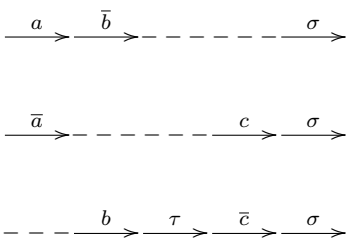
$$P ::= \mathbf{0} \mid \Omega \mid X \mid \sigma.P \mid \lfloor P \rfloor(P) \mid a.P \mid \tau.P \mid P + P \mid P|P \mid P[a \mapsto a] \mid P \setminus a \mid \mu X.P$$

Table 2.5: Syntax of TPL

TPL (Hennessy, 1993; Hennessy and Regan, 1995) can be considered one of the first process calculi to formalise the idea of *abstract time*. Like TCCS, it is a conservative extension of CCS. TPL's syntax can be found in Table 2.4.[7] Its main addition is a binary timeout operator $\lfloor P \rfloor(Q)$ which means, in essence, if the process $P$ cannot perform any internal activity then a timeout to $Q$ is possible, similar to the timeout demonstrated in the previous section. For instance the process $\lfloor a.\overline{b}.\mathbf{0} \rfloor(\overline{c}.\mathbf{0})$ has the following transition system:




Unlike in TCCS, this "timeout" is not a quantified timeout, in that it isn't associated with a particular time limit, but rather refers to a point in the system at which every process should move onward to a new phase in the time-line. Thus the "clock" $\sigma$ in TPL is a *multi-party synchronisation* action – every process in the system must have ceased its activity before the clock can tick. This is achieved by the *maximal progress* assumption, which states that the passage of time can only be observed provided every process in a system has progressed maximally, i.e. completed its internal activity, and thus agrees that time may advance. This is illustrated below:




Three agents engage in a simple conversation, but as soon as all three have finished and thus no more possible $\tau$ transitions remain the clock ticks, indicated by $\sigma$. Specifically, the clock can tick because all three agents have progressed maximally. If we extend CCS's action sort with an action $\sigma$ representing the passage of time, then the maximal progress assumption, in the context of TPL, can be defined formally as follows:

**Definition 2.5.2** ***Maximal Progress***
*For any process $P$, if there exists a process $P'$ such that $P \xrightarrow{\sigma} P'$ then there does not exists a $P'$ such that $P \xrightarrow{\tau} P'$ (also written as $P \xrightarrow{\tau}\!\!\!\!\!\rightarrow$).*

That is, the passage of time and presence of executable silent actions are mutually exclusive. Unlike in TCCS, visible actions like $a$ are not instantaneous but *patient*, meaning that they allow time to pass. Only silent actions are instantaneous, and as a result there is no need for TCCS's $\delta$ operator, since all processes are patient by default, e.g. $a.P \xrightarrow{\sigma} a.P$.

[7] Hennessy actually presents a slightly different syntax to this, but the operators here are the same and I wish to retain uniformity between the different calculi.

$$\mathcal{E} \quad ::= \quad \mathbf{0} \mid \alpha.\mathcal{E} \mid \lfloor\mathcal{E}\rfloor\sigma(\mathcal{E}) \mid \lceil\mathcal{E}\rceil\sigma(\mathcal{E}) \mid \mathcal{E}+\mathcal{E} \mid \mathcal{E}|\mathcal{E} \mid \mathcal{E}\setminus a \mid \mathcal{E}/\sigma \mid \Delta \mid \Delta_\sigma \mid \mu X.\mathcal{E} \mid X$$

Table 2.6: Syntax of CaSE

### 2.5.3 Multiple Clock Calculi and CaSE

TPL and single-clock calculi like it are somewhat limited in the sort of systems they can describe. Effectively TPL is a generalisation of TCCS in that there is a single global (and thus) absolute clock which the whole system uses to observe time passing. Indeed, it is possible to embed the majority of the operators of TCCS into TPL. Much work has been done to take relative time one step further. *Multiple Clock* Timed Process Calculi, rather than having a single global clock which measures time at the same speed for every agent, allows a system to be governed by several independent clocks $\sigma, \rho, \cdots \in \mathcal{T}$, for some clock sort $\mathcal{T}$. Clock ticks are relative to the actions of the system and to each other, and do not necessarily possess an absolute measure, being more logical in nature. The main advantage of multiple clocks rather than a single unified clock is that a system can be *componentised*. Each component can then measure time relative to itself without the need for reference to the global system, which enables compositional modelling.

CaSE(Norton et al., 2003), the *Calculus for Synchrony and Encapsulation*, is an abstract time process calculus with multiple clocks. Along with TPL, it is based on two previous multi-clock calculi called PMC (Processes with Multiple Clocks) (Andersen and Mendler, 1994) and CSA (Calculus of Synchrony and Asynchrony) (Cleaveland et al., 1997). It is a conservative extension of CCS with a timeout operator and it adheres to the maximal progress assumption. The complete syntax for CaSE can be found in Table 2.6.

CaSE retains the timeout operator from TPL, but parametrises it over the clock over which the timeout is made, i.e. $\lfloor P\rfloor\sigma(Q) \xrightarrow{\sigma} Q$ and $\lfloor R\rfloor\rho(S) \xrightarrow{\rho} S$. However, since there are multiple clocks it is possible for them to interact in different ways and therefore CaSE has two timeout operators, both of which are parametrised over the clock:

- **Fragile timeout**, $\lfloor E\rfloor\sigma(F)$ which reduces to $F$ when $\sigma$ ticks, or to $E'$ whenever $E$ performs any other action, i.e. $E \xrightarrow{\alpha} E'$;
- **Stable timeout**, $\lceil E\rceil\sigma(F)$ which is the same as fragile timeout, except that if $E$ performs a clock tick then the timeout operator does not reduce.

Fragile timeout can therefore be seen as a timeout where another clock can interrupt the timeout clock, whereas in stable timeout another clock cannot directly interrupt the timeout clock.

Perhaps the most important addition CaSE introduces is the idea of clock *hiding*, an idea first experimented with by Lüttgen (1998) and further by Kick (1999). Hiding a clock, written as $P/\sigma$, creates a boundary for $\sigma$ in that processes composed after $\sigma$ is hidden cannot detect it ticking. Hiding causes all transitions on $\sigma$ from $P$ to be replaced

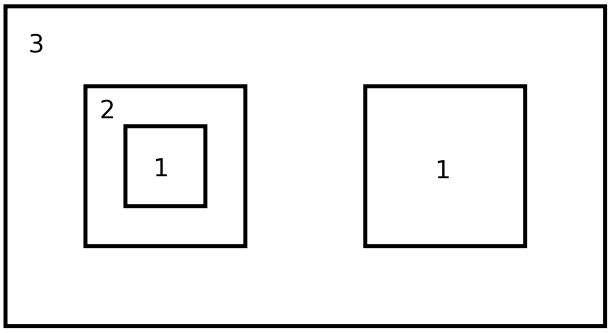



Figure 2.4: Synchronous Hierarchies
(Lower numbered compartments have higher priority.)

by silent actions, in a similar way to CSP's hiding operator. Since maximal progress is global, this causes all other clocks to be held up whenever the hidden clock can tick. Thus hiding a clock effectively *prioritises* it, and the order in which clocks are hidden defines the order in which clocks are prioritised. For instance whenever $P \xrightarrow{\sigma} P'$, then $P/\sigma \xrightarrow{\tau} P'/\sigma$ and therefore any clocks hidden after this will always be prevented from ticking by $\sigma$. Nested hiding of the same clock (e.g. $P/\sigma/\sigma$) has no effect in CaSE, as a clock once hidden is removed permanently from use.

Hiding leads to a further key concept in this process calculus, called *synchronous hierarchies*, which is central to component modelling. The idea is that a component is itself a collection of components, and will be modelled in CaSE via a hierarchy of clock regions delimited by the hiding operator. Each component will have a collection of clocks to guide its execution, for example to indicate when the component has finished executing. Until this clock for each sub-component has ticked, and thus exhausted its silent actions, the clock of the higher-level is held up. All sub-components must therefore complete before the parent component, since their clocks must tick first. This is illustrated by Figure 2.4 where a hierarchy of processes is given an implicit execution order based on their nesting. This introduces a fundamental dependency relation and gives CaSE its power.

Furthermore, CaSE has two additional *urgency operators*, $\Delta$ and $\Delta_\sigma$, which allow even more options and aid in axiomatisation. These two operators allow all clocks or a given clock to be held up, respectively. Specifically, $\forall\sigma.\nexists P'.\Delta \xrightarrow{\sigma} P'$ for any $\sigma$, and $\nexists P'.\Delta_\rho \xrightarrow{\rho} P'$. These operators can be summed with a process to make it insistent, or they can be used to stop one clock from ticking and thus allow another clock in the hierarchy (which would otherwise be held up) to tick. For instance the process $a.E$ is patient on every clock, but by summing it with a $\Delta$ – i.e. $a.E + \Delta$ – it becomes insistent, preventing time from passing until $a$ has occurred (as in TCCS). For the difference between insistent and patient prefix see Figure 2.5 – basically a patient process has self-transitions on all clocks in $\mathcal{T}$ for there are no a timeouts present.

The $\Delta$ operators also make it impossible to decide which clocks can tick in a process simply based on the non-temporal transitions it can do, since the presence of any $\Delta$

$$\text{Idle}\ \frac{}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}} \qquad \text{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \text{Patient}\ \frac{}{a.E \xrightarrow{\sigma} a.E} \qquad \text{Stall}\ \frac{}{\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma}\,1$$

$$\text{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \text{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'} \qquad \text{Sum3}\ \frac{E \xrightarrow{\sigma} E' \quad F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'}$$

$$\text{Com1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \text{Com2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \text{Com3}\ \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{Com4}\ \frac{E \xrightarrow{\sigma} E' \quad F \xrightarrow{\sigma} F' \quad E \mid F \not\xrightarrow{\tau}}{E \mid F \xrightarrow{\sigma} E' \mid F'} \qquad \text{Res}\ \frac{E \xrightarrow{\gamma} E'}{E \setminus a \xrightarrow{\gamma} E' \setminus a}\,2$$

$$\text{TO1}\ \frac{E \not\xrightarrow{\tau}}{\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F} \qquad \text{TO2}\ \frac{E \xrightarrow{\gamma} E'}{\lfloor E \rfloor \sigma(F) \xrightarrow{\gamma} E'}\,3 \qquad \text{Rec}\ \frac{E \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'\{\mu X.E/X\}}$$

$$\text{STO1}\ \frac{E \not\xrightarrow{\tau}}{\lceil E \rceil \sigma(F) \xrightarrow{\sigma} F} \qquad \text{STO2}\ \frac{E \xrightarrow{\alpha} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\alpha} E'}\,4 \qquad \text{STO3}\ \frac{E \xrightarrow{\rho} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\rho} \lceil E' \rceil \sigma(F)}\,1$$

$$\text{Hid1}\ \frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma} \qquad \text{Hid2}\ \frac{E \xrightarrow{\alpha} E'}{E/\sigma \xrightarrow{\alpha} E'/\sigma} \qquad \text{Hid3}\ \frac{E \xrightarrow{\rho} E' \quad E \not\xrightarrow{\sigma}}{E/\sigma \xrightarrow{\rho} E'/\sigma}\,1$$

where: 1) $\rho \neq \sigma$ 2) $\gamma \notin \{a, \overline{a}\}$ 3) $\gamma \neq \sigma$ 4) $\alpha \neq \sigma$

Table 2.7: CaSE Operational Semantics

operators must be taken into account. In particular $P \not\xrightarrow{\tau} \implies P \xrightarrow{\sigma}$ does not necessarily hold.

The CaSE semantics are detailed in several papers, firstly the main paper at CONCUR '03 (Norton et al., 2003) and two workshop papers (Norton and Fairtlough, 2004; Norton, 2005*a*), the latter of which gives the semantics shown in Table 2.7 (though without the labels). Note that these semantics use the notation $P \not\xrightarrow{\gamma}$ as a shorthand for $\nexists P'.P \xrightarrow{\gamma} P'$ – i.e. that $P$ is incapable of performing a $\gamma$. CaSE's actions are drawn from two sorts, $(\alpha \in)\mathscr{A}$ which contains the normal actions of CCS and $\mathcal{T}$ which contains all the clock action $\sigma, \rho, \cdots$. Therefore, the LTS for CaSE is of the form $(\mathcal{E}, \mathscr{A} \cup \mathcal{T}, \rightarrow)$ where $\rightarrow \subseteq \mathcal{E} \times \mathscr{A} \cup \mathcal{T} \times \mathcal{E}$ is the least relation closed under the rules of Table 2.7. The symbol $\gamma$ refers to an action drawn from either sort (i.e. $\mathscr{A} \cup \mathcal{T}$). Many rules in the semantics are familiar from CCS as this is a conservative extension.

First note the way that clock ticks interact with choice in rule Sum3: a clock ticking cannot decide a choice, but rather, as in TCCS, both sides must advance for the clock to tick. This is required because CaSE also adheres to *time determinancy*. For instance in a process $\lfloor P \rfloor \sigma(Q) + \lfloor R \rfloor \sigma(S)$ both sides of the choice must tick on $\sigma$, or an action of either $P$ or $R$ must occur. If $\sigma$ does tick the resulting process is $Q + S$. The rule for

| Action Prefix | | Clock Prefix | |
|---|---|---|---|
| **Insistent** | **Patient** | **Insistent** | **Patient** |
| $\underline{a}.P \xrightarrow{a} P$ | $\overset{\sigma}{\circlearrowleft}\ a.P \xrightarrow{a} P$ | $\underline{\sigma}.P \xrightarrow{\sigma} P$ | $\overset{\rho}{\circlearrowleft}\ \sigma.P \xrightarrow{\sigma} P$ |

Figure 2.5: Insistent vs. Patient Prefix

parallel composition (Com4) simply states that, provided both sides can advance on a $\sigma$ and $E \mid F$ cannot perform a $\tau$ (i.e. they cannot synchronise), then $E \mid F$ can advance on $\sigma$. Naturally this inductively allows any number of parallel processes to synchronise on $\sigma$ provided a $\tau$ transition is not present anywhere.

The semantics of the fragile timeout is given by two rules, TO1 and TO2. The first rule states that the timeout to reduce by doing a $\sigma$ provided the LHS process $E$ cannot perform a $\tau$ action (recall that all the actions in $E$ are enabled in a timeout). The second rule allows any action, including clocks other than $\sigma$, from the left-hand side can reduce the timeout operator, and hence disable $\sigma$.

The semantics of stable timeout is similar, except it has three rules, STO1-STO3. The former two are essentially the same as the two rules of fragile timeout, except that the latter only allow non-clock actions to reduce the operator. The third rule STO3 allows any clock other than $\sigma$ on the LHS to tick, but instead of reducing the timeout operator, it is retained. Stable timeout is therefore static with respect to clocks other than $\sigma$. Only a non-clock action on the LHS can reduce it. This is useful for several reasons, including that it allows the specification of a *patient clock prefix* operator, i.e. $\sigma.E \triangleq \lceil \mathbf{0} \rceil \sigma(E)$. Since $\mathbf{0}$ allows any clock to tick, placing it on the LHS of a fragile timeout operator would cause it to reduce immediately. However, placing $\mathbf{0}$ on the LHS of a stable timeout operator leads to a construct which will always allow any clock to tick, and therefore can only reduce by timing out on $\sigma$. Without stable timeout it would only be possible to build an *insistent* clock prefix operator – i.e. $\underline{\sigma}.E \triangleq \lfloor \Delta \rfloor \sigma(E)$ – whose sole transition is $\sigma$ and therefore holds up all other clocks. The difference between patient and insistent clock prefix is shown in Figure 2.5 – as for action prefix, a patient clock prefix simply means every clock without a timeout has a self-transition.

The final three rules, Hid1-Hid3 provide the semantics for the hiding operator. The first states that if a process $E$ can evolve into $E'$ by doing a $\sigma$ then when $\sigma$ is hidden, it should be instead converted to a $\tau$ action. The second rule simply allows all non-clock actions through without change. The last rule allows any clock other than $\sigma$ to tick across the hiding operator, but only provided $E$ cannot do a $\sigma$. If $E$ can do a $\sigma$ then clearly a $\tau$ is possible in $E/\sigma$ and all other clocks should be held up.

As for CCS, CaSE's equivalence theory is based on *weak bisimulation* which is restated in this way[8]:

[8]The statement "a symmetric relation $\mathcal{R}$" means that all clauses of the definition apply in both directions for each pair. It removes the need to write down each clause twice.

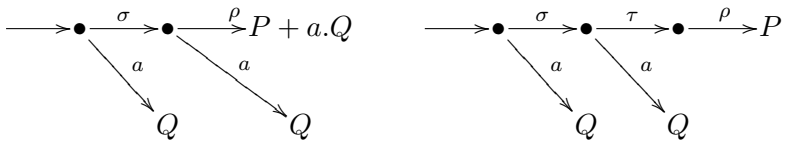



Figure 2.6: Example of why Temporal Weak Bisimulation is not a congruence

**Definition 2.5.3** ***Temporal Weak Bisimulation***
*A symmetric relation $\mathcal{R}$ is a temporal weak bisimulation provided that for all $\langle P, Q\rangle \in \mathcal{R}$:*

- *If $P \xrightarrow{\gamma} P'$ then $\exists Q'.Q \stackrel{\hat{\gamma}}{\Rightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$*

*We write $P \approx Q$ when there exists a temporal weak bisimulation $\mathcal{R}$ such that $\langle P, Q\rangle \in \mathcal{R}$.*

Naturally enough this is not a congruence relation as indeed it isn't for CCS because of the initial silent action and summation issue described in Section 2.3.3. Now, however, there is also an additional problem with summation and additionally timeout, since, as per rule Sum3, the operator is *static* with respect to clock ticks, e.g. $\sigma.P + a.Q \xrightarrow{\sigma} P + a.Q$. As we know a silent action is capable of resolving a choice. Therefore even though $\sigma.\rho.P \approx \sigma.\tau.\rho.P$, the two can be distinguished by summing with $a.Q$ as illustrated in Figure 2.6, since the $\tau$ resolves the choice before $\rho$ occurs in the latter. Since this is more than simply a problem with the first transition, it is necessary to construct a new relation which is called *Temporal Observation Congruence*.

**Definition 2.5.4** ***Temporal Observation Congruence***
*A symmetric relation $\mathcal{R}$ is a temporal observation congruence provided that for all $\langle P, Q\rangle \in \mathcal{R}$:*

- *If $P \xrightarrow{\alpha} P'$ then $\exists Q'.Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P \approx Q$;*
- *If $P \xrightarrow{\sigma} P'$ then $\exists Q'.Q \xrightarrow{\sigma} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.*

*We write $P \approxeq Q$ when there exists a temporal observation congruence $\mathcal{R}$ such that $\langle P, Q\rangle \in \mathcal{R}$.*

The first part is identical to the clauses in Milner's version, it applies only to non-clock action and forces a $\tau$ to be matched by at least one $\tau$, after which temporal weak bisimulation can be followed as before. If a clock transition occurs, then this is *strongly* matched to prevent any clock sequence being broken by $\tau$ transitions. Therefore this definition forces any initial sequence of clock to be strongly matched, and only after an action occurs does it revert to temporal weak bisimulation. Hence this is a congruence relation as demonstrated by Norton et al. (2003). Temporal Observation Congruence has also been finitely axiomatised (Norton, 2005*a*) and uses an enhanced version of Milner's expansion law.

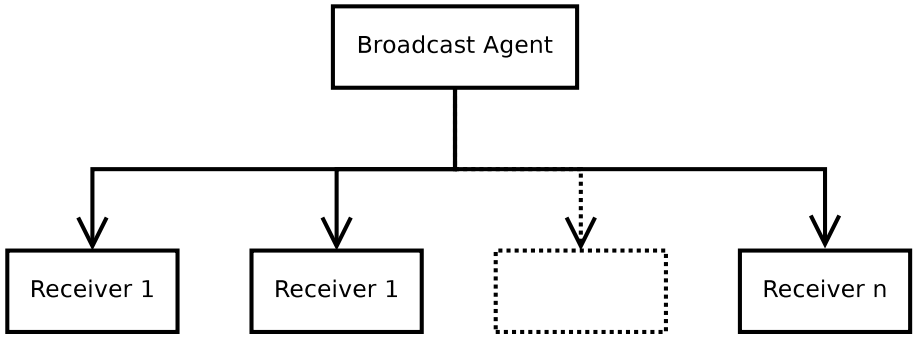



Figure 2.7: Isochronic Broadcast

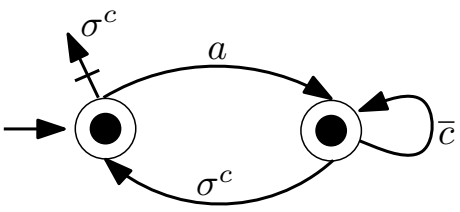
**Component Modelling in CaSE**

Having examined CaSE's theory I now turn attention to looking at its use in Web service modelling. CaSE, like CCS before it, lends itself well to component modelling because of its hierarchical nature. Components can be specified which are entirely abstract, with only input and output actions visible to the environment. Furthermore global maximal progress, as we have seen, induces an implicit dependence relationship between a whole component and its sub-components. Nevertheless, multi-clock abstract timed process calculus can also provide a number of other important features.

One of CaSE's main applications prior to its use in Web service modelling was *dataflow* (Norton and Fairtlough, 2004). Compositional dataflow modelling in CaSE is achieved primarily through a construct called *isochronic broadcast*. This is a *deterministic* form of broadcast which uses a clock to decide when an output has been passed on to every member of an unbounded set of recipients, as illustrated in Figure 2.7. This contrasts with CSP's form of broadcast which is non-deterministic and passes on data to any subset of the waiting receivers. CCS cannot represent this construct for an unbounded set of receivers – prior knowledge is required – but in CaSE a component's dataflow structure need not alter if sub-components are added or removed.

A transition system showing a basic broadcast agent is shown in Figure 2.8. Here an input $a$ is received in the first state, and the broadcast clock $\sigma^c$ is held up. In the second state $c$ is output multiple times. The assumption is that each agent receiving $c$ is within the hiding boundary of $\sigma^c$. This being the case, $\sigma^c$ will only tick once all agents who can receive a $c$ have done so. It then returns to the first state, so as to receive another input to pass on. This simple construction makes compositional representation of dataflow connections possible.

Because of its clear links to component modelling, CaSE has been used to give a compositional operational semantics to the OWL-S semantic web service composition language (Norton et al., 2005). The original application of CaSE was to specification of composite *Digital Signal Processors* (DSPs), imperative transformers of digital signals. The underlying idea is that applications for designing and implementing such systems must allow rapid changes and verification of updated specifications. The advantage




$$\mu X.a.\mu Y.\lfloor c.Y \rfloor \sigma(X) + \Delta_\sigma$$

Figure 2.8: Isochronic Broadcast Agent

of CaSE is its flexibility in creating a compositional semantics for a language, so that model updates need only touch the relevant parts. Temporal Observation Congruence also enables process decomposition, and thus component substitution at the language level. Service oriented applications also require rapid updates as the Web is by nature highly dynamic, and therefore CaSE seems to be a natural fit.

The CaSE OWL-S semantics works on the basis of a synchronous *protocol* which is used to orchestrate execution of the control-flow. Each OWL-S process is represented as a composition of sequential CaSE processes, composed with a series of *schedulers* which administer execution. Each process has a precondition for execution, which usually involves receiving inputs from the environment. The resulting processes can then be recursively composed with other schedulers for different control-flow patterns. The structure of a sequential workflow with schedulers is shown in Figure 2.9, which represents the sequential composition of three schedulers $p_1 \cdots p_3$. Each component abstraction (*performance* in OWL-S) is composed with a scheduler which informs it of when it can execute via a number of channels which will be described shortly. Schedulers for sequential composition are shown in Figure 2.10.

In this semantics all patterns are list based, and therefore have two schedulers, one for the head process and one for each of others (the tail). The process on the left orchestrates the head process, whilst the one on the right orchestrates the tail processes. The precondition for readiness of a sequential composition is simply readiness of the head process, since the head process may inductively satisfy the tail process and so on.

The $r$ channel stands for *ready*, the $e$ channel stands for *execute* and the $t$ channel is a control *token* (as in token ring networks). The $i$ subscript is used to differentiate the channels of the internal process being scheduled from the environment which also follows the scheduling protocol. In this case, the head scheduler first waits for the composed process to indicate readiness via communication on channel $r^i$. The readiness signal is then passed on to the environment via $r$, which replies with permission to execute via $e$. Permission to execute is then passed onto the enclosed process with $e^i$. Completion of this execution is indicated by the cessation of activity from the process (internal activity indicated by the dashed transition), and detected by the ticking of clock $\sigma^n$ (where $n$ is the name of the process in question). As shown in Figure 2.9 each performance has a dedicated clock, $\sigma^{p_1} \cdots \sigma^{p_3}$, for this purpose. The scheduler then outputs on a $t^i$ channel

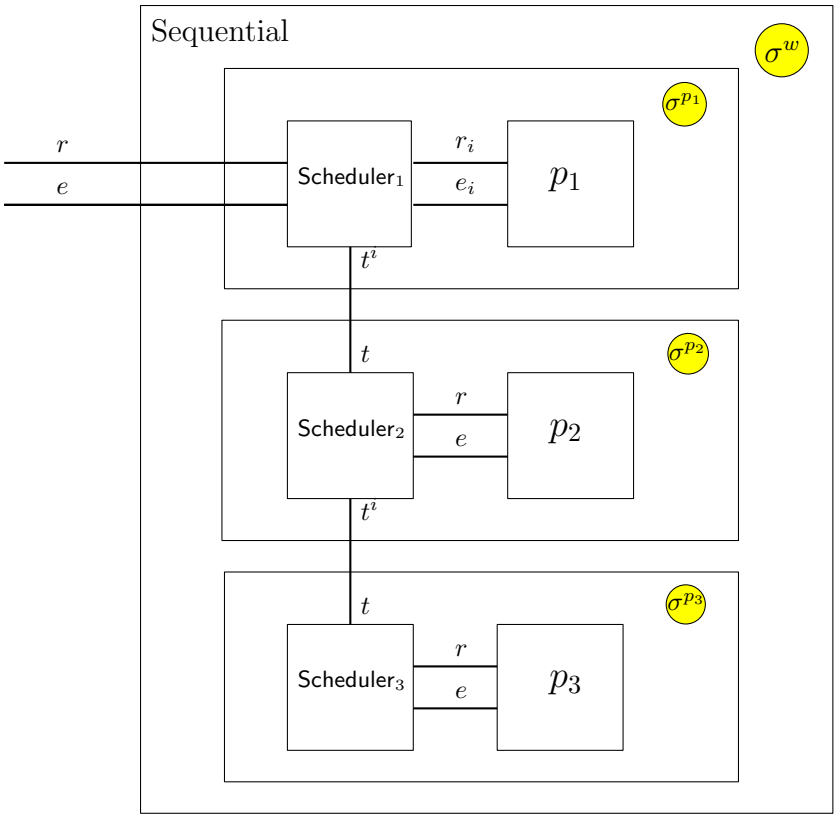
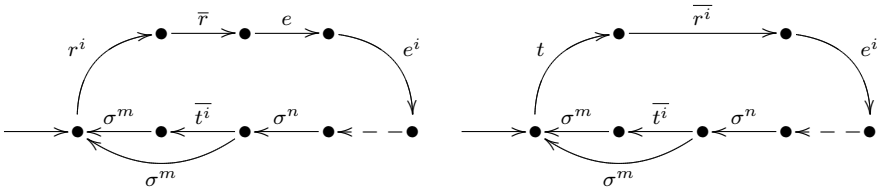



Figure 2.9: Sequential Workflow composition




$\mu X.r^i.\overline{r}.e.\overline{e^i}.\sigma^n.\lfloor\overline{t^i}.\sigma^m.X\rfloor\sigma^m(X)$ $\mu X.t.r^i.\overline{e^i}.\sigma^n.\lfloor\overline{t^i}.\sigma^m.X\rfloor\sigma^m(X)$

Figure 2.10: Transition diagrams for sequential schedulers

(if possible), which passes control onto the next scheduler in the sequence.

The tail process is very similar except it requires a token from the previous scheduler before it can execute its respective process. It does not communicate with the environment as permission for the whole sequence to execute has already been secured. Once it has received the token it proceeds to execute its process in the same way as the head scheduler. Completion of all processes is detected by the ticking of the last clock $\sigma^m$ ($m$ is the name of the sequential workflow, $w$ in Figure 2.9), which synchronises all the schedulers and indicates overall completion.

The complexity of this semantics is primarily caused by the need to handle preconditions, and the need to retain compositionality. In this semantics it is wholly possible to add or remove processes in the sequence without a changes to others, specifically through compositionality of parallel composition. Furthermore, sequential composition does not really do this semantic system justice. A better example is the OWL-S any-order construct, which allows the execution order of a list of components to be determined purely by their preconditions. The scheduler in this case is almost identical, except that, where in sequential composition the $t$ channel is restricted to passage between two components, in any-order the $t$ can be sent to any waiting scheduler.

### 2.5.4 Calculus of Broadcasting Systems

Although not strictly speaking a timed process calculus, CBS (Prasad, 1991, 1995) is nevertheless relevant to this study. It has a similar set of operators to CCS, but has a very different model of communication to the two party "hand-shake". Like CaSE and other abstract time process calculi, CBS focuses on multi-party synchronisation. However, in CBS communications take the form of one-to-many broadcasts. The idea is that one process "speaks" by a $w!$ action, and several processes "hear" by $w?$, in a similar way to CaSE's isochronic broadcast. The basic rules for the two prefix operators look like this:

$$\frac{}{w!p \xrightarrow{w!} p} \qquad \frac{}{w?p \xrightarrow{w?} p}$$

For instance, in process $a!P \mid a?Q \mid a?R \mid a?S$, the first agent outputs an $a$, whilst the remaining three input an $a$, all simultaneously resulting in $P \mid Q \mid R \mid S$. Core to this are the following rules for parallel composition:

$$\frac{p \xrightarrow{w!} p' \quad q \xrightarrow{w?} q'}{p|q \xrightarrow{w!} p'|q'} \qquad \frac{p \xrightarrow{w?} p' \quad q \xrightarrow{w!} q'}{p|q \xrightarrow{w!} p'|q'} \qquad \frac{p \xrightarrow{w?} p' \quad q \xrightarrow{w?} q'}{p|q \xrightarrow{w?} p'|q'}$$

These allow an output to synchronise with an input to form a composite output action and an input to synchronise with another input to form a composite input action. An output action composed with an input action absorbs the input action to remain as an output action, whilst two input actions are combined to one. Two outputs do not synchronise. These rules are very similar to the clock transition rules for parallel composition (cf. Section 2.5.3), the difference being that clocks do not require an "initiator"

action to realise a concrete action. CBS also has an idea of patience, since both the nil and prefix operators hear any message without evolving (provided, of course it is not the message being listened on):

$$\frac{}{\mathbf{0} \xrightarrow{w?} \mathbf{0}} \qquad \frac{}{w?p \xrightarrow{v?} w?p}\, v \neq w \qquad \frac{}{w!p \xrightarrow{v?} w!p}$$

CBS therefore is worthy of consideration, and I will draw on some its ideas in my calculus. Although it will be essentially a timed process calculus, I will be adding a new operator that will enable a CBS style broadcast, though in more limited circumstances (see Chapter 6). Interestingly, CaSE can already do a very similar broadcast to CBS (which is simpler than isochronic broadcast). For example the process $a!P \mid a?Q \mid a?R$ can be written as the following process in CaSE: $\overline{a}.\sigma^a.P \mid \mu X.\underline{a}_{\sigma^a}.\sigma^a.X \mid \sigma^a.Q \mid \sigma^a.R$. Here the clock $\sigma^a$ acts as a proxy for the input action. The additional agent I've added ensures that $\sigma^a$ won't tick until $a$ has been received. Having a clock as an input action may not seem desirable, but this nevertheless show the similarity CaSE and other calculi like it have with CBS.

## 2.6 Conclusion

The aim of this Chapter has been to demonstrate the clear links between process algebra and Web services. A secondary aim has been to show that timed process algebra, and particularly abstract timed process algebra provide a beneficial paradigm for modelling service composition. In the remainder of this thesis I take these ideas further, and use an abstract time process calculus to give a compositional semantics to a wider variety of service composition constructs.

# Chapter 3

# Functional Programming in Haskell

*In this Chapter I will review the core features of the purely functional programming language,* ***Haskell****. Haskell is a language which contains a number of very advanced programming features, particularly related to abstraction and type theory. It is very well suited to the implementation of formal systems, as I hope this Chapter will demonstrate. Haskell will be used for implementation of my process calculus and Web service orchestration engine in Chapter 8, and therefore it merits a review. I will survey the basic features of Haskell, such as types, inductive functions and classes. I then describe how Haskell handles imperative computation through* ***Monads****, how it handles concurrency, and finally examine some of the most recent type-system extensions.*

## 3.1 Introduction to Haskell

HASKELL (Peyton-Jones, 2003) is a purely functional programming language. In contrast to imperative programming, where a program is a sequence of commands acting on an abstract state, a functional program is built by composing together functions, which produce outputs by calculation on the inputs. Haskell is *purely* functional because unlike classical functional programming languages like LISP, the output to a function must be calculated *solely* from the inputs, without any form of interaction with the real world. Haskell's functions are thus *referentially transparent* – they must always produce the same output from the same inputs, since there are no references and no pointers directly involved.

Functional programming is also characterised by the heavy use of *primitive induction*. Most of the time functions and types are created using primitive induction, the input is used against one or more inductive cases, and one or more base cases. This makes Haskell useful for solving problems where a large amount of pure mathematics is involved, since it incorporates many of the required concepts as core constructs.

Haskell is by no means a static language definition, and although it remains true to

its conceptual roots there is much work on expanding it. The latest formal standard is Haskell '98, but this usually provides a bare minimum for what a Haskell implementation will provide. One of the most popular Haskell implementations is the *Glasgow Haskell Compiler* (GHC) which is very much at the fore-front of cutting edge development[1]. It turns out that many of these latest developments suit my needs very well and so in this Chapter I provide a concise outline of the key features of Haskell along with some pertinent recent developments.

## 3.2 Types

Haskell is *strongly typed*, meaning that each function must be assigned a type at compile-time and this cannot change at run-time (except in the case of polymorphism, but see below). Haskell's type-system is one of its defining characteristics since it prevents a large number of faulty programs from compiling which other less strongly typed languages permit. In this chapter we look at the basic features of Haskell's type-system, though more advanced concepts (such as *Generalised Algebraic Datatypes*) will be dealt with later on.

### 3.2.1 Defining Types

There are three ways to define types in Haskell:

1. By *type synonyms*, which effectively act as short cuts to other types. For instance:

$$\textbf{type}\ \mathsf{Number} = \mathsf{Int}$$

2. By *data functor*, known as **newtype**, which does the same as a type synonym, but creates a new concrete type with a data constructor instead of simply a synonym. For instance:

$$\textbf{newtype}\ \mathsf{Number} = \underline{\texttt{NumCons}}\ \mathsf{Int}$$

   Hence, a new constructor is created – $\underline{\texttt{NumCons}} :: \mathsf{Int} \rightarrow \mathsf{Number}$.

3. By *data type* definition, the most commonly used method which allows the definition of a type as a sequence of constructors, each with a set of type parameters. For instance:

[1] Much of this work is also being used to create a new standard called *Haskell'* (Haskell prime). For more information see `http://hackage.haskell.org/trac/haskell-prime/`.

```
data Number =
  IntCons Int |
  FloatCons Float |
  VectorCons Float Float
```

Each constructor definition is separated by a pipe |, and again defines a function into the new type, for instance VectorCons :: Float → Float → Number.

Datatypes can be pattern matched in function definitions and are closely associated with *structural induction*, one of the most common ways of defining types and functions in Haskell. The list type in Haskell, as in most functional languages, is one of the most commonly seen instances of an inductive datatype. A list is defined as either 1) an empty list (the base case) or 2) an element followed by a list (the inductive case). For instance, we could represent an inductive list in Haskell using the following data-type definition:

```
data List a =
  EmptyList | Cons a (List a)
```

Notice that the list type is itself parametrised over another type $a$, which is the type of element in the list. Parametric types are also a key feature in Haskell, since they allow partially defined types, whose parameters can be filled in later by the user. Parametric types can also be found in Java in the form of *generics*[2]. An inductive function over the list datatype is written like so:

```
length :: List a → Int
length EmptyList = 0
length (Cons x xs) = 1 + length xs
```

Notice we distinguish each case of the type and either bottom out or reapply the function inductively. The list type is so ubiquitous that it is built in to Haskell, written as [a] and its two constructors are [] and (:), though the notation [1, 2, 3, 4] :: [Int] may also be used.

### 3.2.2 Type-class Polymorphism

In addition to monomorphic type definitions, Haskell also supports *polymorphism*. Using polymorphism it is possible to write functions which act on inputs constrained by a *type class*, rather than a single type. A type class in Haskell consists of a collection of methods

[2]See http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html.

and must be instantiated for each type which the methods can act on. For instance one of the most simple type-classes is the *Eq* class which contains all the types whose values can be tested for equality:

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
```

This class has a two methods $(==)$ and $(/=)$ (the brackets mean they are infix functions) which take two values of the given type $a$ and test them for equality or inequality, respectively. An instance for the datatype *Bool* would look like this:

```
instance Eq Bool where
  True == True  = True
  False == False = True
  True == False = False
  False == True = False
```

Inequality is usually defined automatically as simply the negation of equality, and this is the case here. If the programmer then wishes, for example, to constrain a function `refl` using this type class, he could define the function type as follows:

```
refl :: Eq a ⇒ a → Bool
```

The constraint Eq a can be thought of as a logical constraint, that is if Eq a is true (i.e. a satisfies Eq), then this function can be applied to values of type a. All such constraints are implicitly placed under a universal quantifier $\forall$ (written as **forall** in Haskell), thus the full type of `refl` is $\forall$a.Eq a $\Rightarrow$ a $\rightarrow$ Bool. Haskell's **forall** quantifier can also be used to create *existentially quantified* types, where a type is identified merely in terms of its constraints. For instance a datatype containing any value in the Num class, the class of Numbers, is defined like this:

```
data NumT = ∀ a . Num a ⇒ NumC a
```

Notice that NumT isn't parametric – I have "hidden" the type of the encapsulated value behind the quantifier in variable a. The constructor NumC therefore has type $\forall$a.Num a $\Rightarrow$ a $\rightarrow$ NumT – it will inject any type in the Num class into the NumT type. When we deconstruct it we must therefore get a value in the Num class out – specifically of type $\exists$a.Num a $\Rightarrow$ a, although Haskell doesn't have an explicit **exists** keyword,

it is implied by the context. The only thing we can do with the contained value is apply functions in the Num class to it, as we don't know anything else about it. Existentially quantified types are therefore useful for defining heterogeneous collections where the only functions available on the values are those defined in a type-class.

## 3.3 Monads

In Haskell, a *monad* is a unary type constructor M along with two functions. A monad can be considered, at this point, as a "container" for some type a. The first function known as *unit* or **return** in Haskell injects a value into the monad and has type a → M a. The second function called *bind* or ≫= in Haskell composes a monad containing a type a with a function from a to the same monad containing another type b. It has type M a → (a → M b) → M b. These functions are gathered together in the type-class Monad.

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

For our purposes, a monadic value M a can be thought of as a computation for acquiring a value of the parametrised type a. Therefore, **return** is a computation which does no work and simply returns a constant value. Bind (≫=) takes a computation producing a value of type a, a function mapping such a value to a computation producing another type b, and binds these two together producing a value of type b. Bind is therefore a kind of sequential composition which threads a value through. Using these two functions Haskell effectively has a kind of imperative sub-language, where the abstract state is represented by M. Haskell provides an alternative syntax for monads built using the two functions which shows this even more clearly:

```
addInputs :: MyMonad Int
addInputs = do x ← getNumber
               y ← getNumber
               return (x + y)
```

The syntax used here is called the **do**-syntax. Basically, it is a shorthand for writing a collection of computations and combining them sequentially using bind (≫=). In addition, intermediate values of the computation may be stored in simple aliases using x ← c, which assigns the value produced by computation c to alias x.

The idea in addInputs is that there is a computation called getNumber :: MyMonad Int which will, by some method, acquire a value of type Int using the monad called MyMonad. This method is applied twice in sequence, and the returned values are stored

in variables `x` and `y`. These values are then added together and their sum is returned. Notice that the return value is not simply of type Int – this value returned cannot be automatically separated from its computational context. Clearly we don't know how MyMonad is implemented, it is just an abstract monad from which we know a number can be acquired and the two operations can be executed. It could be the case that it is a completely pure computation involving only an internal state, in which case the value can be extracted, however this may not be the case.

One of the most complicated monads in Haskell is the IO monad, which allows Haskell access to the real world. A value may never be separated from the IO monad as it depends on the state of the world which can never be completely known. The IO monad is used to do all the "dirty" work in Haskell, such as performing communications over a network – transactions which are external to the CPU and thus inherently unreliable. The IO monad is how Haskell can do real-world computations and still remain pure, because impurely acquired values are locked in the box, and therefore the language still remains referentially transparent. A "magic" function of type IO `a` $\rightarrow$ `a` breaks this referential transparency. Nominally, IO can be viewed as a transformation of an imaginary type called "RealWorld", which represents the state of the world. A typical IO computation is `readLine` :: IO String, which takes no functional inputs and produces the string which the user types in on the keyboard.

However, apart from IO there are a large number of *pure* monads in Haskell which are essentially function abstractions. That is to say, the method of producing their value can be collapsed to a series of pure functions, rather than operating on an external state. One of the simplest pure monads is the Maybe monad, a parametric type which can either convey a value or nothing.

```
data Maybe a = Just a | Nothing
```

The Maybe monad represents the computation of a value with the possibility of an error occurring, where failure is represented by the `Nothing` constructor. This idea is conveyed by its instance declaration of Monad:

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

Returning a value in the monad simply wraps it in a `Just` constructor. When sequentially binding two computations, if the first computation is `Nothing` signifying failure, the composition is clearly also `Nothing`. If the first computation returns a value, signified by `Just`, then the value is applied to the function to give the resulting computation.

For instance, the following computation (using the **do** syntax) always fails, resulting in `Nothing`:

```
myComp = do x ← doSomething
            y ← doSomethingElse
            z ← Nothing
            return (x, y, z)
```

Even though `x` and `y` are both assigned values from computations which may succeed, both of them are followed by a failure indicated by `Nothing` and therefore the fourth line returning the triple is never reached. The computation always results in `Nothing`. Apart from being able to sequentially compose potentially failing computations, it is also useful to have an alternative branch if something does go wrong. This is where another class closely related to Monad comes in called MonadPlus.

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

MonadPlus provides 0 and a + operator for monads, which may mean several things depending on the context. Much of the time though MonadPlus is used to implement a form of choice between computations, along with an explicit notion of failure embodied by `mzero`. It can be thought of as a form of *back-tracking*, where the failure of one computation leads to the execution of another.

The instance for Maybe is as follows:

```
instance MonadPlus Maybe where
  mzero = Nothing
  mplus Nothing m = m
  mplus (Just x) m = Just x
```

The semantics of `mplus` is a very intuitive way of handling failure. If the first computation returns a value it is returned by the whole, otherwise the second computation is returned (which may in turn contain additional choices). For instance, if I were to compose the failing computation `myComp` with a constant value, i.e. `myComp` ‘`mplus`‘ **return** $(1, 2, 3)$, since the left-hand side always fails, the right hand-side is used to output an actual value.

Another example of a pure monad is the State monad. In contrast to the IO monad which gives facilities not otherwise available in Haskell, State allows imperative code with a mutable state variable. The structure of the State monad is as follows:

```
newtype State s a = State {runState :: s → (a, s)}
```

A state monad is very intuitive. It consists of a function mapping the current state (of type s) to an a value and a new state. For example, the state s may contain a collection of variable assignments, and the computation of the output value a may involve use (and alteration) of these variables. The instance of Monad for State looks like this:

```
instance Monad (State s) where
  return x = State (λ s → (x, s))
  (State f) >>= m = State (λs →let (x, s') = f s
                                   State g = m x in g s')
```

The implementation of **return** simply returns the given value without changing the state. The implementation of $\gg=$ first constructs a new function which applies the incoming state to the first computation, giving an output pair $(x, s')$. The output value x is then applied to the function on the RHS of bind and the intermediate state $s'$ is applied to the new computation. Two further important functions of the State monad called `set` and `get` are for mutating and accessing the state of the monad respectively.

```
set :: s → State s ()
set s = State (λ s' → ((), s))

get :: State s s
get = State (λ s → (s, s))
```

The first is a function which ignores the incoming state and replaces it with the state passed to the function. The second simply returns the incoming state as the output (as well as maintaining it).

We have now examined three monads, all of which have a different feature set. The IO monad provides real-world computations. The Maybe monad allows composition of failing computations. The State monad provides a mutable state variable. But how do we go about combining the features of multiple monads? The answer is *Monad Transformers*, which allow monads to be "stacked", enabling the combined feature set. The idea is that a monad transformer is a monad which is itself parametrised over another monad, and then computations in the encapsulated monad are performed via a "lift".

```
class MonadTrans t where
  lift :: Monad m ⟹ m a → t m a
```

For instance the StateT monad is the equivalent transformer for the State monad. It allows the combination of a mutable state variable with whatever features another monad provides (which may itself be another monad transformer).

```
newtype StateT s m a = StateT {runStateT :: s → m (a, s)}
```

In this instance what was previously simply a function producing an output value and new state becomes a computation in the encapsulated monad.

```
instance MonadTrans (StateT s) where
  lift m = StateT (λs → (m >>= λx → (x, s)))
```

The instance of MonadTrans implements `lift` for StateT. It takes the monad to lift, and constructs a state transformation function which copies the original state, and outputs the value produced by the encapsulated monad in the parent StateT monad.

## 3.4 Concurrent Haskell

Concurrent Haskell is an extension which supports concurrent execution of IO monad computations. It is similar to imperative concurrency extensions, such as *Concurrent Java*, in that it is based on the *thread* concept, where a collection of independent threads are scheduled for parallel execution, possibly across multiple processors. The main function of Concurrent Haskell is `forkIO` :: IO () → IO ThreadId, which takes an IO computation and spawns a thread for it to execute in, returning its identifier, ThreadId. The threads are by default managed internally by GHC's own scheduler, but there is an additional function `forkOS` which runs the thread under any compatible operating system scheduler, such as the Unix process scheduler. Concurrent Haskell provides a number of additional functions for manipulating threads, usually found in its imperative counterparts. For instance:

- `killThread` :: ThreadId → IO () aborts a thread's execution;
- `yield` :: IO () forces the execution of the current thread to pause until other peer-threads have been given the opportunity by the scheduler to progress;
- `threadDelay` :: Int → IO () forces the execution of the current thread to be suspended for the specified number of milliseconds.

An important part of Concurrent Haskell is MVar, a polymorphic type which represents a thread shared variable. A value of type MVar `a` is a container for type `a` which

can be either empty or full. When initialised using `newEmptyMVar` :: IO (MVar a) it begins empty. A thread can then place a value inside with `putMVar` :: MVar a → a → IO (), which another thread can then read using `takeMVar` :: MVar a → IO a. If a thread tries to take from an empty MVar, or populate an already full MVar, the thread will block until the variable becomes populated. Using MVar it is possible to prevent *race conditions*, since only one thread at a time may modify an MVar. In contrast multiple reads can be achieved using `readMVar` :: MVar a → IO a, which leaves the value in place. Care must of course still be taken, because a dead-lock can easily occur when two threads compete for two MVars.

GHC's implementation of Concurrent Haskell is extremely competitive in performance terms with its imperative peers. In recent benchmarks[3] it has out-performed languages such as *Java* and *GNU gcc* in a number of concurrency related areas, such as fast thread switching. In particular its memory requirements are many times less than those of several of its competitors. It is not the most efficient in all areas, but is most certainly a contender and therefore the choice of GHC Haskell for implementing a program where concurrency is central is certainly justifiable.

## 3.5 Generalised Algebraic Datatypes

Generalised Algebraic Datatypes, or GADTs as they are more commonly known, are a relatively new extension to Haskell which allow the constructors of data-types to be assigned a type directly. I will use the standard example given by Peyton-Jones et al. (2006) since it illustrates the point well. Consider the datatype for an expression using Haskell 98 datatypes:

```
data Term =
  LitI Int           | -- Integer literal
  LitB Bool          | -- Boolean literal
  Inc Term           | -- Increment the given term
  IsZ Term           | -- Is the given term zero?
  If Term Term Term  | -- If -- then -- else--
  Pair Term Term     | -- Construct a pair from two terms
  Fst Term           | -- Decompose a pair to the first element
  Snd Term           | -- Decompose a pair to the second element
```

The syntax of this simple language allows many expressions which are not well typed. For instance `IsZ` (`LitB True`) would fail to evaluate, as `IsZ` expects a number. Ideally we need a type parameter to `Term` which publishes the type of the given expression. Standard algebraic datatypes, however, only ever allow this term to be the most

[3] See `http://shootout.alioth.debian.org/gp4/`

general type a, for instance LitI :: Int → Term $a$. This does not restrict the types of expression which may be passed at all.

GADTs provide a solution – instead of the above type, we can define it as follows:

```
data Term a where
  LitI :: Int → Term Int
  LitB :: Bool → Term Bool
  Inc  :: Term Int → Term Int
  IsZ  :: Term Int → Term Bool
  If   :: Term Bool → Term a → Term a → Term a
  Pair :: Term a → Term b → Term (a, b)
  Fst  :: Term (a, b) → Term a
  Snd  :: Term (a, b) → Term b
```

GADTs allow the programmer to define each constructor as if it were a regular function, and thus instantiate the type parameters. Thus in this definition IsZ will only take a term of type Term Int and produce Term Bool. Similarly, Fst expects a pair type and produces a value of the type of the first element. GADTs represent a strengthening of Haskell's type system. Where in Haskell 98 different data constructors had to give the most general type, now they can restrict this and provide greater guarantees about data.

One disadvantage of GADTs though is that since the type of the encapsulated data depends on the type parameters, pattern matching cannot infer the types of this data. As a result functions which use pattern matching must provide explicit type signatures.

## 3.6 Type Families

*Type families* (Schrijvers et al., 2008) are another recent extension to Haskell's type-system. A type family allows a datatype or type synonym signature to be separated from its body, in a similar way to type-classes and instances. The body of a type will be determined by the parameters of that type.

*Type synonym* families allow the declaration of what effectively amount to *type functions*: functions which take types as input and produce other types as output. In the previous section we saw how GADTs can be used to form datatypes whose type depends on the type of the constructors' parameters. Type functions allow this to be taken one step further and have the type body formed by the invocation of a function on the constructor types. An example given by Schrijvers et al. (2008) is the Vector type, which is parametrised over its length and element type:

```
data Z
data S n
data Vector n a where
  VNil  :: Vector Z a
  VCons :: a → Vector n a → Vector (S n) a
```

They first define two types to represent natural numbers at type-level, Z for zero and S for successor. These are *phantom types* in that they have no constructors, and thus contain only one possible value, $\perp$. Using these types, the number 3 would be represented, for instance, as the type S S S Z. The Vector type is an inductively defined GADT similar to the standard list type. It can either be VNil, in which case it has length Z, or VCons in which it has length S n where n is the length of the tail vector. The question then is how to define the type of the concatenation function for two vectors. Clearly the length of the resultant vector will be the sum of the two vectors' lengths. Type families provide the answer, as shown in the following definition of vconcat.

```
vconcat :: Vector n a → Vector m a → Vector (n :+: m) a
vconcat VNil ys = ys
vconcat (VCons x xs) ys = VCons x (vconcat xs ys)
```

The body of vconcat is simple – it follows the standard definition for lists. The type though has to sum the two vector lengths n and m. It uses the :+: type constructor to do this, which is a type function as defined below:

```
type family x :+: y
type instance Z :+: y = y
type instance (S n) :+: y = S (n :+: y)
```

The first line declares the *kind* of the type family, specifically that it takes two other types as parameters. We could equivalently write $:+: \; :: * \to * \to *$. The remaining lines define the body of the type depending on the parametrised types. The body follows the usual inductive definition for addition on natural numbers. Notice that the definition uses type family recursion. Recursion on type families is very limited, since type families must be decidable. The current implementation of type families in GHC as given by (Schrijvers et al., 2008) supports only a single recursive call per body. Therefore it is currently impossible to define multiplication, for instance.

A type family may also be associated with a class, in which case its members are called *class associated types* (Chakravarty et al., 2005). The body of a type associated with

a class is then determined by the individual class instances, with the class just providing only the type signature. For instance, we could declare:

```
class MyClass a where
  type MyType a
  myFunc :: MyType a → Int
```

This is similar to declaring a normal type family outside a class, but the difference is that `myFunc` can make use of the knowledge about the structure of MyType `a`. If the type family were declared outside a class, any function can only be declared forall `a`, and not on an instance basis. This is useful, for instance, in defining an datatype index type without using multiple class parameters:

```
instance Collection c where
  type Index c
  lookup :: Index c → c a → a
instance Collection List where
  type Index List = Int
  lookup k xs = xs !! k
instance Collection (Map k) where
  type Index (Map k) = k
  lookup k m = Map.lookup k m
```

In this example we have a class Collection which instantiates a parametric type representing an indexed collection. The index type is populated by specifying a body for the type Index in each instance. Then each collection can use its respective index types to perform a lookup. The index for a list, for instance, is simply Int, and therefore this is the body of Index [a]. Likewise, a Map is indexed by the parametrised key type `k`, and therefore this is also the body of Index (Map `k`).

## 3.7 Conclusion

Haskell, as I have demonstrated in this Chapter, is a very capable language. It is also increasingly the case that features from functional programming are finding their way into more mainstream languages. The most well-known example is Java's Generics[4], introduced in Java 5.0, which are very similar to Haskell's parametric types and typeclasses. There are also moves afoot to include other functionally inspired features, such

[4]See `http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html`

as *Closures*[5], which are similar to Haskell's lambda abstractions. Therefore Haskell cannot simply be thought of as a toy, it is a serious language which is both useful, and inspiring future programming features.

It therefore seems right that Haskell be the choice of language for my work. Haskell is both formally based and practically useful, and can therefore help bridge the gap between my process algebra theory and implementation of Web service composition. Furthermore, it also provides an elegant and highly customisable syntax which, while difficult for the novice programmer, is wholly suitable for specifying a formal system. In addition, since it is equipped with powerful concurrency features and my own recent advances in Haskell Web services (Foster, 2005), it seems Haskell is the ideal language for creating a Web service composition server, which will integrate with existing tools. I will therefore use Haskell in Chapter 8 to build the genesis of such an implementation.

[5]See `http://www.javac.info/`

# Part II

# Theory

# Chapter 4

# Contribution Overview

*In this chapter I describe the main contributions of this Thesis. I frame a Web service composition model called* **Cashew**, *and provide the groundwork for the following chapters. The model consists of a three-level stack which describes how a Web service description is refined from a diagrammatic representation, to a process model, to an executable machine.*

HAVING covered all the necessary theoretical groundwork for this thesis I now proceed to outline my main contributions. My goal is to provide a novel approach to the study of service composition. At its core the aim of my work is to provide a generic *director model* for Web service composition. Specifically, I will use a Timed Process Calculus to represent the behaviour of a workflow as a synchronous exchange between a collection of actor processes. Each behavioural element of a composite Web service's workflow will be represented as an actor process representing its behaviour and the communications it makes with its environment. The communications will adhere to a common *protocol* which will be used to constrain if and when the actor performs. These processes will then be joined together by semantic combinators which schedule their execution according to a particular pattern. For instance, a sequential scheduler will require that the first element be executed to completion before the second can begin.

This completed director model will yield a transition graph representing the behaviour of a workflow. Then, for verification, certain conclusions can be drawn about the workflow. For instance, we might check if all the preconditions of a process are provided by its context. If not, that branch of the workflow will not be able to execute and a deadlock may result. Furthermore, the nature of the director model will enable the extraction of a Web service *interface model* (or choreography), which can be used to advertise its behaviour on the Web. More detail of the process architecture of my semantics meta-model will be given in Chapter 6. For now I concentrate on my general approach to modelling Web services.

In my model I highlight three distinct but related aspects of a workflow, each of

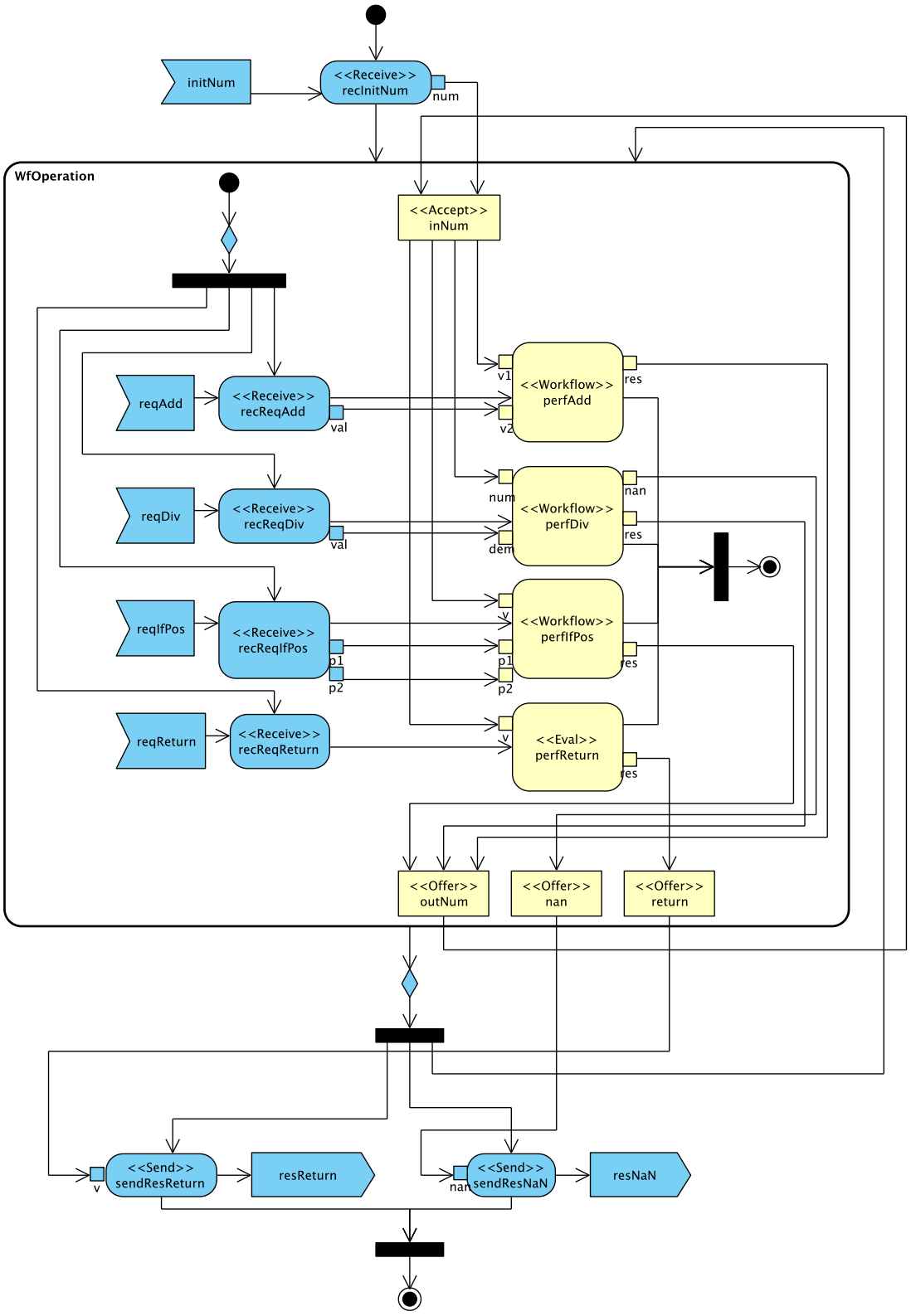



Figure 4.1: A simple calculator orchestration as a UML 2 Activity Diagram

which has a stake in determining the execution order of a component. First of all is the **control flow**, which as expected provides a number of combinators which strongly constrain the order in which a group of processes can be executed. But since control flow alone is rather static in nature, a workflow also has **data flow** which describes how the data provided by one component are fed into another. An input may be mandatory or optional and therefore data flow also enforces an execution order on components. Finally since we are dealing with a Web service which necessarily communicates with its client and other Web services, the execution order is also constrained by **message flow**. Specifically, a workflow may require another Web service to complete an operation and return a message before it can continue. Message flow also allows the differentiation of internal from external choice, since it provides a method by which the client can make a choice based on which message he provides.

All three of these aspects are found, to a greater or lesser extent, in the majority of modern workflow systems, in particular WS-BPEL. For example, Figure 4.1 demonstrates a simple calculator Web service as a UML 2 Activity Diagram. The service first inputs an initial value, and then allows the execution of three operations on an accumulator, in the style of a traditional push button calculator. The three operations are (for simplicity) addition, division and a simple if statement, which returns the first value if the accumulator is positive, or the second if negative. The process ends either when the return message is received, indicating the user wishes the output to be given, or when a division by zero is encountered. The main workflow of the calculator is called WfOperation which takes an input, represented by object inNum, and outputs one of three outputs, outNum, nan or return. This workflow inputs a number, taken from either the initial message or from a previous computation, and waits for one of four operation messages (the fourth being the return button). Once one of these is received, a corresponding sub-workflow (encapsulated in a *performance*) is executed to compute the next value. If the incoming message corresponds to one of the three operations an intermediate value is output which is then fed back into WfOperation. This workflow is then placed in a loop, with the two exit conditions being the output of a return or nan.

This simple example shows clearly all three aspects of a workflow. There is a control flow which determines the overall order of activity, data flow which determines the results of an operation, and message flow which is used to choose the next operation. My approach will consider all three of these aspects rigorously and individually. I also emphasise the importance of *compositionality* – the idea that the model's semantics must be expressible purely in terms of its parts. This property is of particular importance for Web services for two main reasons:

- The model should support *rapid application development* with real-time verification which allows incremental model construction. I envisage the user using an interactive environment to build a diagram modelling their intended Web service composition. Every edit that the user makes on the diagram will induce an edit on

the underlying formal model. Thus, there will be no compile-execute cycle as the model will be constantly up-to-date. For instance, we may wish to add additional operations to the calculator but without recomputing the semantics of the operations we already understand. The underlying formal model will be used to inform the user about any deficiencies in their existing model using a variety of model checking techniques. Furthermore, compositionality ensures that any conclusions we reach about part of the workflow (for instance using a temporal logic formula) are still retained in a different component context.

- Web services are by nature highly dynamic and different parts will frequently need replacing as the face of the Web changes. This becomes particularly the case with the *Semantic Web* where services are constructed on the fly at the user's behest. For instance in the event that the Web service for adding numbers disappeared from the Web, we would need to be able to replace it with another equivalent service seamlessly.

Since my behavioural meta-model has compositionality as a core property, it is important that the behaviour of two components can be related through a semantic equivalence and/or pre-order. This determines when a substitution can be made without affecting the behaviour of peer processes. Hence, I will follow the rich collection of research outlined in Chapter 2 Section 2.3 and create a novel *process calculus*. My approach differs from other approaches such as *Orc* (Misra and Cook, 2007) in that I will not extend CSP, CCS or another abstract process calculus with Web service operators. Rather my intention is to remain abstract and describe the director model through a variety of *synchronisation patterns*. Through a small number of CCS-style operators and the use the of *abstract time* each actor in the workflow can be scheduled and linked with its peers via data flow. By describing a Web service's orchestration this way we achieve compositionality, but in addition a Web service interface can be extracted from the resulting process. My new calculus, $\mathsf{CaSE}^{\mathrm{ip}}$ , will be discussed and given a detailed theoretical analysis in Chapter 6, the main emphasis being on the compositionality result.

However, since the process calculus itself will not provide the constructs necessary to directly build workflows, and it is indeed cumbersome to model them directly in a CCS-like setting I will begin, in Chapter 5, by outlining an experimental language for describing Web service orchestration called Cashew-A. The aim of this language is not to supersede any existing workflow and Web service languages. On the contrary, my aim is to provide what can be described as a *mediation layer* for workflow languages. That is to say, my language will provide a core set of constructs with sufficient expressivity to represent the ideas found in a wide variety of languages. Thus, suitable mappings from languages such as WS-BPEL into my language will allow their programs to be studied in a unified environment. I will demonstrate the generality of this language by reference to the *workflow patterns* (van der Aalst et al., 2003) discussed in Chapter 2 Section 2.2.4.

Cashew-A is *block-structured* (rather than graph based) language and defines a composition in terms of *workflows*, where a workflow is simply a collection of abstracted components called *performances* with a given ordering schema and data flow, including inputs and outputs. The concepts behind Cashew-A originated from our study into giving an operational semantics to the process model of OWL-S (Norton et al., 2005). OWL-S does not allow the composition of Web services generally, but rather it allows "operation composition" (Norton, 2005*b*) in which Web services are viewed as simple input-output remote procedure calls. These procedures can then be joined together into the control flow using the various constructs available, which as usual bear resemblance to standard process algebraic constructs. In addition the OWL-S model allows for the specification of dataflow connections between performances.

Cashew-A is built on these same ideas, but with a number of key differences. First, the control flow element is given in the style of a basic process algebra. Since I will be giving this language a compositional semantic theory using an actual process calculus it is necessary that workflows can be easily built by composition. This will allow evolutions in a Cashew-A composition to be directly represented at the lower levels. Second the dataflow model is more flexible. Rather than having a data flow boundary around each n-ary workflow construct, Cashew-A allows a workflow with control-flow built using algebraic control-flow combinators. Third, in keeping with the WSMO philosophy we will allow Web service composition via goal composition. That is, we still compose what are effectively one-shot input-output agents (goal preconditions and postconditions), but we also allow the specification of direct Web service communication through message passing. However, the message passing will be restricted to strictly one partner within a single component to avoid the need for $\pi$-calculus style channels.

The overall idea is, therefore, to present what is described by Norton et al. (2007) as a *three-level semantic stack*, through which a diagram or high-level language can be mapped into my intermediate workflow language, Cashew-A, and then finally into a formal process-based behavioural meta-model. The three levels are as follows:

1. **Diagram** – a high-level user-end graphical description of the workflow, using a notation like UML Activity Diagrams (UML2AD) or the Business Process Modelling Notation (BPMN);

2. **Cashew-A** – a block-structured language in the style of a basic process algebra with workflow constructs;

3. **Process Model** – a low-level timed process calculus, which provides labelled transition systems for the workflows.

The process specification can then be used for verification via a variety of model checking techniques. In addition the process calculus will be mapped into a form of abstract executable machine. Although the process calculus is abstract, I will make use

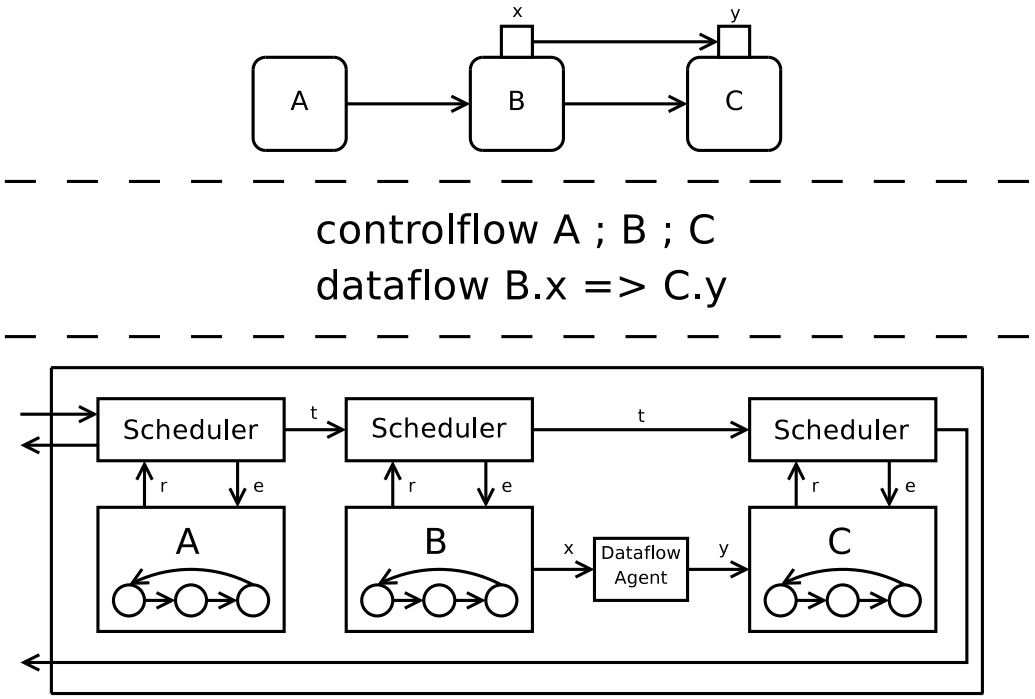



Figure 4.2: An example of refinement from a diagram into a process model

of the various *silent actions* produced to attach real-world actions, such as sending a message to a Web service.

An outline of such a refinement into a process specification is shown in Figure 4.2. Here we have a UML Activity diagram description of a simple sequential process at the top-level. This is then refined into control flow and data flow facets in the Cashew-A language below. Finally this is converted into the process meta-model at the bottom level. Each of the three activities, A, B and C is allocated an actor process to describe its individual behaviour (which of course may in turn be composite). Various schedulers control the execution of these actor processes by synchronising with them according to a predefined pattern. For instance in this case the environment must allow execution of the construct as a whole before the first element can be permitted to execute. Notice also that the single dataflow connection also has an associated process which in a more complicated process would ensure that the data is communicated between appropriate actors.

Having described the conceptual stack, I now outline my proposed software architecture. It consists of three components, a high level **Service Designer** which allows a software architect to describe a Web service using a diagram notation (or rather a subset of one). Then there is the main part of the system, the **Service Composer** server. The server holds an abstract model of the service being described in the Service Designer. Every time the architect acts on the diagram, this action will be replicated by the server on the model. The model contains the stack described above, but also some other pieces. Since the low-level Abstract State Machine model is implemented in terms of media-

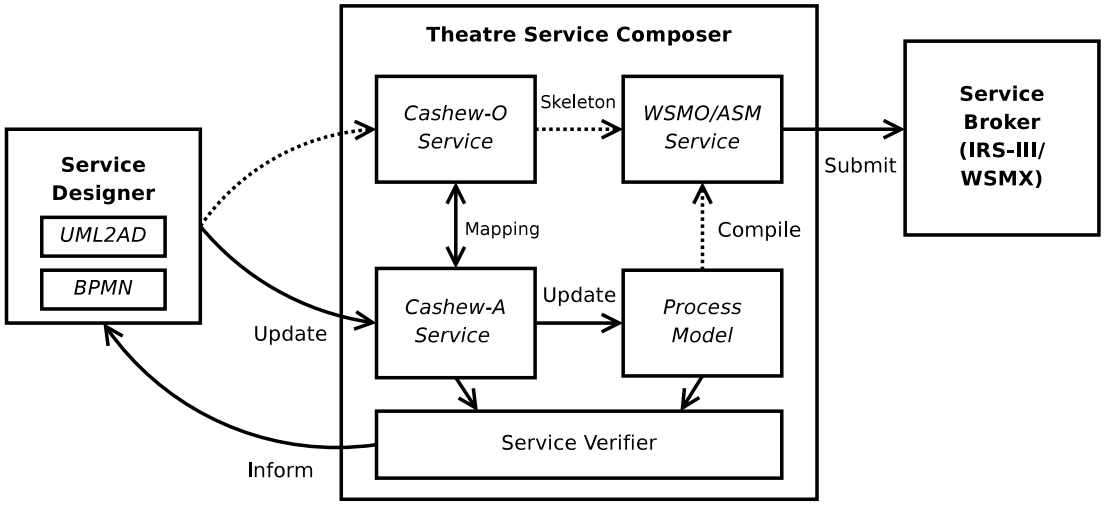



Figure 4.3: Overall Cashew Software Architecture

tors which are represented at a lower-level in the Process Model, it is necessary to map Cashew-A onto the Cashew WSMO Ontology (Cashew-O) and store both models separately. Cashew-A then feeds into the Process Model with updates to the service model, and both feed into the **Service Verifier** which provides information back to the designer about potential problems. Finally, the Process Model is mapped to an ASM and, using the groundings represented in Cashew-O to provide a skeleton, a complete WSMO description is produced. This can then be submitted to a **Service Broker** to be published on the Web. This software architecture is not implemented in this Thesis, but much work is devoted towards a reference implementation of both the process calculus and Cashew-A in Chapter 8. This final piece of work demonstrates the *practical viability* of my approach.

This concludes the overview of my contributions. To summarise, my contributions in this thesis are as follows:

- As study of the Cashew-A language (Chapter 5);
- A novel abstract time process calculus called CaSE$^{ip}$ (Chapter 6);
- An extensible semantic framework using CaSE$^{ip}$ for Cashew-A (Chapter 7);
- An implementation of CaSE$^{ip}$ in Haskell (Chapter 8);
- A partial implementation of the Cashew-A semantic framework (Chapter 8).

All of these together provide the underlying framework by which the software architecture proposed above can be implemented. In the next Chapter I will begin the discussion of Cashew-A.

# Chapter 5

# Service Composition Language

*In this chapter I describe* ***Cashew-A*** *, my experimental service composition language.* Cashew-A *is block-structured component language with process algebra style operators for building control flow and data flow connections. I will extend this language with compensation. I will also seek to justify my choice of operators for this language by comparison with the* ***Workflow Patterns*** *and* ***WS-BPEL****.*

## 5.1 Overview

CASHEW-A IS A FORMAL LANGUAGE for describing Web service composition in a component-wise manner. My aim has been to design a concise language with maximum generality in terms of the representable behaviour. Thus, unlike previous work based on OWL-S (Norton et al., 2005), I use binary control-flow operators in order to achieve a more algebraic syntax, with n-ary operators being used only when necessary. Along with our existing work, I use the *workflow patterns* (van der Aalst et al., 2003) as an inspiration for the selection of operators which should be provided in a modern workflow language. Though I have tried to integrate the maximum number of these patterns, not all can be catered for as Cashew-A is a *block-structured* rather than graph oriented language. For instance a workflow using the "arbitrary cycles" patterns cannot be built by composition as this pattern is inherently graph-based (see Chapter 2, Section 2.2.4). Nevertheless, it is my belief that the inclusion of both control and data flow as first-class entities gives the maximum degree of flexibility, and a certain amount of graph structuring can be achieved. A full review of my language with respect to the workflow patterns is given in Section 5.8.

As explained in Chapter 4, Cashew-A is not intended to be used directly, but is rather an intermediate language for a variety of other workflow languages. An example of readable syntax is shown in Section 5.7. The idea of the language is to represent the fundamental concepts which a service composition language should have, into which other languages can be mapped.

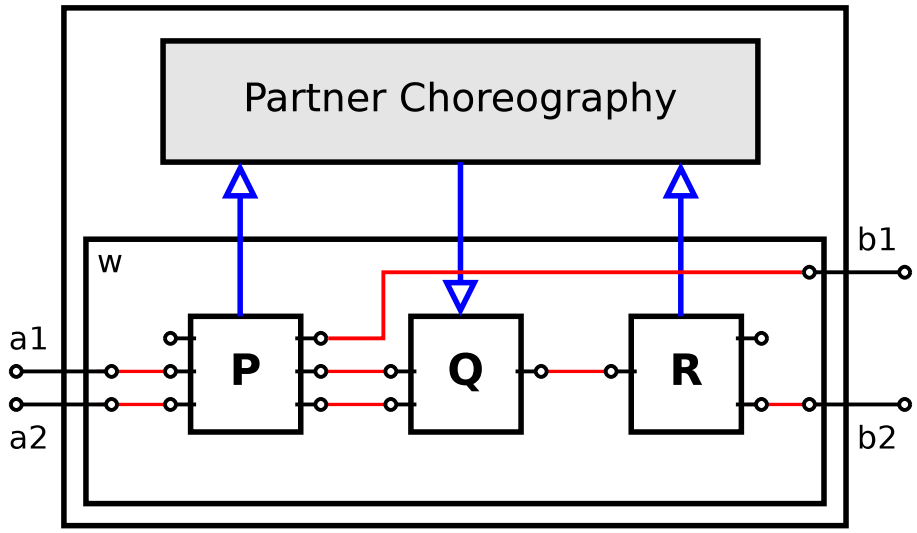



Figure 5.1: Overall view of a Cashew-A workflow

Cashew-A has two fundamental concepts which are used to construct a Web service, the **performance** – an abstracted component with inputs, outputs, and internal behaviour (possibly including a message exchange), and the **workflow** which composes a collection of performances. A performance represents the instantiating of a component in a particular context, whilst hiding the internal details. The execution order of performances in a workflow is determined by three factors:

1. **Control flow** , the explicit order in which components are executed;
2. **Data flow** , how data moves between components, fulfils preconditions and thus provides a further implicit order;
3. **Message flow**, the exchange of messages with a partner (the service choreography). As in the WS-BPEL pick activity, messages can be used to determine the direction of flow.

A workflow is within the scope of a single *partner* at any one time. A partner can either be the client of the Web service or alternatively another Web service, for which the workflow acts as client. This approach avoids the *partner-links* concept found in WS-BPEL, where there may be communication with any number of partners at any one time via $\pi$-calculus style channels. Thus we can avoid the need for any sort of mobility or channel passing. It also fits much better into a block-structured component system, since we abstract communication with a Web service to an ad-hoc one-shot component, whose implementation and provider can be easily changed.

An example of workflow structure is shown in Figure 5.1. In this example we have a workflow w, consisting of three performances **P**, **Q** and **R**. The control-flow is unspecified, but we could assume a simple sequential composition of three performances. The workflow has two inputs a1 and a2, and two outputs b1 and b2, which are connected

to the performances via dataflow connections. Each of the performances has a collection of inputs and outputs, and these are also connected together with dataflow connections. For instance, a1 is connected to one of the inputs of **P**, meaning this workflow input feeds into **P**, and one of **P**'s outputs is connected to the workflow output b1. Inputs to a performance need not always be connected, it depends on whether they are optional or mandatory (one of **P**'s inputs is left unconnected). Furthermore, a single output from a performance can be copied to multiple locations since each output is broadcast to all outgoing dataflow connections. Similarly, an input can be drawn from several different places, though only one will be chosen at execution time. The three performances also engage in a message exchange with the composed partner, represented by the partner choreography. **P** sends a message, whilst **Q** receives a message.

Each workflow is also timed using a discrete *real-time clock* (RTC), which is approximated in terms of an abstract clock at the process level. Each component in a Cashew-A Web service observes time according to a predefined *granularity* – the amount of time that elapses between each tick of the RTC. The motivation for including an RTC measure will be revealed in more detail when we consider the semantics of *compensable transactions* in Chapter 7, Section 7.3, but their general usefulness should be clear. For instance having an RTC allows the specification of *timeouts*, which are useful for deciding whether a specific operation is taking too long. Also they allow for a greater degree of fairness between parallel processes, since approximately equal time can be devoted through lock-stepping.

Having outlined the structure of a workflow, I now turn to the formal definition. For convenience in defining the syntax of Cashew-A, I assume the existence of a number of sorts:

- $(a \in)\mathbb{A}$ and $(b \in)\mathbb{B}$, the sets of workflow input and output labels;
- $(w \in)\mathbb{W}$, the set of workflow labels;
- $(p, q \in)\mathbb{P}$, the set of performance labels;
- $\mathbb{A}^{\mathbb{P}} = \{a^p | a \in \mathbb{A} \wedge p \in \mathbb{P}\}$ and $\mathbb{B}^{\mathbb{P}} = \{b^p | b \in \mathbb{B} \wedge p \in \mathbb{P}\}$, the sets of performance input and output labels;
- $(m \in)\mathbb{M}$, the set of messages.

Each of these sets is treated as isomorphic with a BNF definition. I now proceed to examine the workflow structure in detail. I will present a mutually recursive definition of the main *workflow* construct, which will consist of data flow and a control flow, which may in turn be made of other workflows. I begin by examining exactly what a workflow is, then describing the data flow layer in Section 5.4, the control flow layer in Section 5.5, and the various components which can be used in workflows in Section 5.6.

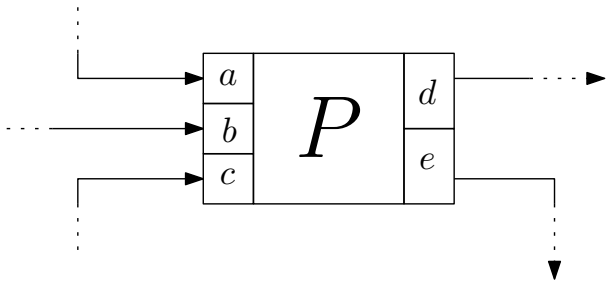



Figure 5.2: Performance with dataflow context

## 5.2 Workflows

Workflow is one of the two fundamental units of composition. In Cashew-A a workflow is a collection of abstracted components connected by both control flow and data flow, with input and output data mediated through a number of *dataflow connections*. Components are composed into a workflow via wrappers called *performances* (the second fundamental unit of composition), the name indicating that we are looking at an instance of the component's behaviour within a particular context. Performances, therefore, provide the *"glue"* between the encapsulated component and the workflow, effectively enabling workflow to take control of the component, whilst also distinguishing it from other components and disregarding its internal structure. To this end, a performance consists of a component prototype, a name unique within the workflow and a data flow mediation layer, which consists of locally named data buffers for each input and output. The buffers are necessary so that the workflow can decide exactly when the component takes its inputs, as once the component has them they are outside workflow control. A performance, therefore, is bound to and controlled by the workflow it exists in – it cannot be removed to another workflow. Performances are important as they allow a workflow to have a degree of control over each component without changing the component's internal structure (which would undermine compositionality).

Cashew-A performances are also similar to UML *activities*; they are basic abstracted executable units which fulfil their preconditions (which includes receiving inputs), execute, and provide outputs to the workflow environment. Performances can have a wide variety of pre- and postconditions depending on their body, though normally a performance will simply accept a set of inputs and offer a set of outputs. This is exemplified in Figure 5.2 by a performance with three inputs and two outputs which waits for each of its inputs to be populated. On the behavioural level each input is represented by a buffer which can either be full or empty (beginning empty of course). In the example the three input buffers $a, b$ and $c$ are populated either by inputs from the workflow containing the performances, or from the outputs of other performances. Once a sufficient subset of these inputs has been populated the performance will be able to execute. However, it will first have to secure permission from its environment. For example if the performance is part of a choice block it may be denied permission to execute if one of its peers

has already secured permission (in which case its input buffers will be emptied). Otherwise the performance will execute to completion and then populate its output buffers. The data in these buffers will then be passed on to other performances in the workflow by dataflow connections, or perhaps to an output of the workflow itself (see Section 5.4).

We use the following syntax for performances $\mathcal{P}$, compensable performances $\mathcal{P}_\mathcal{T}$, and vectors of these:

$$\begin{aligned} \mathcal{P} &::= \mathbb{P}[\mathsf{Component}] \mid \epsilon \mid \delta \\ \vec{\mathcal{P}} &::= \mathcal{P} : \vec{\mathcal{P}} \mid \mathbf{Nil} \\ \mathcal{P}_\mathcal{T} &::= \mathbb{P}[\mathsf{CComponent}]_\mathbf{T} \\ \vec{\mathcal{P}_\mathcal{T}} &::= \mathcal{P}_\mathcal{T} : \vec{\mathcal{P}_\mathcal{T}} \mid \mathbf{Nil} \end{aligned}$$

A performance may either be an actual unit of behaviour with name $p \in \mathbb{P}$ and a component body Component, or alternatively a skip $\epsilon$ or halt $\delta$. The latter two effectively define the top and bottom of the precondition lattice. $\epsilon$ is the performance which has an empty precondition, performs no action, and entails the empty postcondition. It is the equivalent of logical **True**, in that it can always execute. $\delta$, by contrast, is a performance with an unsatisfiable precondition, and thus cannot execute under any circumstances. There are two types of performance, normal performances represented by $\mathcal{P}$, which execute normally, and compensable performances $\mathcal{P}_\mathcal{T}$, which may be terminated by the context part way through. For convenience I also define a vector type for each performance class, representing an ordered list of performances.

The different types of components, Component, will be described fully in Section 5.6. Briefly, a component is an action which can be considered atomic or, more appropriately, *one-shot* – the component satisfies its preconditions, executes (with possible message exchange) and provide postconditions; no further input can be provided by the workflow context during execution. Examples of performances are Web service invocation, partner communications, goal achievement, or simply other, abstracted workflows. Each performance must have a locally unique name so that it can be identified within a workflow. This name will be used to refer to its inputs and outputs, to distinguish them from those of other performances. Performances are vital for compositionality since they abstract from the internal behaviour of the component.

Performances are, of course, the constituents of workflows, of which there are two types in Cashew-A, $\mathcal{W}$ and $\mathcal{W}_\mathcal{T}$ as defined below:

$$\begin{aligned} \mathcal{W} &::= \mathbb{W}[\{\mathcal{A}\}(\mathcal{C}\times\mathcal{D})\{\mathcal{B}\}]\mathbb{N} \\ \mathcal{W}_\mathcal{T} &::= \mathbb{W}[\{\mathcal{A}\}(\mathcal{T}\times\mathcal{D})\{\mathcal{B}\}]\mathbb{N} \end{aligned}$$

The two are identical, except that the latter uses a compensable control flow. A workflow consists of the following:

- A name – $w \in \mathbb{W}$;
- A set of data flow inputs – $\mathcal{A}$;
- A control flow, described by the language $\mathcal{C}$ or $\mathcal{T}$ (if compensable), which composes the performances;
- A data flow – $\mathcal{D}$;
- A set of data flow outputs – $\mathcal{B}$;
- A natural number $\mathbb{N}$ representing the real-time granularity of the workflow in an arbitrarily defined unit (for instance milliseconds).

I will now describe all of these elements in detail.

## 5.3 Real-time Granularity

Each Cashew-A workflow includes a discrete real-time clock (RTC) which can be used to time its duration. The period between two clocks is given by the parametrised natural number, measured in suitable units. In a Web service context, it is unlikely that a finer granularity than milliseconds would be required; for instance under perfect conditions it takes about 13ms to contact a server on the other side of the Earth.

The motivation for inclusion of real time, although useful in itself, is the inclusion of *speculative parallelism*. Speculative parallelism, broadly, is when a goal has several paths to completion, and all of these paths are executed in parallel to achieve optimal speed. Once one of them has "won", the remaining paths are cancelled. To implement this it is necessary to synchronise the parallel threads regularly to enforce fairness. Thus an RTC is needed to observe the threads objectively.

Exactly how these clock ticks will be represented and how the various heterogeneous clocks will be mediated will be dealt with in Chapter 7. Goals will also be equipped with granularities, which will indicate the time-frame.

## 5.4 Dataflow

The data flow layer of the workflow consists of the inputs and outputs of the workflow itself, performance input/output buffers and connections between these. Data flow serves approximately the same purpose as variables in a state-based language, passing data from one part of the program to another. By connecting together the data flows of several components an implicit dependency in the form of a partial order results. This

means that data flow, as I have already indicated, plays a role in determining the execution order of the encapsulated performances. I first give the BNF definition of the workflow input logic $\mathcal{A}$, workflow output logic $\mathcal{B}$, and dataflow connections logic $\mathcal{D}$:

$$\begin{aligned}
\mathcal{A} &::= \mathcal{A} \sqcup \mathcal{A} \mid \mathcal{A} \sqcap \mathcal{A} \mid \mathbf{0} \mid \mathbf{1} \mid \mathbb{A} \\
\mathcal{B} &::= \mathcal{B} \sqcup \mathcal{B} \mid \mathcal{B} \sqcap \mathcal{B} \mid \mathbf{0} \mid \mathbf{1} \mid \mathbb{B} \\
\mathcal{D} &::= \mathbb{A} \rightarrowtail \mathbb{P}.\mathbb{A} \cdot \mathcal{D} \mid \mathbb{P}.\mathbb{B} \rightarrowtail \mathbb{P}.\mathbb{A} \cdot \mathcal{D} \mid \mathbb{P}.\mathbb{B} \twoheadrightarrow \mathbb{P}.\mathbb{A} \cdot \mathcal{D} \mid \mathbb{P}.\mathbb{B} \rightarrowtail \mathbb{B} \cdot \mathcal{D} \mid \mathbf{1}
\end{aligned}$$

In general terms, the workflow input and output logics represent a simplified view of pre- and post-conditions. These can usually be described using a boolean lattice, and thus this is the approach I take. An input expression gives the subset of inputs required for a workflow to execute, whilst the output expression gives the subset of outputs offered. The operators are defined as follows:

- $A_1 \sqcup A_2$ – either $A_1$ or $A_2$ must be satisfied for the whole to be satisfied;
- $A_1 \sqcap A_2$ – both $A_1$ and $A_2$ must be satisfied for the whole to be satisfied;
- **0** – cannot be satisfied (logical false);
- **1** – satisfied immediately (logical true);
- $a$ – satisfied when input $a$ is received;
- $b$ – satisfied when the output $b$ has is satisfied by an element of the workflow.

An expression represents a set of sets of inputs or outputs, representing the possible combinations which will satisfy the workflow and allow it to execute. So $a \sqcup b$ means (inclusively) either input $a$ or input $b$ is required, corresponding to the input set $\{\{a\}, \{b\}, \{a, b\}\}$. Specifically, a workflow with the precondition $a \sqcup b$ needs at least an $a$ or a $b$ input to execute, but can also input both if available. Similarly, $a \sqcap b$ means both inputs $a$ and $b$ are required, corresponding to the singleton set $\{\{a, b\}\}$. The inclusion of **1** in the language may at first seem a little vacuous, but it is very useful for specifying *optional inputs* – inputs which are desirable but not required for execution. For instance, defining $a? \triangleq a \sqcup 1$ (i.e. an optional input $a$), a more realistic example of a workflow precondition is:

$$\mathsf{flatNo}? \sqcap \mathsf{houseNo} \sqcap (\mathsf{postcode} \sqcup (\mathsf{street} \sqcap \mathsf{city}))$$

This represents the common situation where different levels of information can be used to express street addresses. In its simplest form an address can be simply a postcode and property number, but can also contain a street name, a city name or a flat number. This expression is satisfied by members of the set $\{\{\mathsf{houseNo}, \mathsf{postcode}\}, \{\mathsf{flatNo}, \mathsf{houseNo},$

$\mathsf{postcode}\}, \{\mathsf{houseNo}, \mathsf{street}, \mathsf{city}\} \cdots \}$. The language does not include a negation operator, as this would mean an input must not be present which is difficult to check for at the operational level.

The output logic has the same structure as the input logic, although it has a rather different interpretation. It indicates the possibilities for output, for instance $a \sqcup b$ means that either an $a$ or $b$ (or both) will be output from a workflow, whilst **1** means that the workflow has no output at all. Unlike preconditions, where an input choice can be ignored, a workflow must provide behaviour for each possible output combination of a workflow. Otherwise a dead-lock may occur.

Once the inputs and outputs to a workflow have been defined, it becomes necessary to connect the various inputs and output of encapsulated performances together. $\mathcal{D}$ describes these connections. Each connection bridges two dataflow buffers in the workflow, be they a workflow input/output buffer or a performance input/output buffer. There are four possible connections:

- From a workflow input to a performance input ($a \rightarrowtail p.a$);
- From a performance output to a workflow output ($p.b \rightarrowtail b$);
- Between a performance output and another performance's input ($p.b \rightarrowtail q.a$) *asynchronously*;
- Between a performance output and another performance's input ($p.b \twoheadrightarrow q.a$) *synchronously*.

The two types of performance-performance connections are needed to complement *iteration*, which will be described in due course. An *asynchronous* connection will repeatedly pass data on, independently of the control flow. In contrast a *synchronous* connection, having passed data once, requires that the control flow *yields* before it will pass data on again.

The different connection types mirror closely the connections which can be defined in the OWL-S process model. However, as in our previous work (Norton et al., 2005) we ensure that dataflow is always completely contained within workflow scope – the contents of a workflow may only rely on its immediate inputs and outputs, and not on anything within the enclosing context (as OWL-S and other Web service languages can). This naturally aids in composability by making components fully self-contained and interface equivalent to WSMO Goals, but goes against the WS-BPEL idea of having variables accessible by all enclosed scopes.

Thus a workflow contains a set of dataflow "wires" which pass data between the components, as well as from the inputs and to the outputs of the workflow. For a workflow to complete successfully there must of course be at least one complete data pathway from the inputs, through the performances, to the outputs. This completes the discussion of the data flow language.

## 5.5 Control flow language

The control flow language defines the order of execution of a workflow's performances. It provides algebraic constructs which allow different types of flow control to be specified in a compositional way in the style of Dijkstra's guarded command language (Dijkstra, 1975) and CSP (Hoare, 1985).

Rather than define the entire control flow language $\mathcal{C}$ in one go, I will initially define it in two stages. The basic language provides the usual process algebraic operators, and a more advanced language builds on this. A third stage will define the transaction calculus, $\mathcal{T}$. The additional stages are not necessary for understanding the Cashew-A framework as a whole, and the reader may prefer to read only Section 5.5.1 and then skip over to Section 5.6 to continue the overview (returning if desired to read the other two language fragment descriptions).

A more important aim of this work is to represent the maximum number of *workflow patterns* (van der Aalst et al., 2003). Since the control flow topology will remain static, certain patterns are impossible to achieve, in particular the "Multiple Instance" workflow patterns (see Chapter 2 Section 2.2.4), where several instances of the same workflow can be spawned in parallel, sometimes undetermined in number, before runtime. These patterns are more suited to a mobile calculus such as the $\pi$-calculus , and cannot easily be modelled or verified in an inherently static calculus like CCS or CaSE. Thus all of the patterns I consider are finite state, though I will not define precisely what I mean by a "state" until Chapter 7. Instead this section will focus on giving an informal descriptive semantics to each control flow construct.

### 5.5.1 Cashew-A Core

The most basic form of the language is called *Cashew-A Core*. It is a basic control flow language, and provides the constructs normally found in an imperative programming language or process algebra. [1]

$$\mathcal{C} \quad ::= \quad \mathcal{C} \fatsemi \mathcal{C} \mid \mathcal{C} \oplus \mathcal{C} \mid \mathcal{P}\{ \mid \mathcal{P}\}^{+} \mid \mathcal{C}^{*}\mathcal{C} \mid \mathcal{P}$$

The basic language consists of the usual operators of a process algebra (cf. Baeten and Verhoef (1995)) namely, sequence $\fatsemi$, choice $\oplus$, parallel composition $\mid$ and binary Kleene star $*$, which stands for iteration with an exit condition. The behaviour of sequence $\fatsemi$ is as expected, namely the first element is executed to completion followed by the second element. The precondition of a sequential composition is simply the precondition of the first element – this allows for situations when execution of the first element enables the second. Choice is similar to that of CCS in that it is non-deterministic when both sides have equal preconditions. Otherwise execution is determined by which becomes

[1] I have taken the liberty of using $\{\cdots\}^{+}$ to indicate repetition of the enclosed item at least once.

ready first. Iteration (Kleene star) is similar to choice in that it makes a non-deterministic choice, but the left-hand side can be repeated arbitrarily often until the right-hand side is chosen. Parallel composition $P \mid Q$ allows several peformances to be executed in parallel, with synchronisation at the beginning and end.

Though Cashew-A Core is simple, it nevertheless represents a decent cross section of functionality, and includes all the control flow patterns from OWL-S except split (which does not synchronise) and anyOrder (which will be added in Section 5.5.2).

### 5.5.2 Cashew-A AC

$$\mathcal{C} \quad ::= \quad \mathcal{C} \fatsemi \mathcal{C} \mid \mathcal{C} \oplus \mathcal{C} \mid \mathcal{C} \mid \mathcal{C} \mid \mathcal{C} \parallel \mathcal{C} \mid \; |||_{\mathbb{N}} \tilde{\mathcal{C}} \mid \mathcal{C}^* \mathcal{C} \mid \uparrow \mathcal{C} \mid \circ \mathcal{C} \mid \; \upharpoonleft \mathcal{C} \mid \mathcal{P}$$

*Cashew-A AC* (Advanced Choices) includes all the constructs from the core fragment and introduces additional constructs which allow greater flexibility for choices and parallelism. There are two forms of parallel operator, the synchronous variety $\mid$ which requires that the preconditions of both sides are satisfied before executing, and the asynchronous variety $\parallel$ which only requires one to be satisfied.

This workflow language also introduces the *yield* construct $\uparrow$. A yield defines a *synchronisation point* in the workflow – all process which yield must do so at the same instant or else wait for the other processes. Yield forces the thread to wait until the next "cycle" before it can execute (similar to its concurrent programming counterparts). Although Cashew-A does not have variables for which race conditions can exist, yield can be used in conjunction with optional inputs to ensure a particular component receives a maximal amount of data from the workflow's other parts. Then, when a yield occurs, the data flow layer is purged – all performances clear their buffers – which has the effect of invalidating existing data. The one exception involves asynchronous data flow connections ($\rightarrowtail$) which ignore yields and simply hold their data until they can be passed on. Synchronous connections ($\twoheadrightarrow$) in contrast wait for the yield before accepting input. Practically this means that asynchronous connections are useful when combined with iteration, since they allow data to be passed between consecutive iterations.

Yield is also in many ways similar to a *deprioritisation operator* (Cleaveland and Hennessy, 1990) and can be used as such. For example, it can be used in deprioritising a control flow branch in a multiple choice causing it to act as a default case. Assuming $n$ cases and an additional default action, we could write:

$$(\mathsf{case}_1 \fatsemi \mathsf{action}_1) \oplus (\mathsf{case}_2 \fatsemi \mathsf{action}_2) \oplus \cdots \oplus (\mathsf{case}_n \fatsemi \mathsf{action}_n) \oplus \uparrow \mathsf{default\text{-}action}$$

The $\uparrow$ before default-action means that prior to the action being allowed to execute, all behaviour in the encapsulating workflow must cease. Thus, if one of the cases becomes ready and executes, the default branch will be ignored. Alternatively, if none of them can

become ready, the workflow activity will cease and the default action will be chosen. In a similar way yield can be used to turn the Kleene star operator into a while or until loop, where the condition is prioritised over the body. Without this construct it is impossible to build anything other than loops with a negated condition, and not loops which exit based on an event (i.e. *deferred choice* loops).

For example; consider

$$\mathsf{while}\ C\ \mathsf{do}\ P \triangleq (C \fatsemi \uparrow P)^* \uparrow \epsilon$$

Here we have a while loop consisting of a condition process $C$ and a body process $P$. If $C$ becomes ready before the yield clock can tick, the body executes, otherwise the loop exits. The additional yield preceding $P$ ensures that the dataflow layer is cleared at this point, otherwise old data could be fed into the condition causing it to be always satisfied. This way it must be satisfied by data proceeding from $P$.

We can use a similar structure to build a multiple choice, where all the workflows which become ready will be executed:

$$\mathsf{chooseMany}(\vec{P}) \triangleq (P_1 \oplus \uparrow\epsilon) \parallel (P_2 \oplus \uparrow\epsilon) \parallel \cdots \parallel (P_n \oplus \uparrow\epsilon) \ \ (\text{where } P_i \in \vec{P})$$

Yield embodies *logical time*, where a workflow synchronises when all internal work ceases. In addition to yield, Cashew-A AC also includes a wait operator $\circ P$, which embodies *physical time*. Although possessing a similar syntax to yield, wait is subtly different. Rather than protecting its parameter until the entire workflow has ceased activity, the wait operator requires one unit of time to elapse before executing the parameter. The actual physical period of a unit is determined by the workflow head, but this is only really relevant in a theoretical sense when mediating between workflows. The assumption made is that any internal communications in a workflow effectively happen instantly – it takes no time, for example, for dataflow to be propagated. Time only elapses when performances are executing, or when explicit wait states are implemented. I use the shortcut $⓷P$ to refer to a wait of $n$ units, e.g. $⓷P = \circ\circ\circ P$. We can form timeouts using the choice operator, for instance $P \oplus ②Q$ refers to a workflow which will execute $P$ unless two time units elapse before it becomes ready, at which point $Q$ may be executed (provided it is ready).

In order to take advantage of the asynchrony inherent in the semantic model, I also include a *fork* operator $↰\,C$. The fork construct, in a similar style to concurrent programming, executes a process in the background, preventing it from holding up the workflow's progress. For instance the control flow $(↰\,P) \fatsemi Q$ does not require $P$ to complete before starting $Q$, though it does at least require $P$ to be ready to execute. A forked control flow cannot live beyond the life of the workflow in which it exists. Since the semantic model exhibits *implicit termination* of workflow, it is necessary for all enclosed activity to cease, even when all control flows have declared their completedness. Furthermore, the

fork construct will only ever spawn up to one copy of the control flow, in keeping with the finite state nature of my language. If placed in a loop, a forked workflow must complete before it can be executed again.

Finally, this workflow language also introduces an n-ary workflow interleaving construct $|||_{\mathbb{N}}\tilde{\mathcal{C}}$. Known as AnyOrder in OWL-S , this construct allows the control flow to be determined solely by the process's preconditions. Effectively a graph based pattern, interleaving executes a certain number (based on the given natural number) of performances at a time from its bag, non-deterministically picking one of the remaining ready performances. It is therefore Cashew-A's equivalent of the flow construct from WS-BPEL (see Chapter 2 Section 2.2.2). Interleaving is an extremely versatile construct, but also theoretically complicated due to its reliance on dataflow. If we return to the idea that dataflow enforces a partial order on performances, it is clear that a dataflow will cause them to be resolved into a well-defined sequence. Interleaving is one of the main reasons (along with yield) why algebraic reasoning cannot be used to study Cashew-A– it effectively leads to a graph based ordering of processes. Indeed, interleaving cannot even be represented as a binary pattern, because the requirement that every process in the bag execute only once requires some form of centralised control.

### 5.5.3 Cashew-A T

$$\mathcal{T} \quad ::= \quad \mathcal{T} \fatsemi \mathcal{T} \mid \mathcal{T} \oplus \mathcal{T} \mid \mathcal{T} \mid \mathcal{T} \mid \mathcal{T} \div \mathcal{P} \mid \mathcal{P} \mid \lightning \mid \Uparrow \mid \circ \mathcal{T} \mid |*|\vec{\mathcal{P}_{\mathcal{T}}}$$

The final iteration of the language is *Cashew-A* $T$. Cashew-A T is a cut-down variant of Cashew-A AC which introduces *compensable transactions*. A transaction $\mathcal{T}$ may be wrapped in a performance and placed in a normal workflow, where it is observed like any other performance. A compensable transaction is a workflow with a normal *forward flow*, supplemented with a *compensation flow*, built from contained compensation actions, which will be executed when an exception is raised in the workflow. The compensation flow represents what happens should an error occur, and consists of compensation actions for each action executed up until the point the error occurred in reverse order. A compensation action attempts to mitigate the effects of the action.

The approach I am taking to compensation stems from the model demonstrated in cCSP (Butler, Hoare and Ferreira, 2005). A transaction is a volatile all-or-nothing region of a Web service orchestration where there exists the possibility of failure. When a performance or workflow in a transaction completes it installs a compensation action. Then, if failure occurs in the transaction before completion, all the compensations installed so far are executed in approximately reverse order. Unlike in StAC (Chessell et al., 2002), there is no manual calling of compensation, and only one compensation context per transaction. Compensation actions cannot fail; this is for the sake of simplification rather than

lack of capability[2]. All compensable workflows are self-contained – they have no effect on, nor are they effected by their environment, even if the context is another compensable workflow.

The main addition to the new control flow language, $\mathcal{T}$, is the compensation operator $\mathcal{T} \div \mathcal{P}$ which associates a process with a compensation action given by a performance. At this stage compensations are error-free, though since they can themselves contain compensable transactions a degree of hierarchy can be built. In addition, Cashew-A T also has the $\lightning$ construct which raises an exception in a workflow. My approach to raising exceptions is similar to the way yielding works. Rather than simply forcing a subworkflow to terminate where it is, I introduce a new yield construct $\Uparrow$, which yields to compensation. Unlike the normal yield construct, which guards a process and changes its precondition to force a yield, compensation yield is not a condition of execution (indeed, it has no parameter). Rather compensation yield states that at this point in the workflow, either compensation can be triggered if need-be, or else the workflow may continue. For instance, one may build a workflow:

$$(\mathsf{PickBook} \div \mathsf{UnpickBook}) \fatsemi \Uparrow \fatsemi \mathsf{OrderDelivery}$$

This workflow will not initiate compensation until after PickBook has been executed. If the workflow were parallel composed with a $\lightning$, PickBook would be executed, the workflow would then yield to compensation and finally UnpickBook would be executed. If no exception is thrown, the workflow will continue to execute OrderDelivery.

In addition to the compensation operators, Cashew-A T also has a *speculative parallelism* operator $\circledast\vec{\mathcal{P}_{\mathcal{T}}}$. Speculative parallelism is a form of concurrency where there exists multiple competing ways to achieve a goal. In order to optimise the speed at which this is accomplished, all the methods are run in parallel, with the "winner" finishing first, and others being cancelled afterward. In Cashew-A, each parallel branch is a compensable performance which can be cancelled. Cancellation forces the compensation procedure to start. Thus, the first performance to finish is effectively "committed" and the others mitigated. Speculative parallelism must allocate time fairly to all branches and therefore at the semantic level the RTC forces them into lock-step.

Note that Cashew-A T does not have a looping construct, because we wish to have a finite state semantics for our language. Looping introduces the need for "stacking" compensations when they run, which is not something my underlying process calculus can easily handle. It also doesn't have the fork $\curvearrowleft$ operator, as it would be difficult to keep track of which background processes are active and need compensating.

To aid in formally describing a compensable transaction I now define the function $\mathsf{comp} :: \mathcal{T} \to \mathcal{C}$ which gives the compensation for a given $\mathcal{T}$. To do this assume the existence of an undefined deterministic oracle $\mathsf{reduce} :: \mathcal{T} \to \mathcal{T}$ which decides unguarded choices based on the data-flow and message-flow context.

[2]Originally I set out with the intention of allowing a more liberal approach, and indeed the underlying calculus ought to be capable of it. However, time constraints prevent further development.

$$
\begin{aligned}
\mathsf{comp} &:: \mathcal{T} \to \mathcal{T} \\
\mathsf{comp}(P \fatsemi Q) &\triangleq \mathsf{comp}(Q) \fatsemi \mathsf{comp}(P) \\
\mathsf{comp}(P \mid Q) &\triangleq \mathsf{comp}(P) \mid \mathsf{comp}(Q) \\
\mathsf{comp}(P \div Q) &\triangleq Q \\
\mathsf{comp}(P \oplus Q) &\triangleq \mathsf{comp}(\mathsf{reduce}(P \oplus Q)) \\
\mathsf{comp}(\lightning) &\triangleq \epsilon \\
\mathsf{comp}(\circ P) &\triangleq \mathsf{comp}(P)
\end{aligned}
$$

This is by no means a complete definition of compensation, as it doesn't say anything about how interruption and compensation yield work (which will be dealt with in Chapter 7). What it shows is the order in which compensations will be run when a complete control-flow is compensated.

## 5.6 Components

$$
\begin{aligned}
\mathsf{Component} &::= Wf \mid CWf \mid GoalTemplate \mid Eval \mid Send \mid Receive \\
\mathsf{CComponent} &::= CWf \\
Wf &::= \mathbf{Wf}\ \mathcal{W} \\
CWf &::= \mathbf{CWf}\ \mathcal{W}_{\mathcal{T}} \\
Send &::= \mathbf{Send}\ \mathbb{M}\ \tilde{\mathbb{A}} \\
Receive &::= \mathbf{Receive}\ \mathbb{M}\ \tilde{\mathbb{B}} \\
Eval &::= \mathbf{Eval}\ Expr\ \tilde{\mathbb{A}}\ \tilde{\mathbb{B}} \\
GoalTempate &::= \mathbf{GoalTemplate}\ \mathcal{A}\ \mathcal{B}\ \phi_{ass}\ \phi_{eff}\ \mathbb{N}
\end{aligned}
$$

To complete the definition of Cashew-A I turn to the actual elements which make up a workflow, called *components*. Each component Component is a one-shot activity which maps its preconditions to postconditions by executing some internal activity, and possibly exchanging messages with a partner. A component can be a workflow $\mathcal{W}$ with its dataflow context closed, a compensable workflow $\mathcal{W}_{\mathcal{T}}$, a **GoalTemplate**, an expression evaluation **Eval**, a message **Send**, or a message **Receive**.

Goal Templates (Stollberg and Norton, 2007) are inherited from WSMO and provide the **capability** 4-tuple of precondition, postcondition, assumptions and effects. The last two are included in the language, but not yet used in any way. I assume that when given an interface semantics, the Goal Templates will be dynamically bound by a *service request broker*. Preconditions and postconditions use the same syntax as those for workflows.

Message communications are components which either consume a set of inputs and then send a message, or receive a message from the partner (if possible) and then produce a set of outputs. Importantly, a **Receive** does not declare readiness until it has received its associated message. This allows choices to be constructed based on message receipt.

An **Eval** uses its inputs to evaluate the enclosed expression. This expression can either evaluate to True and return some outputs, or it can evaluate to False, simply failing to become ready. I assume the existence of a function eval :: $Expr \rightarrow$ DfS which evaluates the expression to either **1** (True) or **0** (False). For the purpose of this Thesis I use Haskell expressions of type $(\mathtt{a_1} \cdots \mathtt{a_j}) \rightarrow$ Maybe $(\mathtt{b_1} \cdots \mathtt{b_k})$ to specify these conditions. For convenience I also define two components derived from **Eval**:

$$\begin{array}{lcl}
\textbf{Id}\ \{a^1 \cdots a^i\}\ \{b^1 \cdots b^i\} & \triangleq & (\textbf{Eval}\ \lambda \mathtt{xs}.\underline{\mathtt{Just}}\ \mathtt{xs})\ \{a^1 \cdots a^i\}\ \{b^1 \cdots b^i\} \\
\textbf{Const}\ \{b^1 \cdots b^i\}\ \mathtt{vs} & \triangleq & (\textbf{Eval}\ \lambda \mathtt{xs}.\underline{\mathtt{Just}}\ \mathtt{vs})\ \emptyset\ \{b^1 \cdots b^i\} \\
\textbf{Null}\ \{a^1 \cdots a^i\}\ \{b^1 \cdots b^j\} & \triangleq & (\textbf{Eval}\ \lambda \mathtt{xs}.\underline{\mathtt{Just}}\ \bot)\ \{a^1 \cdots a^i\}\ \{b^1 \cdots b^j\}
\end{array}$$

The first, **Id** echoes the input values onto the output values. The second, **Const**, outputs a specific tuple of outputs whose values are contained in vs. The third, **Null**, inputs an arbitrary tuple of inputs, and outputs an (unrelated) arbitrary tuple of outputs with no values.

## 5.7 Examples

In this section I describe a number of workflow examples using Cashew-A. For ease of reading, I use a slightly different syntax:

$$\mathcal{W} \ ::= \ \textbf{workflow}\ \mathbb{W}\ \textbf{\{ precondition}\ \mathcal{A}\ \textbf{postcondition}\ \mathcal{B}\ \textbf{dataflow}\ \mathcal{D}\ \textbf{control \{}\ \mathcal{C}\ \textbf{\}\}}$$

I also use implicit performance names where possible to avoid replication. Instead of writing pfPerformance[**Send**$\cdots$] for instance, I simply write **Send**$\cdots$. The structure of a performance name is pf[type][contents]. For example a performance sending a message called MyMessage has the name pfSendMyMessage. The exception is expressions which do not have a natural name. When these are used an explicit name is provided.

The first workflow describes simple address lookup component:

**workflow** wfLookupAddress
**{** **precondition** postcode $\sqcap$ houseNum
**postcondition** address $\sqcup$ **1**
**dataflow** postcode $\rightarrowtail$ pfSendReqLookup.postcode $\cdot$
houseNum $\rightarrowtail$ pfSendReqLookup.houseNum $\cdot$
pfReceiveReqLookupYes.address $\rightarrowtail$ address $\cdot$ **1**
**control**
**{** **Send** ReqAddrLookup {postcode, houseNum}$;$
(**Receive** ResAddrLookupYes {address} $\oplus$ **Receive** ResAddrLookupNo {})
**}**
**}**

This workflow takes two inputs, a postcode and house number, and outputs either an address or nothing (indicated by the **1**). The control flow simply sends a message to the partner containing the postcode, and then one of two messages are returned. If a ResAddrLookupYes (response address lookup yes), then the encapsulated address is output from the workflow, otherwise no output is forwarded. This workflow would typically be composed with a partner via a co-ordination at a higher level with a compatible service choreography.

I now turn to another example, this time using the calculator example in Figure 4.1 from Chapter 4 as a template. First, of all the outer workflow which represents the Web service as a whole can be described as:

**workflow** wfCalculator
**{** **precondition 1**
**postcondition 1**
**dataflow** recInitNum.num $\twoheadrightarrow$ pfWfOperation.inNum $\cdot$
pfWfOperation.outNum $\rightarrowtail$ pfWfOperation.inNum $\cdot$
pfWfOperation.nan $\rightarrowtail$ sendResNaN.nan $\cdot$
pfWfOperation.return $\rightarrowtail$ sendResReturn.return $\cdot$ **1**
**control**
**{** **Receive** InitNum {num}$;$
(pfWfOperation[WfOperation]$^*$(**Send** ResReturn {v} $\oplus$ **Send** ResNaN {nan}))
**}**
**}**

This workflow has no inputs or outputs, and can thus be used directly as a Web service, getting all its data via message flow. It first waits to receive the InitNum message, which supplies the initial number, and then begins execution of the Operation workflow which performs the actual calculations. The workflow is executed in a loop (Kleene star), with the exit condition being that it is possible to send either the ResReturn or ResNaN messages, because the required body for either is available from dataflow. The body of

the workflow is described below:

```
workflow wfOperation
{ precondition inNum
  postcondition outNum ⊔ nan ⊔ return
  dataflow  inNum ⤖ perfAdd.v1 ·
            inNum ⤖ perfDiv.num ·
            inNum ⤖ perfIfPos.v ·
            inNum ⤖ perfReturn.v ·
            recReqAdd.val ⤖ perfAdd.v2 ·
            recReqDiv.val ⤖ perfDiv.dem ·
            recReqIfPos.p1 ⤖ perfIfPos.p1 ·
            recReqIfPos.p2 ⤖ perfIfPos.p2 ·
            perfAdd.res ⤖ outNum ·
            perfDiv.res ⤖ outNum ·
            perfIfPos.res ⤖ outNum ·
            perfDiv.nan ⤖ nan ·
            perfReturn.res ⤖ return · 1
  control
  { (Receive ReqAdd {val} ⨟ perfAdd[WfAdd]) ⊕
    (Receive ReqDiv {val} ⨟ perfDiv[WfDiv]) ⊕
    (Receive ReqIfPos {p1, p2} ⨟ perfIfPos[WfIfPos]) ⊕
    (Receive ReqReturn ∅ ⨟ perfReturn[Eval (λx.Just x) {v} {res}])
  }
}
```

The Operation workflow inputs a number to calculate with inNum. It outputs either an output (intermediate) result, an error (not a number), or a final returned value. The control flow of the workflow consists of a four way choice, each branch of which is guarded by the receipt of particular command message: either add, divide, if-positive (i.e. *if* inNum *is positive then* v1 *else* v2) or return. Each message guards a performance which executes the respective operation. All but one of them are enclosed workflows, the exception being return which is an identity evaluation on the current intermediate result.

## 5.8 Workflow Patterns

In this Section I review Cashew-A with respect to the revised workflow patterns (van der Aalst et al., 2006). Readers unfamiliar with the workflow patterns may desire to skip this section, or just read Section 5.8.8 which presents my conclusions.

The authors' intent in creating the workflow patterns is to find the fundamental patterns present in workflow languages. They do this through a mixture of reviewing ex-

isting languages and software, and theoretical analysis using Petri-nets (for a fuller discussion see Chapter 2 Section 2.2.4). There are a total of 43 enumerated control-flow patterns, and the intention of this section is to see how Cashew-A compares. I will not provide a formal mapping, nor is this work complete, but it illustrates of how general Cashew-A is. Throughout this section I will reference the patterns with the name and number given in the paper and on the workflow patterns website[3], for instance Sequence (1).

However, a number of caveats must first be considered. First, since I am are using an immobile or static calculus to give the language a semantics, multiple-instance patterns (that is patterns which can spawn additional copies of themselves in the style of $\pi$-calculus's $!P$) are not possible. As a result, patterns (12)–(15) and (34)–(36) are not directly representable. Secondly, since the workflow patterns are given in a graph structured paradigm and Cashew-A is block-structured, splits and joins are not considered as separate patterns. For instance, Parallel Split (2) is combined with Synchronisation (3) within the $\mid$ construct. Nevertheless, the different styles of synchronisation and merges can be recaptured in Cashew-A through the use of dataflow inputs and outputs, along with the dataflow oriented control-flow constructs.

### 5.8.1 Basic Control Flow Patterns

| | | |
|---|---|---|
| 1. | Sequence | $P \fatsemi Q \fatsemi R \fatsemi \cdots$ |
| 2+3. | Parallel Split + Synchronisation | $P \mid Q \mid R \mid \cdots$ |
| 4+5. | Exclusive Choice + Simple Merge | $(\mathbf{Eval}\ Condition_1 \fatsemi P) \oplus$ $(\mathbf{Eval}\ Condition_2 \fatsemi Q) \oplus \cdots$ |

These are "bread-and-butter" workflow patterns and should not require verbose explanation, suffice it to say they are clearly represented. Exclusive choice uses a collection of disjoint logical expressions to make the decision. Also notice that a parallel split converges with a synchronisation, whilst choice converges with a merge.

### 5.8.2 Advanced Branching and Synchronization Patterns

This section (much expanded in van der Aalst et al. (2006)) contains various patterns for different styles of parallel branching and synchronisation. It includes constructs such as taking multiple branches of a choice in parallel.

[3] http://www.workflowpatterns.com

| | | |
|---|---|---|
| 6+7. | Multi-Choice + Structured Synchronizing Merge | $(P \oplus \uparrow\epsilon) \parallel (Q \oplus \uparrow\epsilon) \parallel (R \oplus \uparrow\epsilon) \cdots$ |
| 2+9. | Parallel Split + Structured Discriminator | $(\curvearrowleft P \parallel \curvearrowleft Q \parallel \curvearrowleft R) \fatsemi S$ with dataflow |
| 2+29. | Parallel Split + Cancelling Discriminator | $\|*\|\{P, Q, R\}$ |
| 2+30. | Parallel Split + Structured Partial Join | $(\curvearrowleft P \parallel \curvearrowleft Q \parallel \curvearrowleft R) \fatsemi S$ with dataflow |

- **Multi-Choice** (6) (alt. N-out-of-M split) is combined with Structured Synchronising Merge (7), which together allow several branches of the choice to be executed simultaneously, and all branches to synchronise before flow can continue. This is achieved via asynchronous parallel composition of several choices between running or not running each task, based on whether it becomes ready first or yields, in which case a skip is executed.

- **Multi-Merge** (8) is not directly possible, since it requires the activation of the proceeding workflow once for each completion of the branches (similar to multiple-instance patterns).

- **Structured Discriminator** (9) enables the continuation of activity following a parallel split when at least one thread completes, and can best be achieved with dataflow. All the parallel branches should be forked so they run in the background, and there should be a dataflow dependency between each of the branches and the continuation.

- **Blocking Discriminator** (28) is related to the context of multiple instances, and so is not possible.

- **Cancelling Discriminator** (29), which cancels all preceding threads once one has completed, is equivalent to the speculative parallelism construct from Cashew-A T.

- **Structured Partial Join** (30) is a more general version of Structured Discriminator (9), which allows continuation after a certain subset of branches has completed. Once again, this is best achieved using dataflow in the same way as the standard structured discriminator.

- **Local Synchronizing Merge** (37) allows convergence and synchronisation of a subset of split branches chosen from a bag, where this subset is defined at runtime. It is supported directly through the use of the interleaving construct and dataflow connections.

None of the remaining 6 patterns in this Section (patterns 31, 32, 33, 38, 41 and 42) can be supported directly, mainly due to their relatedness to multiple-instances, therefore I do not consider them.

### 5.8.3 State-based

Having moved over the Multiple-Instances section because none of the patterns can be directly supported, I now approach the patterns associated with state. The workflow patterns website describes these so:

> *State-based patterns reflect situations for which solutions are most easily accomplished in process languages that support the notion of state. In this context, we consider the state of a process instance to include the broad collection of data associated with current execution including the status of various activities as well as process-relevant working data such as activity and case data elements.* [4]

Thus the patterns in this Section relate to decisions related to the internal state of a workflow. In the context of Cashew-A, they are specifically related to dataflow and message flow.

| | | |
|---|---|---|
| 16+5. | Deferred Choice + Simple Merge | $P \oplus Q \oplus R \oplus \cdots$ |
| 17/40+7. | Interleaved (Parallel) Routing + Structured Synchronizing Merge | $\|\|\|_1\{P, Q, R, \cdots\}$ |
| 18. | Milestone | Dataflow |

**Deferred Choice** (16) is, due to the nature of the precondition system, the equivalent of the basic choice operator. Specifically, the basic choice operator begins to execute when either side is ready, and therefore has a great deal flexibility in application. It fits in well with this description:

> *The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.*[5]

The first task to declare readiness is the one executed (the race winner), whilst the others are terminated. Readiness can, for instance, be based on whether an enclosed message-receive is ready – i.e. when it has received the message. This assumes that the choice options $P, Q$ and $R$ are all guarded by an atomic task which makes the decision.

**Interleaved Parallel Routing** (17) and the simpler **Interleaved Routing** (40) represent a collection of processes with an implicit partial order between them which dictates the execution order. The latter also allows an arbitrary order to be assigned. Both are

[4] See `http://www.workflowpatterns.com/patterns/control/index.php`
[5] See `http://www.workflowpatterns.com/patterns/control/state/wcp16.php`

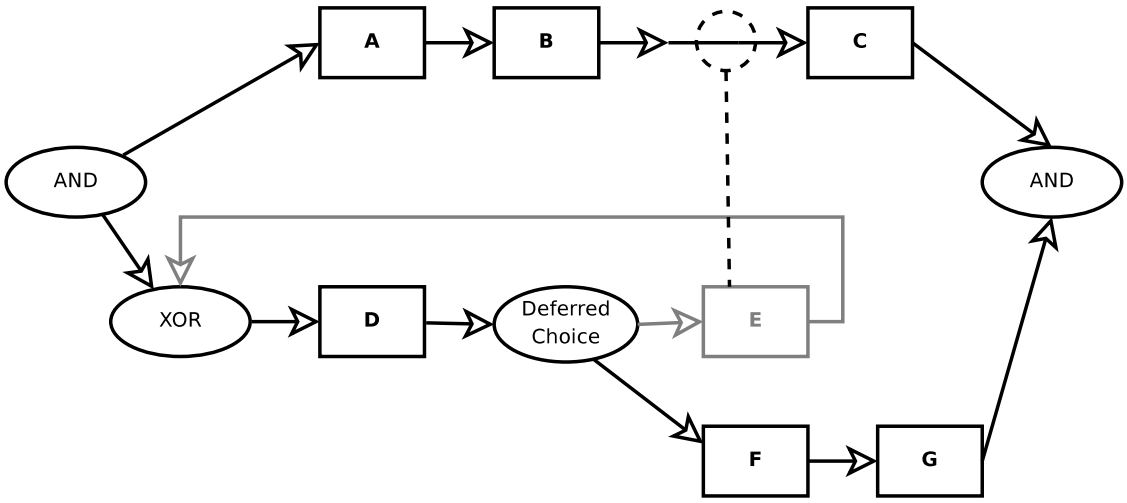



Figure 5.3: An example milestone workflow taken from the workflow patterns site.

achieved directly via the interleaving construct present in Cashew-A, constrained to one process executing at a time. The difference is simply that the former must have an associated dataflow to determine a specific order, while the latter is truly non-deterministic (or random).

**Milestone** (18) requires that a parallel branch have a region which is impassable until another parallel branch has reached a specific point (the milestone). This can be achieved directly through the setting up of dataflow connections. For instance in Figure 5.3 a workflow is shown[6] where the loop beginning with **E** can only execute when the milestone between **B** and **C** is active. Otherwise only **F** can be executed. We can adapt this workflow to Cashew-A thus:

$$A \fatsemi B \fatsemi (\mathsf{pt}[\mathbf{Null}\ \emptyset\ \{t\}] \fatsemi \uparrow\epsilon)^* C \parallel D \fatsemi ((\mathsf{qt}[\mathbf{Null}\ \{t\}\ \emptyset] \mid E \fatsemi \uparrow D)^* F) \fatsemi G \times \mathsf{pt.t} \rightarrowtail \mathsf{qt.t} \cdot Df$$

Here we have a workflow with a loop whose continuation is predicated on performance qt receiving t, which comes from the parallel branch. The input is passed on by the first process and then a yield occurs which clears the input. If **C** executes, the loop passing on the input becomes inactive, and therefore the other loop cannot progress any further.

**Critical Section** (39) requires that several areas of a workflow be mutually exclusive, so that only one may execute at a time. This cannot be achieved in Cashew-A, as there is no way of temporarily disabling a performance. One possible way to do this might be through some form of negative preconditions, where the presence of an input would exclude a component from executing. For the time being, this is deemed out of the scope of this work, as it would require a great deal of effort and possibly a major overhaul of the precondition system.

[6]Taken from `http://www.workflowpatterns.com/patterns/control/state/wcp18.php`

### 5.8.4 Cancellation and Forced Completion Patterns

These patterns relate to the premature termination of a workflow. There are five variants of cancellation, two of which are multiple instance and so are ignored for the reasons I explained above. The remaining three are:

- **Cancel Task** (19)
- **Cancel Case** (20)
- **Cancel Region** (25)

Only the last two are directly supported, as in the transaction language Cashew-A T it is not possible to terminate a performance once it has started. Cancel Region is roughly equivalent to compensation of a workflow using the throw $\lightning$ construct, though the workflow patterns specify that even non-block structured regions should be cancellable.

### 5.8.5 Iteration

| | | |
|---|---|---|
| 21. | Structured Loop | $P^*Q$ |

There are three forms of iteration in the workflow patterns, **Arbitrary Cycles** (10), **Structured Loop** (21) and **Recursion** (22). Structured Loop is supported directly through the Kleene Star operator of the language. The Arbitrary Cycles pattern isn't supported and can probably never be supported in a block structured language, as it requires that loops be set up between arbitrary parts of a control-flow. Recursion also isn't possible.

### 5.8.6 Termination Patterns

| | | |
|---|---|---|
| 11. | Implicit Termination | Directly supported through workflows |

**Implicit Termination** (11) is a property of a workflow, where when all enclosed activity has completed the workflow terminates. It is included purely as a consequence of using Timed Process Calculus with maximal progress to give the operational semantics. **Explicit Termination** (43) is when a workflow ends at a specific point, even if internal activity still remains. It is only partially supported and only within compensable workflows.

### 5.8.7 Trigger Patterns

| | | |
|---|---|---|
| 23/24. | Persistent/Transient Trigger | Dataflow connections act as triggers |

Persistent and Transient triggers can both be represented in Cashew-A, though they are both a little complex due to the low-level nature of the language. A persistent trigger

causes a process to be enabled only after a specific event occurs. A transient trigger is the same, but the process will only be activated for a limited time afterwards.

A persistent trigger can be represented by a parallel looping process which waits for an event to occur, and then ensures that a particular dataflow input is made available to act as the trigger for another process. For instance, consider

$$\begin{aligned}(&\mathbf{Receive}\ \mathsf{m}\ \emptyset\ ;\ (\ (\circ \mathsf{t1}[\mathbf{Const}\ ()\ \{\mathsf{t}\}])^{*}\mathsf{dti}[\mathbf{Null}\ \{\mathsf{t}\}\ \emptyset] \\ &\qquad\qquad \parallel \uparrow \mathsf{dto}[\mathbf{Null}\ \emptyset\ \{\mathsf{t}\}]) \\ &^{*}\mathsf{tt}[\mathbf{Null}\ \{\mathsf{t}\}\ \emptyset])\end{aligned}$$

If this were composed with a workflow (which does not yield), and a dataflow connection is made from t1.t to a mandatory input of triggered processes, then this would act as a persistent trigger on receiving message m. When the message is received, two processes run in parallel. The first repeatedly executes the **Const** performance, which ensures that the trigger is maintained. The loop exits when the second parallel process is activated following the yield (a dataflow connection dto.t $\rightarrowtail$ dti.t is required). The trigger runs in a loop and must be deactivated by populating the input t of tt at the end of the workflow.

The transient trigger is represented similarly:

$$\begin{aligned}(&\mathbf{Receive}\ \mathsf{m}\ \emptyset\ ;\ (\ \circ\, \mathsf{t1}[\mathbf{Const}\ ()\ \{\mathsf{t}\}]^{*}\mathsf{dti}[\mathbf{Null}\ \{\mathsf{t}\}\ \emptyset] \\ &\qquad\qquad \parallel \textcircled{\scriptsize v}\mathsf{dto}[\mathbf{Null}\ \emptyset\ \{\mathsf{t}\}]) \\ &^{*}\mathsf{tt}[\mathbf{Null}\ \{\mathsf{t}\}\ \emptyset])\end{aligned}$$

The difference lies in the conditions under which the trigger is removed. Rather than a yield, in this case it happens when $v$ time units have passed in the workflow. This results in a trigger which is only available temporarily.

### 5.8.8 Analysis

Of the 43 workflow patterns identified by van der Aalst et al. (2006), Cashew-A directly supports 20. At first sight this seems a fairly unimpressive score, but it compares favourably with other Web service languages as reviewed by van der Aalst et al. (2006). For instance, Oracle's implementation of WS-BPEL scores 22, and the other WS-BPEL implementations score very comparably. Of the 14 workflow languages and implementations reviewed, the average score was 21, with the highest score being 33, tied between XPDL (XML Process Definition Language) and BPMN (Business Process Modelling Notation). No language evaluated supports the full 43 patterns.

Nevertheless, it is perhaps more instructive to see which patterns BPEL4WS 1.1 supports, which Cashew-A doesn't, since this language is very much an inspiration for my

work. Cashew-A supports Structured Discriminator (9) because of its precondition system, which allows an OR precondition to be specified in dataflow. In BPEL4WS 1.1, this isn't possible, as all incoming links must be known before an activity can begin executing. Multiple Instances (12) with no synchronisation could in theory be expressed in Cashew-A by using an external Web service to drive the additional process instance (albeit this isn't instrisically part of the language). Milestone (18) can be achieved in Cashew-A, it seems, primarily because of the very general choice operator. In BPEL, a deferred choice can only be built using the pick construct, which is only intended for choices based on an event. Cancel Task (19) cannot as yet be supported in Cashew-A, because the transaction system is fairly basic, whereas WS-BPEL is well featured in this respect. Cashew-A supports both transient and persistent triggers, since dataflow is inherently transient and therefore can easily be removed, or alternatively maintained. The remaining three missing patterns (39,41 and 42) are unsupported because they are multiple instance based (or multi-threaded).

I therefore contend that Cashew-A compares positively with BPEL4WS 1.1 and workflow languages in general. Furthermore it is likely that additional patterns could be supported with additional work without changing the overall model. For instance, Cancelling Partial Join (32) is essentially speculative parallelism, but with several winners, rather than just one. However, it must also be noted that WS-BPEL 2.0 has many extensions, including some $\pi$-calculus based constructs. Therefore it certainly supports additional workflow patterns. But bearing in mind that Cashew-A has a comprehensive formal under-pinning (described in Chapter 7) the score is satisfactory.

## 5.9 WS-BPEL Comparison

In Section 5.8 I compared the workflow control patterns of (van der Aalst et al., 2006) with Cashew-A and argued that a reasonable subset are representable. In this final section I will provide show that several of the more interesting constructs in WS-BPEL are also representable in Cashew-A. The purpose of this exercise is to show that Cashew-A is also comparable with a real-world Web service language.

I cannot as yet provide a complete mapping between the two languages, as there are a great many small technicalities to work out before this is possible. Foremost of these is how WS-BPEL's mutable variables should be handled using Cashew-A's dataflow paradigm. A complete mapping is clearly impossible, due to many of WS-BPEL's $\pi$-calculus oriented features. For instance the following constructs will never be representable:

- Parallel **forEach** (replication in $\pi$), representing the replication in parallel of a WS-BPEL process according to some universally quantified expression;
- Partner Links (channel passing in $\pi$), the ability to talk to several Web services simultaneously through parametrised link passing.

| BPEL Construct | Cashew-A Representation |
|---|---|
| sequence$\langle P, Q..\rangle$ | $P \fatsemi Q \fatsemi ..$ |
| if $e$ then $P$ else $Q$ | (ifcon $\left[\mathbf{Eval}\ (\mathbf{if}\ e\ \mathbf{then}\ (\underline{\texttt{Just}}\ ())\ \mathbf{else}\ \underline{\texttt{Nothing}})\ \mathsf{fv}(e)\ \emptyset\right] \fatsemi P) \oplus \uparrow Q$ |
| while $e$ do $P$ | (wcon $\left[\mathbf{Eval}\ (\mathbf{if}\ e\ \mathbf{then}\ (\underline{\texttt{Just}}\ ())\ \mathbf{else}\ \underline{\texttt{Nothing}})\ \mathsf{fv}(e)\ \emptyset\right] \fatsemi \uparrow P)^* \uparrow \epsilon$ |
| repeat $P$ until $e$ | $P \fatsemi (\uparrow P)^*$(rcon $\left[\mathbf{Eval}\ (\mathbf{if}\ e\ \mathbf{then}\ (\underline{\texttt{Just}}\ ())\ \mathbf{else}\ \underline{\texttt{Nothing}})\ \mathsf{fv}(e)\ \emptyset\right]$) |

Table 5.1: Basic BPEL constructs

Nevertheless, for simpler WS-BPEL processes where communication is limited to one partner at a time in a finite state system, a certain degree of mapping ought to be possible. For the time being, I concentrate on certain useful aspects of the WS-BPEL language.

One of my motivations for Cashew-A is that it should have the ability to support as many forms of WS-BPEL workflow as possible, so, we first examine how WS-BPEL's Structured Activities can be represented. These mappings only deal with dataflow in so far as it is required for executing the correct workflow sequence, otherwise it is glossed over. I first consider the basic programming constructs in Table 5.1 (for the sake of convenience I am using Haskell as an expression language).

Sequence is, of course, directly supported. If-then-else is supported via a choice between evaluating the expression $e$ and then executing the then clause process $P$. The if clause consists of a performance called ifcon (the if condition) containing an evaluation component, which decides if the expression (within the implicit dataflow context) evaluates to true or not. If the expression $e$ evaluates to true, the performance will execute to completion and allow $P$ to execute. Otherwise, ifcon will stall and a yield will occur, activating the else clause process $Q$.

A while loop (i.e. repeat the process execution while an expression evaluates to true) is similar to if-then-else, though using the binary Kleene star operator instead of the choice operator. The expression is wrapped into an evaluation performance called wcon and guards the process $P$ on the left hand side. The exit condition on the right hand side is a yield followed by a skip. This means that as long as the expression $e$ can be evaluated successfully, the yield will not happen and $P$ will be repeated. Otherwise the loop will exit.

Repeat-until is like while, except that the condition is now on the RHS, and the body

| **BPEL Construct** | **Cashew-A Representation** |
|---|---|
| pick<br>onMessage$\langle m^1, P^1\rangle$<br>$\cdots$<br>onMessage$\langle m^i, P^i\rangle$<br>onAlarm$\langle t^1, Q^1\rangle$<br>$\cdots$<br>onAlarm$\langle t^j, Q^j\rangle$ | (rec$m^1$ $\left[\textbf{Receive}\ m^1\ \mathsf{parts}(m^1)\right] \fatsemi P^1) \oplus \cdots \oplus$<br>(rec$m^i$ $\left[\textbf{Receive}\ m^i\ \mathsf{parts}(m^i)\right] \fatsemi P^i) \oplus$<br>$\textcircled{t^1} Q^1 \oplus \cdots \oplus \textcircled{t^j} Q^j$ |

Table 5.2: The pick construct

must execute at least once. Thus, we first sequentially compose the initial execution of $P$. The process then executes $P$ repeatedly, but this time only provided a yield can occur first (i.e. $\uparrow P$). The yield will not occur if the expression performance rcon evaluates to true, meaning that the loop exits at this point.

I now turn to the more advanced, Web service oriented constructs. First, in Table 5.2, a pick in WS-BPEL is a particular type of choice, which is resolved by an event, which can be either a timeout or a message receipt. It is useful, for instance, to allow a client Web service to make a choice (i.e. external choice). It is less general than the generic choice operator in Cashew-A and hence easily representable. As Cashew-A has both message flow and a real-time clock, both types of events can be represented. A receipt of message $m_i$ is based on a **Receive** component wrapped in an appropriately named performance, which guards the process which should become active when this message is received. A timeout is represented using the RTC operation $\textcircled{n}P$, which will activate $P$ after $n$ time units elapse, unless one of the other events happens first.

A sequential forAll construct, shown in Table 5.3 is like a for loop, in that it executes the body with a free variable, where the variable is populated on each iteration by an incremented number. Unlike the previous examples I use a complete workflow to represent a forAll loop. The inputs to the workflow are set to the inputs of $P$, plus an additional input n for the number of times to execute.

The control flow $C$ includes a performance init which executes first and assigns the initial number to the accumulator (v=0) by providing an output called $v$. A repeat loop then continues executing the body $P$ until the current loop value $v$ reaches $n$. The body yields (indicating the condition isn't yet true), executes $P$ and then runs a performance called inc which inputs the current iteration variable $v$ and the final value $n$, increments $v$ and then outputs them both as $w$ and $m$ respectively. These two values are then fed back into the workflow through the dataflow for the second iteration.

The exit condition uses a performance called test which inputs the current iteration variable $v$ and the final value $n$, and checks if $v > n$. If so the yield will not be allowed

| BPEL Construct | **Cashew-A** Representation |
|---|---|
| forAll$\langle P, n\rangle$<br>(sequential) | forAll$[A\{C \times D\}B]$ where<br>$A = \mathsf{n} \cdot \mathsf{inputs}(P)$<br>$B = \mathsf{outputs}(P)$<br>$C = \mathsf{init}[\mathbf{Const}\ \{v\}\ (0)]\ ;$<br>$(\uparrow P\ ;\ \mathsf{inc}[\mathbf{Eval}\ (\underline{\texttt{Just}}\ (v+1, n))\ \{v, n\}\ \{w, m\}])^*$<br>$(\mathsf{test}[\mathbf{Expr}\ (\mathbf{if}\ (\mathtt{v}{>}\mathtt{n})\ \mathbf{then}\ \underline{\texttt{Just}}\ ()\ \mathbf{else}\ \underline{\texttt{Nothing}})\ \{v, n\}\ \emptyset])$<br>$D = \mathsf{n} \rightarrowtail \mathsf{inc.n} \cdot \mathsf{n} \rightarrowtail \mathsf{test.n} \cdot \mathsf{test.m} \rightarrowtail \mathsf{test.n} \cdot$<br>$\mathsf{init.v} \rightarrowtail \mathsf{test.v} \cdot \mathsf{init.v} \rightarrowtail \mathsf{inc.v} \cdot \mathsf{inc.w} \rightarrowtail \mathsf{test.v} \cdot \mathbf{1}$ |

Table 5.3: The forAll construct

| BPEL Construct | **Cashew-A** Representation |
|---|---|
| flow<br>link $\langle l^1\rangle$<br>link $\langle l^2\rangle$<br>...<br>link $\langle l^i\rangle$<br>(.. workflow ..) | flow$[A\{\mathsf{Links} \parallel (\mathsf{Flow}\ ;\ \mathsf{End}) \times D\}B]$ where<br>$\mathsf{Links} = l^1[\mathbf{Id}\ \{\mathsf{il}\}\ \{\mathsf{ol}\}]^*(\mathsf{e\_}l^1[\mathbf{Null}\ \{\mathsf{il}\}\ \emptyset])$<br>$\parallel\ l^2[\mathbf{Id}\ \{\mathsf{il}\}\ \{\mathsf{ol}\}]^*(\mathsf{e\_}l^2[\mathbf{Null}\ \{\mathsf{il}\}\ \emptyset])$<br>$\parallel\ \ldots$<br>$\parallel\ l^i[\mathbf{Id}\ \{\mathsf{il}\}\ \{\mathsf{ol}\}]^*(\mathsf{e\_}l^i[\mathbf{Null}\ \{\mathsf{il}\}\ \emptyset])$<br>(Flow and End described below in text) |

Table 5.4: The flow construct

and the loop will exit.

The final WS-BPEL construct I consider is flow, shown in Table 5.4. A WS-BPEL flow is a collection of workflows which run in parallel, together with a set of links which give a dependency relation between their activities. This is quite complicated to represent in Cashew-A, since the links can target and source processes which are some way down the encapsulated control-flow hierarchy. I use Flow to represent the unlinked control flow in the workflow and then compose it with the 'links' process called Links and an End process to clean up at the end.

This is essentially a simplified Petri-net pattern. Each link is represented as a looping performance called $l^n$, which copies the input to the output. Each performance has an input called il and an output called ol, which are used to source and target particular performances in Flow via dataflow $D$. For instance if we wish to link a performance $p$ to a performance $q$ with link $l_n$ then $D$ would have links $\mathsf{o}^p \rightarrowtail \mathsf{il}^{l_n}$ and $\mathsf{ol}^{l_n} \rightarrowtail \mathsf{i}^q$. The process End is there to terminate all the link loops when the main flow terminates.

I claim, therefore, that Cashew-A is capable of representing many of the constructs in

WS-BPEL. Although this work is by no means complete, it demonstrates that Cashew-A is a language which is useful for Business Process Modelling.

## 5.10 Conclusion

In this Chapter I have given a detailed, but informal description of the service composition language, Cashew-A. I described the overall paradigm, that an orchestration is described in terms of control flow, data flow and message flow. I described three control language fragments, including one for specifying compensable transactions. I then gave some examples of Cashew-A orchestrations and evaluated the language against a comprehensive set of workflow patterns and WS-BPEL. What I have demonstrated is that my approach of combining data flow with abstract and physical time, along with a pre-condition / post-condition layer entails a highly versatile language which is comparable with recent Web service and Business Process Modelling languages. In the following chapters I will equip the language with a formal behavioural semantics.

# Chapter 6

# A Timed Process Calculus for Component Oriented Systems

*In this Chapter define the behavioural meta-model for Cashew-A, the service composition language introduced in the previous chapter. I introduce a novel* ***Abstract Timed Process Calculus*** *called CaSE$^{ip}$, which will be used to equip Cashew-A with a fine-grained behavioural semantics, the bottom-level of the three-level semantic stack presented in Chapter 4. The calculus generalises an existing timed process calculus, CaSE, and allows the representation of a greater number of synchronisation patterns. I fully explore the calculus and develop an equivalence theory.*

## 6.1 Motivation

IN THIS CHAPTER I construct a behavioural meta-model for Cashew-A. The underlying meta-model is an *Abstract Timed Process Calculus* called CaSE$^{ip}$, which I will introduce and formally describe. However, I first justify my choice of a Timed Process Calculus to model Web service composition. At this point it must be stressed that the purpose of this exercise is not to model Web services from a *physical* perspective. That is, the calculus I introduce does not deal directly with low-level networking issues such as timeout and availability. There are already several calculi which take this route, for instance *Orc* (Misra and Cook, 2007) and several $\pi$-calculus extensions. Nor is this an attempt to model real-time aspects of Web services, for which again several calculi already exist. Rather the use of this calculus is purely from an abstract *component* perspective, and undoubtedly the Web service architecture encapsulates one of the most important component paradigms of the moment. The "time" which I speak of is *relative* time – the ordering of different events with respect to each other.

My calculus will provide a basis for modelling different *synchronisation constraints* between components of a Web service orchestration. Such constraints will allow components to be orchestrated according to different patterns of execution. My contention

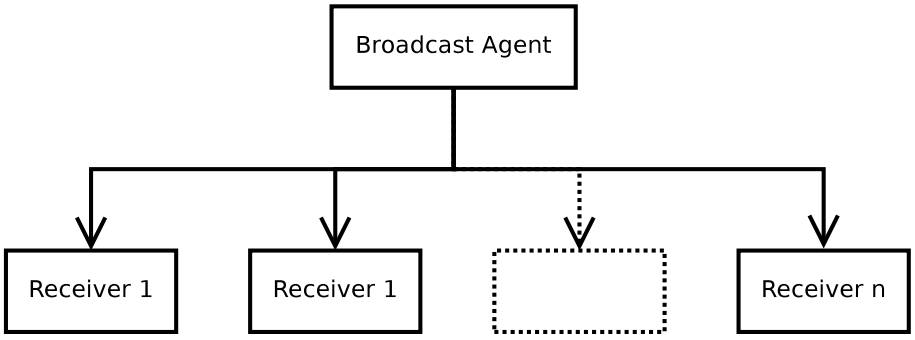



Figure 6.1: Isochronic Broadcast

is that the different patterns of Chapter 5 may be modelled via a variety of synchronous *protocols*, and I will build a calculus to exhibit these. The crucial factor is *compositionality* – the calculus model into which Cashew-A can be translated, and also provide on-the-fly evolution. There will be two aspects to the meta-model, modelling the internal synchronisations between various components and external communication between the Web service and client.

Abstract Timed Process Calculus is itself well-suited to component modelling because of the many different synchronisation concepts that may be represented. One such concept already mentioned in Chapter 2 is *isochronic broadcast*, which models a broadcast between an unbounded number of recipients as shown in Figure 6.1. This sort of broadcast contrasts with that of CSP (Hoare, 1985) in that it is *deterministic* – there is an implicit guarantee that all agents within a given scope must participate. Isochronic broadcast therefore allows a compositional approach to dataflow modelling, where data synchronisations can be made without prior knowledge about recipients. Hence, a dataflow connection is guaranteed to pass its incoming data on to all recipients. Isochronic broadcast can therefore also be used to schedule an unbounded collection of processes.

The variant of process calculus I focus on is particularly suited to component modelling due its inherent notion of *synchronous hierarchies*. This concept, first suggested by Kick (1999) and Lüttgen (1998), forms a major part of the CaSE process calculus, created by Norton et al. (2003). CaSE allows a process topology (i.e. a collection of parallel sequential agents) to be divided into a hierarchy of compartments, each of which limits the scope of a multi-party clock synchronisation. Inherent in this is a notion of *priority*, where actions from inner compartments pre-empt those from outer compartments as illustrated in Figure 6.3. This concept, which from a theoretical standpoint is facilitated by the *clock hiding* operator, allowed us to use clocks to detect completion of encapsulated components in a workflow; see our earlier paper (Norton et al., 2005). Since Cashew-A also allows for *compensable transactions*, this model will also need a form of *localised interruption* where processes in a limited scope may have their execution halted and associated compensation processes started, as illustrated in Figure 6.2.

Perhaps the most important pattern we have successfully described from the OWL-S

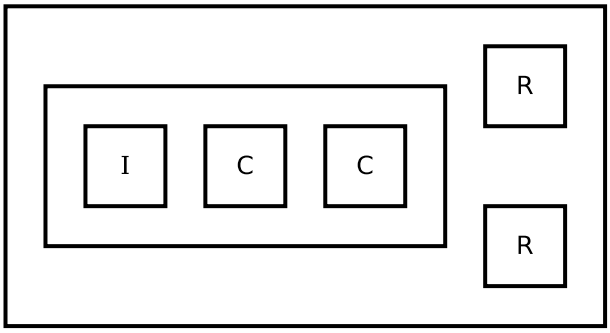



Figure 6.2: The process marked I interrupts the two processes marked C. The other processes marked R are unaffected.

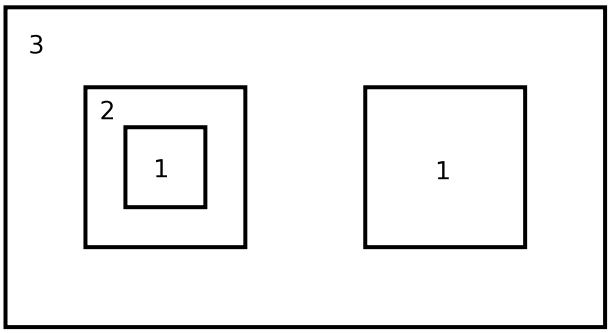



Figure 6.3: Synchronous Hierarchies
(Lower numbered compartments have higher priority.)

process model using CaSE is *interleaving*, known as AnyOrder. This pattern represents a bag of processes whose execution order is determined purely by their individual preconditions, with the added constraint that only one process may execute at any instant. Completion of the pattern as a whole requires that each of the processes execute once. In addition there may be independent dataflow between processes, allowing the resolution of an ordering. In our semantics each constituent process and dataflow connection is represented by a CaSE process. Completion is detected when an encapsulating clock ticks, which cannot occur until all internal activity has been exhausted. This is impossible to represent in a compositional way with CCS alone, since there would be no way of independently determining when the final process had finished without storing an external process list (which is non-compositional). Hence the addition of abstract time to CCS is imperative to model the workflow patterns.

This Chapter is, to a greater or lesser extent, the result of an exploration into what other forms of constraints can be modelled in an abstract timed process calculus. In Chapter 5 I demonstrated several types of workflow pattern which Web service composition languages include. It is my contention that timed process calculus provides a useful abstract theory for studying these patterns. Further, I believe such a process calculus provides a stepping stone in refining an abstract composite Web service specification into executable form. In this Chapter I take the initial steps to backing up my contention.

## 6.2 Overview

Although CaSE is already a very powerful calculus and has been shown to be useful for modelling service composition, I have decided to extend it. During my experiments with CaSE in attempting to give a semantics to several of the operators from Chapter 5 it became apparent that a number of additional concepts not found in OWL-S are either cumbersome or impossible to represent in a compositional way. I also decided that CaSE required some further refinement and generalisation from a theoretical perspective. An essential feature of the new calculus is that, like its predecessors, it will be equipped with a congruent equivalence relation based on weak bisimulation. Weak bisimulation is the variant of bisimulation which abstracts over internal actions, and therefore allows a component composition to be viewed at different levels of abstraction. This is a vital feature for it to be suitable for component composition, without which it would be impossible to compare interfaces and perform process decomposition.

Before looking at the process calculus theory it is necessary to briefly examine the context of the meta-model. A key assumption throughout this chapter and the remainder of the thesis will be: **a process is a composition of sequential agents**. Specifically, I am limiting myself to the sub-language of CCS where static operators may not be placed within dynamic operators . For instance the process $\mu X.(P|X)$, representing the replication of process $P$ will not be allowed. This is a sensible restriction to make since I am only concerned in this work with finite-state processes. I will not, therefore, be mapping the operators of Cashew-A directly onto operators of my process calculus, but rather will be representing their behaviour through a collection of synchronous agents.

Indeed, it has been noted previously by van der Aalst (2005) that the operators of CCS and $\pi$-calculus do not naturally fit in with workflow operators. That is true, because CCS (and perhaps even $\pi$), being a low-level theory, is more suited to providing a behavioural *meta-model* for workflow, where a variety of synchronous protocols express the behaviour of individual processes. Therefore, the calculus will be used to express the fundamental concepts of control flow and data flow via synchronous communication and abstract time. The advantage of this approach is a fine-grained compositional semantics, which precisely describes the execution of a workflow. The semantics of a Cashew-A process like $[\![P \fatsemi Q]\!]$ will be described (broadly) as $[\![\fatsemi]\!] \mid [\![P]\!] \mid [\![Q]\!]$, that is the semantics of the two processes parallel composed with the semantics of the operator, which will be some sort of scheduler .

Having discussed the context of the new calculus I explore in the next two sections some of the issues which CaSE$^{\text{ip}}$ seeks to address.

### 6.2.1 Interruption

The first problem which I considered is that CaSE apparently lacks any form of support for *interruption*. Interruption is a key concept in Cashew-A transactions (see Chapter 5

Section 5.5.3). When a workflow member wishes to invoke a compensation it must first interrupt the rest of the workflow using the $\lightning$ operator. From a process perspective, this must stop the activity of all workflow sub-processes and initiate the roll-back. The question is, how should this be represented at the process level? My first approach was to judge that compensation should take place with greater priority than simply continuing the workflow's normal execution. Hence, interruption may be viewed as a higher priority action taking precedence over the process's continuation. A process would therefore be *forced* to compensate immediately, in a similar manner to which a computer's CPU can have its execution interrupted by a peripheral. This type of behaviour is not representable in timed process calculus since clocks are *lower* not higher priority actions, and all other actions are merely interleaved, which could potentially leave an interruption to very last, which is hardly ideal.

Since this seems a fundamental problem of the calculus, I began by creating a new, experimental process calculus called *ICCS* based on the work of Lüttgen (1998). A summary of this calculus can be found in Appendix A, Section A.1. ICCS (Interruptible CCS) is an extension of CCS which adds interruption actions like $\langle a \rangle$ with higher priority than other actions. Interruption is also *localised* so an interruption action can only pre-empt peer actions in much the same way that clocks can only be pre-empted by peer silent actions. Nevertheless, ICCS was rejected because it is ad-hoc and the equivalence theory doesn't work well. I also decided that my fundamental judgment about what interruption is was wrong. Since an orchestration engine is dealing with distributed *Web services* it simply does not have the control like that of a CPU over its hardware. It is impossible to force a Web service to stop immediately, unless such behaviour has been explicitly implemented. Therefore, I concluded that the only feasible way to represent localised interruption was not through an extension, but using abstract time itself.

This is the main reason why in Chapter 5 I added *yield* actions to Cashew-A. A yield action broadly maps onto waiting for a clock to tick. Specifically, a yield can only allow an interruptible workflow to continue if there exists no way that an interruption can be raised by a sequence of silent actions. This effectively synchronises the workflow before compensation is necessary, as this is the only way to ensure that all components are listening and will react when required. They cannot be interrupted at will. Therefore, in this respect CaSE provides an adequate basis for modelling interruption. Nevertheless there are still further issues to deal with.

### 6.2.2 Generalisation

Further extensions were required as experimentation showed up CaSE's inflexibility in a number of other areas. There are two main issues I wish to resolve. The first concerns the structure of workflow semantics.

**Phasing** is an important technique I will be using for component modelling. It involves splitting a component's behaviour into a number of phases or *intervals*, separated

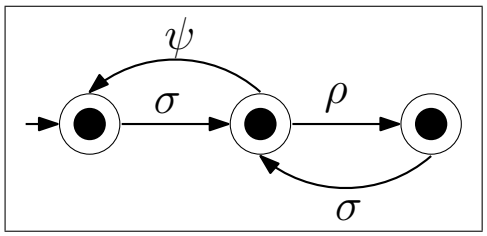



Figure 6.4: A simple phase transition system

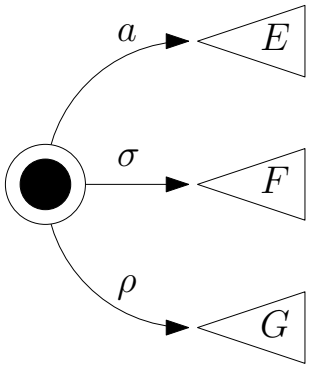



Figure 6.5: A Patient Clock Choice LTS

by clock ticks. Each phase represents a distinct work unit where the component is engaging in a particular activity, for example trying to satisfy its preconditions or executing. Once the activity has completed, that is when all internal activity ceases, the next phase clock ticks to begin the successor phase. Thus, each component will have an associated *phase transition system*, which sub-components will be able to hook into to guide their execution (see Chapter 7, Section 7.2.1). For instance, Figure 6.4 illustrates a simple phase transition system, which ought to be representable.

This particular requirement motivates a re-evaluation of clocks in CaSE, which has a degree of inflexibility with respect to modelling different types of timed transition systems. Consider, for instance, the simple transition system in Figure 6.5. Here is a transition system which makes a choice between doing a non-clock action $a$ to become process $E$, a tick on $\sigma$ to become $F$, or a $\rho$ tick to become $G$. Meanwhile the process is fully patient on all other clocks (represented by the bulls-eye state), so that it will not restrict any clock context; that is, other as yet unknown clocks can tick at will. Such processes are foundational to a workflow's phase transition system.

CaSE cannot represent a construction like this, where a choice is made by one of several clocks, whilst observing total patience. In order to represent patient clock prefix several versions of CaSE have the *stable* timeout operator $\lceil E \rceil \sigma(F)$, a partially static operator which will reduce only if a non-clock action happens in $E$, or if the given timeout clock ticks (see Chapter 2, Section 2.5.3 for the semantics). Patient clock prefix ($\sigma.E$) can be represented as $\lceil \mathbf{0} \rceil \sigma(F)$, where $\mathbf{0}$ can be replaced by any patient process (e.g. $a.E$), but this does not generalise up to multiple clock choice. The only way to represent this is to

nest stable timeouts like so: $\lceil\lceil a.E\rceil\sigma(F)\rceil\rho(G)$. However this isn't correct, because whilst $a$ will indeed lead to $E$ and $\rho$ to $G$, $\sigma$ leads to $\lceil F\rceil\rho(G)$ and not simply to $F$ as it should. The reason is that the outermost timeout is not removed when $\sigma$ ticks, and will remain in place until $F$ performs a non-clock action. The $\rho$ path is thus still available which may lead to unexpected behaviour if a pure choice is desired.

A possible solution is to define stable timeout over a set of clocks and resultant processes, e.g. $\lceil\mathcal{E}\rceil\tilde{\sigma}(\tilde{\mathcal{E}})$, but this seems ad-hoc. Another option is to use CaSE's fragile timeout, and represent the clock context explicitly, as suggested by Lüttgen as a way of rewriting the stable timeout operator. For instance $\lceil E\rceil\sigma(F) \triangleq \mu X.\lfloor E + \rho_1 \cdots \rho_n\rfloor\sigma(F)$, i.e. explicit creation of self-transitions for all clocks other than $\sigma$, provides a possible way of allowing all other clocks to effectively idle by making them recurse. However, this won't suffice either because it is impossible to build an arbitrary clock context for $\mathscr{T}$ (the set of all clocks) because the set will expand over time as a process grows. To overcome this problem, my new calculus will be built specifically to allow such constructions directly. This is important, as a hierarchical system of components should collapse to a process identical to a simple component, not restricting clocks.

The second issue is rather more nuanced, the need to represent a **choice** operator across parallel composition. This may seem an odd desirable as CCS already has a choice operator. However, this is a *primitive* choice operator, not one which allows the composition of parallel processes generally. My process architecture, in line with previous work (Norton and Fairtlough, 2004) is *reactive* in nature – the processes can be non-terminating (i.e. process graphs are strongly connected), and it should be possible to execute any workflow process *ad infinitum*, clearly this is the case since we include an iteration operator. Nevertheless, we need to avoid representing this iteration by placing parallel processes within the fixpoint operator, lest we run the risk of becoming undecidable. Hence, since all workflows will be described by parallel composition of primitive reactive agents, it becomes necessary to investigate how exclusive choice may be described.

These two issues led me to construct a second calculus which I call CaSE$^{\text{mt}}$ after Moller and Tofts (1990), which I describe in Appendix A, Section A.2. CaSE$^{\text{mt}}$ is a generalisation of CaSE which replaces the timeout operator with a clock prefix operator and additionally adds a *negative clock prefix* operator $\neg\sigma.E$ which advances only when a clock other than $\sigma$ can tick. Additionally it has *two* variants of the choice operator, one of which allows a clock to resolve it, in contrast to CaSE. These two changes allows the redefinition of stable timeout in terms of fragile timeout as required, specifically $\lceil E\rceil\sigma(F) \triangleq \mu X.\lfloor E + \neg\sigma.X\rfloor\sigma.X(F)$. This in turn allows the representation of clock choice as required.

However, this calculus does have a rather unfortunate problem – because it is so low-level it leads to extremely complicated processes for even simple choices. For instance the choice in Figure 6.4 is represented as

$$\mu X.(\sigma.\mu Y.(\psi.X + \rho.\mu Z.(\sigma.Y + (\neg\rho.Z + \neg\psi.\mathbf{0})) + \neg\sigma.Y) + (\neg\rho.X + \neg\psi.\mathbf{0}))$$

Clearly this is a very complicated process for what is really quite a simple transition system. Bearing this in mind I set about creating the third and final process calculus, CaSE$^{ip}$. As we shall see, the main contrast with CaSE (and indeed CaSE$^{mt}$) is the way that the semantics are described. Nevertheless, the discussion of CaSE$^{mt}$ is interesting because it illustrates how CaSE eventually became CaSE$^{ip}$.

## 6.3 Introduction to CaSE$^{ip}$

CaSE$^{ip}$ is an abstract timed process calculus based on CCS. It can be seen as an extension of CaSE in that it retains many of its concepts, but generalises them and introduces a new style of operational semantics. The main difference is that CaSE$^{ip}$ makes patience *implicit* (hence CaSE with *implicit patience*). In CaSE if a process is patient on a particular clock, it has a self-transition. This can be seen in the semantics in Chapter 2 Section 2.5.3, where both processes **0** and $a.E$ have a $\sigma$ self-transition on any clock. A clock transition is only absent from a process when that process explicitly holds that clock up. For instance $\Delta$ has no clock transitions at all, and $\Delta_\sigma$ has no $\sigma$ transition (but has self-transitions on all other clocks). Such processes likewise prevent a $\sigma$ transition in processes with which they are composed. This is one of the main reasons why having a clock decide a choice in a CaSE-style calculus is cumbersome, because it is impossible to distinguish a process which simply *allows* a tick from one which *causes* a tick.

The new calculus will, therefore, explicitly differentiate between a clock tick arising from a process simply allowing it, but not actually reacting to it, and one which does react and changes as a result. This seems sensible – if a clock tick does not cause any state change in a process surely it is a wasted transition to include it at all. So, as in PMC (Andersen and Mendler, 1994), clock transitions are enabled only via *inclusion* – every clock is not enabled by default but only if it is explicitly so. But in contrast to PMC, clocks which have no transitions are not necessarily stalled, because it is convenient to retain CaSE's property that processes are patient by default. Instead, there are now *three* states a clock can be in, instead of just two: **active**, **patient** and **stalled**. A process $P$ will only prevent another composed process $Q$ from producing a tick on a particular clock if that clock is explicitly stalled by inclusion of a $\Delta$ or $\Delta_\sigma$ in $P$.

One advantage of such an approach is that dealing with choices made by clocks is far more elegant. For instance, and in contrast to both CaSE and CaSE$^{mt}$, this will allow the following process to have the obvious meaning:

$$a.E + \sigma.F + \rho.G$$

This means if $a$ occurs then choose $E$, if $\sigma$ then $F$, or if $\rho$ then $G$. Furthermore, this process will idle over all other clocks automatically, with only $\sigma$ and $\rho$ active, but no clocks stalled. A process may only decide a choice with a clock tick if the process in question actively engages that clock, rather than simply allows it. A process which

simply idles over a clock tick will not resolve a choice, as it is not an active participant. On the other hand a clock *will* decide the choice if the process actively engages it. If both sides of a choice react to a clock tick then both advance in parallel, as before. I also add the requirement that for a tick on one side to decide the choice, the other side must not explicitly prevent the clock from ticking.

For instance, in $E + F$ the choice can only be decided by $E \xrightarrow{\sigma} E'$ provided $F$ does not stall $\sigma$. The overall effect of this is that the complexity of deciding clock choices is dealt with in the transition system semantics. Thus the process syntax is free from the need to explicitly customise patience.

## 6.4 Clock Renaming

In addition to having a new form of patience, CaSE$^{\text{ip}}$ also has a new operator, called *clock renaming*. The clock renaming operator in CaSE$^{\text{ip}}$ allows clock ticks to be renamed to visible actions (where hiding "renames" them to silent actions) . Although the semantics for renaming has not changed, I have added an additional clause to the syntax, i.e. $E\{\!\![\sigma \mapsto a]\!\!\}$. Renaming clocks to actions is the *only* possible renaming which can be performed on clocks that does not break clock determinism. Renaming an action to a clock would, because actions can be non-deterministic – e.g. $(a.P + a.Q)\{\!\![a \mapsto \sigma]\!\!\}$ would lead to two possible $\sigma$ transitions. Likewise, renaming clocks to other clocks can break determinism, for instance in the process $(\sigma.P + \rho.Q)\{\!\![\sigma \mapsto \rho]\!\!\}$. But renaming clocks to actions does not because, although actions can be non-deterministic, there can be only one proxy action for the renamed clock and therefore the clock can only advance through that action instance.

Clock renaming enables several possibilities: firstly, the problem with clock hiding imposing an implicit total order is solved. Instead of hiding a collection of clocks in sequence, which places an order on them, e.g. $E/\rho/\sigma$ where $\rho$ takes precedence over $\sigma$, multiple clocks can instead be renamed to a given action. This action can then be synchronised with a co-action, which will produce a $\tau$ for each clock. For instance, we may write $(E\{\!\![\sigma \mapsto a]\!\!\}\{\!\![\rho \mapsto a]\!\!\} \mid \mu X.\overline{a}.X) \setminus a$, which is effectively clock hiding but without the order. Naturally this relies on $a$ not being used by $E$.

Secondly, it allows one-to-many synchronisations in a similar style to the *Calculus of Broadcasting Systems* (Prasad, 1995), where a single output synchronises with many inputs. This in turn, along with the other changes made in CaSE$^{\text{ip}}$, allows the representation of a static compositional choice operator. Such choices are needed to represent decisions made on the basis of a message's receipt, which is needed for the Cashew-A choice operator (see Chapter 5). For example:

$$\left(\sigma^a.\overline{z}.\rho.E + \rho.\mathbf{0} \;\middle|\; \sigma^b.\overline{z}.\rho.F + \rho.\mathbf{0} \;\middle|\; \sigma^c.\overline{z}.\rho.G + \rho.\mathbf{0} \;\middle|\; \underline{z}_{\rho}.\mathbf{0}\right) /\rho \; \{\!\![\sigma^n \mapsto n \,|\, n \in \{a,b,c\}]\!\!\} \setminus z$$

This process makes a choice based on the receipt of messages $a, b$ or $c$. The three messages are represented by three clock proxies $\sigma^a, \sigma^b$ and $\sigma^c$, respectively, which are renamed using the new operator to their respective messages. In addition a further clock, $\rho$, is used to cancel out the other choices once any one of the message clocks has ticked. This clock, $\rho$, is hidden first, so that it has highest priority. The other three clocks are renamed and so have equal priority. Each message has an associated agent which is responsible for receiving the message and then cancelling the other two receivers. A fourth agent holds up the $\rho$ clock until it has received a $z$ communication (indicated by the underlining of $z$), which can be sent out by one of the three receivers. Therefore, when an output message from the environment synchronises with one of the three inputs, the other inputs are immediately disallowed because of the presence of $\rho$ which pre-empts them away, meaning only one of the three processes $E, F$ and $G$ may be activated. This construct can be extended to any number of choice processes. Such a choice semantics is compositional because an additional message proxied by $\sigma^d$, for instance, can easily be included by adding another agent and clock renamer.

Naturally this simple example could be represented simply by $a.E+b.F+c.G$ in regular CCS. However, if we wish each message receipt to initiate execution of a collection of recursive parallel composed agents this solution no longer becomes viable. In particular it goes against the philosophy of a process being a composition of sequential agents and increases the possibility of processes becoming unverifiable. Furthermore, since we are using clocks it opens up the possibility for other processes to pre-empt away the choice branches if necessary. I therefore content that this representation of compositional choice is a much more flexible solution.

## 6.5 Syntax and Operational Semantics

CaSE$^{\text{ip}}$ uses a mixture of the syntax from CaSE$^{\text{mt}}$ and CaSE. I favour a clock prefix operator over a binary timeout but only admit one form of summation. I assume the existence of two infinite sets $\mathscr{A}$ and $\mathscr{T}$, the set of all actions and clocks. In particular, it should be noted that $\mathscr{T} \setminus T = \mathscr{T}$, for any finite set of clocks $T$ – $\mathscr{T}$ is the top set of all clocks.

In line with the *Unified Semantic Framework*, a framework designed to help comparison of differing abstract timed process calculi (Lüttgen and Mendler, 2005), I define the three sets $\Sigma_{\mathcal{E}}$, $\mathscr{A}_{\mathcal{E}}$ and $\mathcal{T}_{\mathcal{E}}$ – the instability, initial actions and initial clocks sets. The instability set and initial clock set are used to define the clocks stalled by a process, and the clocks which have an outgoing transition, respectively. A clock found in neither set is deemed patient. These three sets ensure that the semantics are well-defined, especially in the case of testing whether a process is incapable of performing a particular action. Defining this in terms of the semantic transition relation itself introduces undecidability problems because of the nature of recursion, and thus I use finite sets to ground the se-

mantics. These three sets allow the formal definition of the concepts **active**, **patient** and **stalled** outlined above.

**Definition 6.5.1** ***Active Clocks***
*A clock $\sigma$ is said to be **active** in process $E$ provided that $\sigma \in \mathcal{T}_E$.*

**Definition 6.5.2** ***Stalled Clocks***
*A clock $\sigma$ is said to be **stalled** in process $E$ provided that $\sigma \in \Sigma_E$.*

**Definition 6.5.3** ***Patient Clocks***
*A clock $\sigma$ is said to be **patient** in process $E$ provided that $\sigma \notin \mathcal{T}_E$ and $\sigma \notin \Sigma_E$.*

We can also define the notion of when a process is patient, along with the well known CCS notion of *stability* – when a process cannot perform any silent actions.

**Definition 6.5.4** ***Patient Processes***
*A process $E$ is said to be **patient** when there exists at least one clock patient in $E$. That is, $\Sigma_E \subset \mathscr{T}$.*

If a process is patient it means that there is an infinite number of clocks which are not stalled by $E$, since $\mathscr{T}$ is a theoretically infinite set.

**Definition 6.5.5** ***Stability***
*A process $E$ is said to be **stable** when it cannot silently move into another state, that is $\tau \notin \mathscr{A}_E$.*

Stable processes take on an important role in CaSE$^{\text{ip}}$ and timed process calculi in general. Since no clock can tick in an unstable process because of maximal progress , it is necessary to wait for stability before seeing which clocks can tick.

The syntax of CaSE$^{\text{ip}}$ is found in Table 6.1. It uses a similar syntax to CaSE (the sorts are identical), but with a clock prefix operator $\sigma.E$. Additionally, I provide the *clock renaming* operator $E\{\!|\sigma \mapsto a|\!\}$, described in Section 6.4. Since I will be giving a more formal examination of this new calculus I also define a *restricted* form of syntax for CaSE$^{\text{ip}}$ in Table 6.2. The latter syntax crystalises the assumption that I made at the beginning of this Chapter, that a process is a composition of sequential agents. Thus, the static operators of this language may only compose closed sequential agents, defined by the inductively defined set $\mathcal{E}\langle\tilde{X}\rangle$, where $X$ is the set of free process variables in the given term. Parallel processes $\mathcal{P}$ are then defined using BNF to encapsulate closed agents and the static operators.

Our CaSE$^{\text{ip}}$ semantics, based on the Labelled Transition System $(\mathcal{P}, \mathscr{A} \cup \mathscr{T}, \rightarrow)$, is given in Table 6.3, along with the three set definitions. Perhaps of most significance in these semantics is that all idling rules have been removed, namely tIdle, tPatient and tStall, with added self transitions for idle processes. This is because patient clock ticks are indicated by two side conditions $(a)$ and $(b)$. They are used by the summation and

$$\Lambda = \{a, b, c \cdots\} \quad \overline{\Lambda} = \{\overline{a} | a \in \Lambda\} \quad \mathscr{A} = \Lambda \cup \overline{\Lambda} \cup \{\tau\} \quad \mathscr{T} = \{\sigma, \rho \cdots\}$$
$$\alpha, \beta \in \mathscr{A} \quad \gamma, \delta \in \mathscr{A} \cup \mathscr{T} \quad E, F, G \cdots \in \mathcal{E}$$

$$\mathcal{E} ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \sigma.\mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E}|\mathcal{E} \mid \mathcal{E} \setminus a \mid \mathcal{E}\{\!\{a \mapsto a\}\!\} \mid \mathcal{E}\{\!\{\sigma \mapsto a\}\!\} \mid \mathcal{E}/\sigma \mid \mu X.\mathcal{E} \mid X$$

Table 6.1: Full Syntax of CaSE with implicit patience

$$\frac{-}{\mathbf{0} \in \mathcal{E}\langle\emptyset\rangle} \qquad \frac{-}{\Delta \in \mathcal{E}\langle\emptyset\rangle} \qquad \frac{-}{\Delta_\sigma \in \mathcal{E}\langle\emptyset\rangle}$$

$$\frac{E \in \mathcal{E}\langle\tilde{X}\rangle}{\gamma.E \in \mathcal{E}\langle\tilde{X}\rangle} \qquad \frac{-}{X \in \mathcal{E}\langle\{X\}\rangle} \qquad \frac{E \in \mathcal{E}\langle\tilde{X}\rangle}{\mu X.E \in \mathcal{E}\langle\tilde{X} \setminus \{X\}\rangle}$$

$$\frac{E \in \mathcal{E}\langle\tilde{X}\rangle \quad F \in \mathcal{E}\langle\tilde{Y}\rangle}{E + F \in \mathcal{E}\langle\tilde{X} \cup \tilde{Y}\rangle}$$

$$\mathcal{P} ::= \mathcal{E}\langle\emptyset\rangle \mid \mathcal{P}|\mathcal{P} \mid \mathcal{P} \setminus a \mid \mathcal{P}\{\!\{a \mapsto a\}\!\} \mid \mathcal{P}\{\!\{\sigma \mapsto a\}\!\} \mid \mathcal{P}/\sigma$$

Table 6.2: Restricted Syntax of CaSE with implicit patience

composition rules (tSum2, tSum3, tCom2, tCom3) to permit only one side of the operator to tick if the other side is patient over the particular clock. Patience is indicated by the absence of a clock from both instability and initial clock sets and the side-conditions $(a)$ and $(b)$ enforce this. If both sides tick over the same clock they are composed in the usual way.

An interesting point of note is that, unlike CaSE$^{\text{mt}}$, it is impossible to represent the timeout operator $\lfloor E \rfloor \sigma(F)$ exactly in CaSE$^{\text{ip}}$. The reason for this is there is no way of pruning a clock tick from a sub-expression, as timeout does with the process on the LHS ($E$). Specifically, placing a process on the LHS of a timeout means that any clock transitions on the timeout clock are removed from the process semantically. The only way to prevent a clock from ticking in CaSE$^{\text{ip}}$ is to sum the process with $\Delta_\sigma$, but this prevents the clock ticking altogether, not just in the sub-expression. However, since an exact replication of CaSE's timeout operator is not required for the service composition semantics, this is a satisfactory compromise, and timeout will be broadly derived as $\lfloor E \rfloor \sigma(F) \triangleq E + \sigma.F$.

### 6.5.1 Relationship between transitions and sets

It is neccessary before I can prove anything about my calculus that I firmly establish the relationship between the transition relation and the various sets. The casual reader can omit this section, as it simply ensures that our intuition of what the various sets mean actually reflects reality. The main Lemma to be proved in this Section is as follows:

Table 6.3: CaSE$^{\text{ip}}$ Operational Semantics

$$\textsf{Act}\ \frac{-}{\gamma.E \xrightarrow{\gamma} E} \qquad \textsf{tSum1}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'} \qquad \textsf{tCom1}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E \mid F \xrightarrow{\sigma} E' \mid F'}\,(d)$$

$$\textsf{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \textsf{tSum2}\ \frac{E \xrightarrow{\sigma} E'}{E + F \xrightarrow{\sigma} E'}\,(a) \qquad \textsf{tCom2}\ \frac{E \xrightarrow{\sigma} E'}{E \mid F \xrightarrow{\sigma} E' \mid F}\,(a,d)$$

$$\textsf{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'} \qquad \textsf{tSum3}\ \frac{F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} F'}\,(b) \qquad \textsf{tCom3}\ \frac{F \xrightarrow{\sigma} F'}{E \mid F \xrightarrow{\sigma} E \mid F'}\,(b,d)$$

$$\textsf{Com1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \textsf{Com2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \textsf{Com3}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\textsf{Res}\ \frac{E \xrightarrow{\gamma} E'}{E \setminus a \xrightarrow{\gamma} E' \setminus a}\,(2) \qquad \textsf{Rel}\ \frac{E \xrightarrow{\gamma} E'}{E\{\!|f|\!\} \xrightarrow{f(\gamma)} E'\{\!|f|\!\}} \qquad \textsf{Rec}\ \frac{E\{\mu X.E/X\} \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'}$$

$$\textsf{Hid}\ \frac{E \xrightarrow{\alpha} E'}{E/\sigma \xrightarrow{\alpha} E'/\sigma} \qquad \textsf{tHid1}\ \frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma} \qquad \textsf{tHid2}\ \frac{E \xrightarrow{\rho} E'}{E/\sigma \xrightarrow{\rho} E'/\sigma}\,(1,c)$$

1) $\rho \neq \sigma$ 2) $\gamma \notin \{a, \overline{a}\}$ *a)* $\sigma \notin \Sigma_F \cup \mathcal{T}_F$ *b)* $\sigma \notin \Sigma_E \cup \mathcal{T}_E$ *c)* $\sigma \notin \mathcal{T}_E$ *d)* $\tau \notin \mathscr{A}_{E \mid F}$

| **Instability set** | | | **Initial Clock set** | | |
|---|---|---|---|---|---|
| $\Sigma_{E+F}$ | $=$ | $\Sigma_E \cup \Sigma_F$ | | | |
| $\Sigma_{\sigma.E}$ | $=$ | $\emptyset$ | | | |
| $\Sigma_{\mathbf{0}}$ | $=$ | $\emptyset$ | $\mathcal{T}_{E+F}$ | $=$ | $(\mathcal{T}_E \cup \mathcal{T}_F) \setminus \Sigma_{E+F}$ |
| $\Sigma_{\Delta}$ | $=$ | $\mathscr{T}$ | $\mathcal{T}_{\sigma.E}$ | $=$ | $\{\sigma\}$ |
| $\Sigma_{\Delta_\sigma}$ | $=$ | $\{\sigma\}$ | $\mathcal{T}_{\mathbf{0}}$ | $=$ | $\emptyset$ |
| $\Sigma_{a.E}$ | $=$ | $\emptyset$ | $\mathcal{T}_{\Delta}$ | $=$ | $\emptyset$ |
| $\Sigma_{\tau.E}$ | $=$ | $\mathscr{T}$ | $\mathcal{T}_{\Delta_\sigma}$ | $=$ | $\emptyset$ |
| $\Sigma_{E \mid F}$ | $=$ | $\begin{cases} \mathscr{T} & \text{if } \exists a.a \in \mathscr{A}_E \wedge \overline{a} \in \mathscr{A}_F \\ \Sigma_E \cup \Sigma_F & \text{otherwise} \end{cases}$ | $\mathcal{T}_{\alpha.E}$ | $=$ | $\emptyset$ |
| | | | $\mathcal{T}_{E \mid F}$ | $=$ | $(\mathcal{T}_E \cup \mathcal{T}_F) \setminus \Sigma_{E \mid F}$ |
| | | | $\mathcal{T}_{E \setminus a}$ | $=$ | $\mathcal{T}_E$ |
| | | | $\mathcal{T}_{E\{\!|a \mapsto b|\!\}}$ | $=$ | $\mathcal{T}_E$ |
| $\Sigma_{E \setminus a}$ | $=$ | $\Sigma_E$ | $\mathcal{T}_{E\{\!|\sigma \mapsto a|\!\}}$ | $=$ | $\mathcal{T}_E \setminus \{\sigma\}$ |
| $\Sigma_{E\{\!|a \mapsto b|\!\}}$ | $=$ | $\Sigma_E$ | $\mathcal{T}_{E/\sigma}$ | $=$ | $\begin{cases} \emptyset & \text{if } \sigma \in \mathcal{T}_E \\ \mathcal{T}_E & \text{otherwise} \end{cases}$ |
| $\Sigma_{E\{\!|\sigma \mapsto a|\!\}}$ | $=$ | $\Sigma_E \setminus \{\sigma\}$ | | | |
| $\Sigma_{E/\sigma}$ | $=$ | $\begin{cases} \mathscr{T} & \text{if } \sigma \in \mathcal{T}_E \\ \Sigma_E \setminus \{\sigma\} & \text{otherwise} \end{cases}$ | $\mathcal{T}_{\mu X.E}$ | $=$ | $\mathcal{T}_E$ |
| | | | $\mathcal{T}_X$ | $=$ | $\emptyset$ |
| $\Sigma_{\mu X.E}$ | $=$ | $\Sigma_E$ | | | |
| $\Sigma_X$ | $=$ | $\emptyset$ | | | |

| **Initial Action set** | | | | | |
|---|---|---|---|---|---|
| $\mathscr{A}_{\sigma.E}$ | $=$ | $\emptyset$ | $\mathscr{A}_{E \setminus a}$ | $=$ | $\mathscr{A}_E \setminus \{a, \overline{a}\}$ |
| $\mathscr{A}_{\mathbf{0}}$ | $=$ | $\emptyset$ | $\mathscr{A}_{E\{\!|f|\!\}}$ | $=$ | $\{f(\gamma) \mid \gamma \in \mathscr{A}_E \cup \mathcal{T}_E\}$ |
| $\mathscr{A}_{\Delta}$ | $=$ | $\emptyset$ | $\mathscr{A}_{E/\sigma}$ | $=$ | $\mathscr{A}_E \cup \{\tau \mid \sigma \in \mathcal{T}_E\}$ |
| $\mathscr{A}_{\Delta_\sigma}$ | $=$ | $\emptyset$ | $\mathscr{A}_{\mu X.E}$ | $=$ | $\mathscr{A}_E$ |
| $\mathscr{A}_{\alpha.E}$ | $=$ | $\{\alpha\}$ | $\mathscr{A}_X$ | $=$ | $\emptyset$ |
| $\mathscr{A}_{E+F}$ | $=$ | $\mathscr{A}_E \cup \mathscr{A}_F$ | | | |
| $\mathscr{A}_{E \mid F}$ | $=$ | $\{\tau \mid a \in \mathscr{A}_E \wedge \overline{a} \in \mathscr{A}_F\} \cup \mathscr{A}_E \cup \mathscr{A}_F$ | | | |

**Lemma 6.5.6** *$\gamma \in \mathscr{A}_E \cup \mathcal{T}_E$ if and only if $\exists E'.E \xrightarrow{\gamma} E'$*

Specifically, the presence of an action or clock in the respective sets is equivalent to there being a transition. To do this, I require a notion of *subexpressions* and *subprocesses* which helps link syntax to semantics (and will be particularly important in proofs for recursion).

**Definition 6.5.7** ***Subexpressions***
*An expression is a subexpression of another (sequential) expression if it appears directly under any operator other than prefix. The definition is closed under the following rules:*

- *$E$ is a subexpression of $E$;*
- *If $E$ is a subexpression of $F$ then $E$ is a subexpression of $F + G$;*
- *If $E$ is a subexpression of $G$ then $E$ is a subexpression of $F + G$;*
- *If $E$ is a subexpression of $F$ then $E$ is a subexpression of $\mu X.F$.*

**Definition 6.5.8** ***Subprocesses***
*A process is a subprocess of another process if it appears directly under any operator other than prefix. The definition is closed under the following rules:*

- *$P$ is a subprocess of $P$;*
- *If $E$ is a subexpression of $F$ then $E$ is a subprocess of $F$;*
- *If $P$ is a subprocess of $Q$ then $P$ is a subprocess of $Q \mid R$;*
- *If $P$ is a subprocess of $R$ then $P$ is a subprocess of $Q \mid R$;*
- *If $P$ is a subprocess of $Q$ then $P$ is a subprocess of $Q \setminus a$;*
- *If $P$ is a subprocess of $Q$ then $P$ is a subprocess of $Q/a$;*
- *If $P$ is a subprocess of $Q$ then $P$ is a subprocess of $Q\{\!|\gamma \mapsto a|\!\}$.*

I also need the following two propositions, which prove the relationship between the various sets. Specifically, that the presence of an initial $\tau$ means all clocks are stalled, and that an active clock cannot also be stalled.

**Proposition 6.5.9** *For any process $E$, if $\tau \in \mathscr{A}_E$ then $\Sigma_E = \mathscr{T}$.*

**Proof.** By induction on the structure of process $E$. If $\mathscr{A}_E$ contains a $\tau$ action then there are three ways by which that $\tau$ could have been placed in the set:

1. $\tau.F$ is a subprocess of $E$ for some $F$;

2. $F/\sigma$ is a subprocess of $E$ for some $F$ and $\sigma \in \mathcal{T}_F$;

3. $F \mid G$ is a subprocess of $E$ for some $F$ and $G$, with $a \in \mathscr{A}_F$ and $\overline{a} \in \mathscr{A}_G$.

At least one of these must be true, as the other rules for defining $\mathscr{A}_E$ compose the sets for constituent subprocesses. For case (1), $\Sigma_{\tau.F}$ is clearly $\mathscr{T}$. For case (2), since we know that $\sigma \in \mathcal{T}_F$ then again $\Sigma_{F/\sigma} = \mathscr{T}$. For case (3) we know that $a \in \mathscr{A}_F$ and $\overline{a} \in \mathscr{A}_G$ and therefore $\Sigma_{F \mid G} = \mathscr{T}$. Finally we note that if $\Sigma_F = \mathscr{T}$ for some subprocess $F$ of $E$ it must also follow that $\Sigma_E = \mathscr{T}$, since the definition of the latter composes the former and $\mathscr{T}$ cannot be reduced. □

**Proposition 6.5.10** *For any process $E$ and clock $\sigma$, $\sigma \in \mathcal{T}_E$ implies $\sigma \notin \Sigma_E$.*

**Proof.** By inspection of the definition of $\mathcal{T}_E$ and $\Sigma_E$ in Table 6.3. There are three classes of processes to consider:

- The non-composite processes for which $\sigma \in \Sigma_E$ are $E \equiv \Delta$, $\Delta_\sigma$ and $\tau.G$ (for any $G$). For all of these $\mathcal{T}_E = \emptyset$ and hence $\sigma \notin \mathcal{T}_E$ as required.
- The three composite processes which can remove $\sigma$ from $\mathcal{T}_E$ are $E \equiv F \mid G$, $F + G$ and $F/\sigma$. The definition of $\mathcal{T}_E$ for the first two explicitly removes $\Sigma_E$ and therefore by definition $\sigma \notin \mathcal{T}_E$. The third process has the same condition for setting $\Sigma_E = \mathscr{T}$ as setting $\mathcal{T}_E = \emptyset$.
- The remaining composite processes, such as $\mu X.F$, simply copy $\mathcal{T}_E$ and $\Sigma_E$ for the enclosed process and hence do not alter the definition.

Thus in all cases the statement follows. □

Finally, to prove Lemma 6.5.6 I prove three propositions. The first two prove the implication in one direction, and the third proves it in the other.

**Proposition 6.5.11** *For any process $E$, if $\alpha \in \mathscr{A}_E$ then $\exists E'.E \xrightarrow{\alpha} E'$*

**Proof.** By induction on the process structure of $E$. Specifically, I use the definitions found in Table 6.3 to prove for each CaSE$^{\text{ip}}$ construct $f$ and process sort $\tilde{E}$ that whenever $\alpha \in \mathscr{A}_{f(\tilde{E})}$ it necessarily follows that $\exists E'.f(\tilde{E}) \xrightarrow{\alpha} E'$, on the basis that this is true for the parts in $\tilde{E}$. The full proof can be found in Appendix B, Section B.1. □

**Proposition 6.5.12** *For any process $E$, if $\sigma \in \mathcal{T}_E$ then $\exists E'.E \xrightarrow{\sigma} E'$.*

**Proof.** By induction on the structure of process $E$. See Appendix B, Section B.2. □

**Proposition 6.5.13** *For any process $E$, if $\exists E'.E \xrightarrow{\gamma} E'$ then $\gamma \in \mathscr{A}_E \cup \mathcal{T}_E$.*

**Proof.** By induction on the structure of process $E$. See Appendix B, Section B.3. □

This completes the proof of Lemma 6.5.6, showing that we can treat transition presence and membership of the initial action or clock sets as equivalent.

### 6.5.2 Free variables and Substitution

In this Section I prove some important Lemmas about process expressions with free variables (called *open terms*). Since my calculus has recursive terms with $\mu$ fixpoints over otherwise free variables and substitution, there are certain important properties of substitution which we need to establish for future proofs about the equivalence theory. Once again, the casual reader can safely skip this section.

For the purpose of this thesis I leave the definition of substitution undefined, because this will usually be given by the implementation language (in particular I directly use Haskell's substitution mechanism in Chapter 8). Instead of a formal definition I simply state that $E\{F/X\}$ is the expression $E$ with every free instance of $X$ replaced by $F$, whilst taking account of the free variables in $F$. I assume that any such valid definition of substitution avoids capturing $F$'s free variables, possibly through use of de-Bruijn indices (de Bruijn, 1972). In particular the substitution of the expression $(\mu Y.X) + Y$ into itself as $X$ must distinguish the free $Y$ from the bound $Y$.

I first make the following definition of free variables in sequential terms (since non-sequential terms are always closed):

**Definition 6.5.14** ***Free variables***
*The free variables* $\mathsf{fv}(E)$ *of an expression* $E$ *are the process variables not bound by a* $\mu$*. They are so defined:*

- $\mathsf{fv}(\boldsymbol{0}) \triangleq \emptyset$
- $\mathsf{fv}(X) \triangleq \{X\}$
- $\mathsf{fv}(E + F) \triangleq \mathsf{fv}(E) \cup \mathsf{fv}(F)$
- $\mathsf{fv}(\mu X.E) \triangleq \mathsf{fv}(E) \setminus \{X\}$

In some circumstances we need to alter the order of substitutions. The following Lemma shows when this is possible.

**Lemma 6.5.15** ***Commutativity of nested substitutions***
*If* $Y \notin \mathsf{fv}(E)$ *then* $G\{\mu X.E/X\}\{\mu Y.F\{\mu X.E/X\}/Y\} \equiv G\{\mu Y.F/Y\}\{\mu X.E/X\}$

**Proof.** By induction on the structure of expression $G$.

- $G \equiv \mathbf{0}$. Follows trivially, as no substitution can occur.
- $G \equiv Z$, with $Z \neq X$ and $Z \neq Y$. Also follows trivially.
- $G \equiv X$, with $X \neq Y$. Then since $Y \notin \mathsf{fv}(E)$ and $Y \notin \mathsf{fv}(X)$ it follows that in either case the substitution results in $\mu X.E$.
- $G \equiv Y$, with $Y \neq X$. Then $G\{\mu X.E/X\}\{\mu Y.F\{\mu X.E/X\}/Y\} = \mu Y.F\{\mu X.E/X\}$ since $X \notin \mathsf{fv}(Y)$, and $G\{\mu Y.F/Y\}\{\mu X.E/X\} = \mu Y.F\{\mu X.E/X\}$ as required.

- $G \equiv \mu Z.H$, with $H\{\mu X.E/X\}\{\mu Y.F\{\mu X.E/X\}/Y\} \equiv H\{\mu Y.F/Y\}\{\mu X.E/X\}$. If $Z = X$ or $Z = Y$ then the statement follows as no substitution for $X$ or $Y$ can take place in $H$, as these variables are already bound by $\mu Z$ and thus these apparently identical variables are distinguished. Otherwise the case follows by simple induction.
- $G \equiv H + I$. Follows by simple induction.
- $G \equiv \alpha.H$. Follows by simple induction.

Each case is proven and thus the inductive proof is complete. □

Now I can prove the following important Lemma about recursion derivatives.

**Lemma 6.5.16** *Recursion derivatives of open terms*
*If $\mu X.E \xrightarrow{\gamma} F$ then there exists an $E'$ such that $E \xrightarrow{\gamma} E'$ and $F \equiv E'\{\mu X.E/X\}$.*

**Proof.** Since $E$ can only contain prefixes, sums and recursions it follows that any minimal derivation of $\xrightarrow{\gamma}$ can only have applications of rule Act at its leaves. If $\gamma = \alpha$ then since none of the matching rules (Sum1, Sum2, Rec and Act) have more than one antecedent it follows that there is only one application of Act. However, if $\gamma = \sigma$ then tSum1 requires that both sides have the transition and in that case there will be several applications of Act. Hence, we know that there are $n \geq 1$ subexpressions of $E$ of the form $\gamma.H_i$, for $i = 1, \cdots, n$. It follows that $\exists E'.E \xrightarrow{\gamma} E'$.

If the derivation of $\xrightarrow{\gamma}$ involves applications of Rec, construct a $G$ representing the participative agents such that $G = \sum_{i=1}^{n} \gamma.H_i\{\vec{K_i}/\vec{Y_i}\}$, where each $K_i$ and $Y_i$ represents lists of expressions and variables for each subexpression containing a nested $\mu$, such that $G \xrightarrow{\gamma} E'$.

Now, by rule Rec it follows that $\sum \gamma.H_i\{\mu X.E/X\}\{\vec{K_i}/\vec{Y_i}\} \xrightarrow{\gamma} F$, because $X$ is substituted for *first*, before any of the enclosed bindings are expanded. Since the $X$ substitution occurs first in each subexpression, we know that each expression in each $\vec{K_i}$ may contain $\mu X.E$ substituted for $X$, i.e. each $K \in \vec{K_i}$ can be written as $L\{\mu X.E/X\}$ for some unsubstituted expression $L$. We also know that for any $Y \in \vec{Y_i}$ that $Y \notin \mathsf{fv}(E)$, because substitution of $\mu X.E$ for $X$ is subject to all the variables bound in $E$. That is, since every variable in each $\vec{Y_i}$ is a variable bound by a $\mu$ in $E$, it follows that substitution of $\mu X.E$ into $E$, and hence each of the participative agents, differentiates $E$'s free variables from its bound variables. Therefore by inductive application of Lemma 6.5.15, which tells us that for each $K$ there is a corresponding $L$, the expression can be rewritten as $\sum \gamma.H_i\{\vec{L_i}/\vec{Y_i}\}\{\mu X.E/X\}$, or simply $\left(\sum \gamma.I_i\right)\{\mu X.E/X\}$ with $I_i = H_i\{\vec{L_i}/\vec{Y_i}\}$. Since $E' = \sum I_i$, we have an expression of the required form. □

Another issue which we will encounter is the problem of *unguarded free variables* – variables which occur in an expression unguarded by an action prefix. They are formally defined thus:

**Definition 6.5.17** *Unguarded Free Variables*
*$X$ is an unguarded free variable of expression $E$, written $E \rhd X$, if $X \in \mathsf{fv}(E)$ and $X$ is a subexpression of $E$.*

The presence of such variables needs to be considered carefully when reasoning about processes, as when a substitution occurs the behaviour can be changed in unexpected ways, particularly in the presence of clocks. Hence I now prove the following Lemmas about them.

**Lemma 6.5.18** *If $E \rhd X$ then $\Sigma_{E\{\mu X.H/X\}} = \Sigma_E \cup \Sigma_{\mu X.H}$*

**Proof.** By induction on the structure of expression $E$.

- $E \equiv \mathbf{0}$, $E \equiv \gamma.F$ and $E \equiv Y$ with $Y \neq X$. These cases follow trivially, since then $E \not\rhd X$.
- $E \equiv X$. Then $\Sigma_E = \emptyset$ and hence $\Sigma_{X\{\mu X.H/X\}} = \Sigma_{\mu X.H} \cup \Sigma_E$ as required.
- $E \equiv F + G$. Then either $F \rhd X$, $G \rhd X$ or both, and hence either $\Sigma_{F\{\mu X.H/X\}} = \Sigma_F \cup \Sigma_{\mu X.H}$ or $\Sigma_{G\{\mu X.H/X\}} = \Sigma_G \cup \Sigma_{\mu X.H}$, or both. If $F \not\rhd X$ (and similarly for $G \not\rhd X$) then note that $\Sigma_F = \Sigma_{F\{\mu X.H/X\}}$ since then $X$ can only appear behind a prefix in some subexpression of $F$, and the definition of $\Sigma_F$ ignores guarded expressions. Therefore, in any case $\Sigma_{(F+G)\{\mu X.H/X\}} = \Sigma_E \cup \Sigma_F \cup \Sigma_{\mu X.H} = \Sigma_{E+F} \cup \Sigma_{\mu X.H}$ as required.
- $E \equiv \mu Z.F$. Then $X \neq Z$ and $F \rhd X$. Hence this case follows by simple induction.

Each case is proven and thus the inductive proof is complete. □

**Lemma 6.5.19** *If $E \rhd X$ then $\mathcal{T}_{E\{\mu X.H/X\}} = \mathcal{T}_E \cup \mathcal{T}_{\mu X.H} \setminus (\Sigma_E \cup \Sigma_{\mu X.H})$*

**Proof.** By induction on the structure of expression $E$.

- $E \equiv \mathbf{0}$, $E \equiv \gamma.F$ and $E \equiv Y$ with $Y \neq X$. These cases follow trivially, since then $E \not\rhd X$.
- $E \equiv X$. Then $\mathcal{T}_E = \emptyset$ and hence $\mathcal{T}_{X\{\mu X.H/X\}} = \mathcal{T}_{\mu X.H} \cup \mathcal{T}_E$ as required.
- $E \equiv F + G$. Then either $F \rhd X$, $G \rhd X$ or both, and hence either $\mathcal{T}_{F\{\mu X.H/X\}} = \mathcal{T}_F \cup \mathcal{T}_{\mu X.H} \setminus (\Sigma_F \cup \Sigma_{\mu X.H})$ or $\mathcal{T}_{G\{\mu X.H/X\}} = \mathcal{T}_G \cup \mathcal{T}_{\mu X.H} \setminus (\Sigma_F \cup \Sigma_{\mu X.H})$, or both. As per Lemma 6.5.18 we know that $E \not\rhd X \Longrightarrow \mathcal{T}_E = \mathcal{T}_{E\{\mu X.H/X\}}$ and also $F \not\rhd X \Longrightarrow \mathcal{T}_F = \mathcal{T}_{F\{\mu X.H/X\}}$, since the definition of $\mathcal{T}$ ignores guarded expressions. Therefore, in any case $\mathcal{T}_{(F+G)\{\mu X.H/X\}} = \mathcal{T}_E \cup \mathcal{T}_F \cup \mathcal{T}_{\mu X.H} \setminus (\Sigma_E \cup \Sigma_F \cup \Sigma_{\mu X.H}) = \Sigma_{E+F} \cup \Sigma_{\mu X.H}$ as required.
- $E \equiv \mu Z.F$. Follows by simple induction.

Each case is proven and thus the inductive proof is complete. □

**Lemma 6.5.20** *If $E \not\rhd X$ then $\Sigma_{E\{\mu X.F/X\}} = \Sigma_E$*

**Proof.** This follows from the fact that $\mu X.F$ is guarded in $E\{\mu X.F/X\}$ and since $\Sigma_{\mu X.F}$ is not used in defining $\Sigma_{\gamma.\mu X.F}$ for any $\gamma$, neither is it used in the definition of $\Sigma_{E\{\mu X.F/X\}}$, since $\mu X.F$ is behind such a $\gamma$. Therefore its definition is unaffected by the substitution. □

**Lemma 6.5.21** *If $E \not\rhd X$ then $\mathcal{T}_{E\{\mu X.F/X\}} = \mathcal{T}_E$*

**Proof.** Follows the same argument as Lemma 6.5.20. □

The remaining aim of this section is to prove a Lemma which is the partner of Lemma 6.5.16. Specifically, this Lemma (or Lemmas) will show us that, under certain conditions, a substitution does not affect the transitions an expression can make. Naturally enough there are caveats related to clocks, but these will be considered in due course. It is convenient to do these proofs on vectors of substitutions, rather than singletons, and therefore I give the following definition of substitution vectors:

$$\begin{aligned} E\{H,\vec{H}/X,\vec{X}\} &\triangleq E\{H/X\}\{\vec{H}/\vec{X}\} \\ E\{[]\} &\triangleq E \end{aligned}$$

with the assumption that $\vec{H}$ and $\vec{X}$ have the same length. The two lemmas below prove the substitution property about first actions and then clock ticks.

**Lemma 6.5.22** ***Action Derivatives and Substitution***
*For any expression vector $\vec{H}$ and corresponding variable vector $\vec{Y}$, if $E \xrightarrow{\alpha} E'$ then $E\{\vec{H}/\vec{Y}\} \xrightarrow{\alpha} E'\{\vec{H}/\vec{Y}\}$.*

**Proof.** By induction on the structure of $E$.

- Cases $E \equiv \mathbf{0}$, $E \equiv \Delta$, $E \equiv \Delta_\sigma$, $E \equiv \sigma.F$ and $E \equiv X$ follow trivially as then $\nexists E'.E \xrightarrow{\alpha} E'$.
- $E \equiv \mu X.F$, and for any $\vec{H}$ and $\vec{Y}$, $F \xrightarrow{\alpha} F'$ implies $F\{\vec{H}/\vec{Y}\} \xrightarrow{\alpha} F'\{\vec{H}/\vec{Y}\}$. By Lemma 6.5.16 there is a $F'$ such that $F \xrightarrow{\alpha} F'$ and $E' \equiv F'\{\mu X.F/X\}$. Therefore by induction we also know that $F\{\mu X.F,\vec{I}/X,\vec{Z}\} \xrightarrow{\alpha} F'\{\mu X.F,\vec{I}/X,\vec{Z}\}$, for some vectors $\vec{I}$ and $\vec{Z}$. Furthermore, since $X$ is bound in $\mu X.F$, it follows that $X \notin \mathsf{fv}(I)$ for any $I$ in $\vec{I}$. Then by rule Rec and $E' \equiv F'\{\mu X.F/X\}$ it follows that proving $\mu X.F\{\vec{I}/\vec{Z}\} \xrightarrow{\alpha} E'\{\vec{I}/\vec{Z}\}$ requires proof of $F\{\vec{I}/\vec{Z}\}\{\mu X.F\{\vec{I}/\vec{Z}\}/X\} \xrightarrow{\alpha} F'\{\mu X.F/X\}\{\vec{I}/\vec{Z}\}$. Finally, since $F\{\mu X.F,\vec{I}/X,\vec{Z}\} \xrightarrow{\alpha} F'\{\mu X.F,\vec{I}/X,\vec{Z}\}$, it suffices to show that $F\{\vec{I}/\vec{Z}\}\{\mu X.F\{\vec{I}/\vec{Z}\}/X\} \equiv F\{\mu X.E/X\}\{\vec{I}/\vec{Z}\}$, which follows by inductive application of Lemma 6.5.15.

- $E \equiv \alpha.F$. Then $E' = F$ and $\alpha.F\{\vec{H}/\vec{Y}\} \xrightarrow{\alpha} F\{\vec{H}/\vec{Y}\}$ as required.
- $E \equiv F + G$. Then either $F \xrightarrow{\alpha} E'$ or $G \xrightarrow{\alpha} E'$ and hence the statement follows by induction.

Each case is proven and thus the inductive proof is complete. □

The equivalent lemma for clocks is more complicated than the case for actions because substituting a variable can cause a clock either to be stalled or the substituted process could synchronise on it. Therefore it is also necessary to ensure that whenever a variable is unguarded in $E$, the corresponding processes must neither stall nor enable the clock in question. Hence the following Lemma has a second precondition.

**Lemma 6.5.23** ***Clock Derivatives and Substitution***
*For any expression vector $\vec{H}$ and corresponding variable vector $\vec{Y}$, if $E \xrightarrow{\sigma} E'$ **and** for any $Y \in \vec{Y}$ unguarded in $E$ the corresponding $H \in \vec{H}$ has $\sigma \notin \Sigma_H \cup \mathcal{T}_H$, then $E\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} E'\{\vec{H}/\vec{Y}\}$.*

**Proof.** By induction on the structure of $E$.

- Cases $E \equiv \mathbf{0}$, $E \equiv \Delta$, $E \equiv \Delta_\sigma$, $E \equiv \alpha.F$ and $E \equiv X$ follow trivially as then $\nexists E'.E \xrightarrow{\sigma} E'$.
- $E \equiv \mu X.F$. This case follows the same proof as the corresponding case in Lemma 6.5.22.
- $E \equiv \sigma.F$. Then $E' \equiv F$ and $\sigma.F\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} F\{\vec{H}/\vec{Y}\}$ as required.
- $E \equiv F + G$. We know that whenever $F \rhd Y$ or $G \rhd Y$ with $Y \in \vec{Y}$ the corresponding expression $H \in \vec{H}$ has $\sigma \notin \mathcal{T}_F \cup \Sigma_F$ or $\sigma \notin \mathcal{T}_G \cup \Sigma_G$, respectively. There are three matching rules for $F + G \xrightarrow{\sigma} E'$ which I consider in turn:
  - tSum1. Then $F \xrightarrow{\sigma} F'$ and $G \xrightarrow{\sigma} G'$ with $E' \equiv F' + G'$. By induction we know that $F\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} F'\{\vec{H}/\vec{Y}\}$ and $G\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} G'\{\vec{H}/\vec{Y}\}$. Hence, $(F + G)\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} (F' + G')\{\vec{H}/\vec{Y}\}$ as required.
  - tSum2. Then $F \xrightarrow{\sigma} F'$ with $E' \equiv F'$ and $\sigma \notin \Sigma_G \cup \mathcal{T}_G$. If $F \rhd Y$ it follows that that $\sigma \notin \Sigma_H \cup \mathcal{T}_H$ for the corresponding $H$ and hence, by Lemmas 6.5.18 and 6.5.19, it follows that $\sigma \notin \Sigma_{G\{H/Y\}} \cup \mathcal{T}_{G\{H/Y\}}$. Otherwise, if $F \not\rhd Y$ then by Lemmas 6.5.18 and 6.5.19 it follows that $\Sigma_{G\{H/Y\}} = \Sigma_G$ and $\mathcal{T}_{G\{H/Y\}} = \mathcal{T}_G$, and hence also $\sigma \notin \Sigma_{G\{H/Y\}} \cup \mathcal{T}_{G\{H/Y\}}$. Thus $\sigma \notin \Sigma_{G\{\vec{H}/\vec{Y}\}} \cup \mathcal{T}_{G\{\vec{H}/\vec{Y}\}}$. Therefore, since by induction $F\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} F'\{\vec{H}/\vec{Y}\}$ it follows that that $F + G\{\vec{H}/\vec{Y}\} \xrightarrow{\sigma} F'\{\vec{H}/\vec{Y}\}$ as required.
  - tSum3. This case can be shown by the symmetric proof of tSum2.

Each case is proven and thus the inductive proof is complete. □

Finally in this Section I prove the following very useful Lemma, which shows that a substitution for an unguarded variable yields an expression which retains all the transitions of the substituted expression.

**Lemma 6.5.24** ***Immutability of substituted processes***
*If $E \xrightarrow{\gamma} E'$, then for any process $H$ with $H \rhd X$ and $\gamma \notin \mathcal{T}_H \cup \Sigma_H$, it follows that $H\{E/X\} \xrightarrow{\gamma} \sum E'$ (i.e. an arbitrary length summation: $E' + E' + \cdots + E'$).*

**Proof.** By induction on the structure of process $H$.

- $H \equiv \mathbf{0}$, $H \equiv \Delta$, $H \equiv \Delta_\sigma$, $H \equiv \gamma.G$, $H \equiv \mu X.G$ (for any $G$) and $H \equiv Y$ (with $X \neq Y$). The statement follows trivially since $H \not\rhd X$.
- $H \equiv X$. This case also follows easily, since then $H\{E/X\} \equiv E$.
- $H \equiv F+G$. We know that either $X \rhd F$ or $X \rhd G$. Also, we know that $\gamma \notin \mathcal{T}_F \cup \Sigma_F$ and $\gamma \notin \mathcal{T}_G \cup \Sigma_G$. If $\gamma = \alpha$ then by induction $F\{E/X\} \xrightarrow{\alpha} \sum E'$ or $G\{E/X\} \xrightarrow{\alpha} \sum E'$, and therefore by either Sum1 or Sum2 $(F+G)\{E/X\} \xrightarrow{\alpha} \sum E'$ as required. Alternatively, if $\gamma = \sigma$ then a similar line follows since we know neither $F$ nor $G$ stalls $\sigma$ or has a $\sigma$ transition (by Lemma 6.5.6). The exception is that possibly both $F \rhd X$ and $G \rhd X$, and therefore $(F+G)\{E/X\} \xrightarrow{\sigma} \left(\sum E'\right) + \left(\sum E'\right)$. But this is nevertheless also $\sum E'$ as required.
- $H \equiv \mu Y.G$ (with $X \neq Y$). By rule Rec it suffices to show that $G\{E/X\}\{\mu Y.G/Y\} \xrightarrow{\gamma} \sum E'$. We know that $Y \notin \mathsf{fv}(E)$, because when we substitute $E$ into $\mu Y.G$ we know that the free variables of $E$ are distinct from $Y$ (by our restrictions on substitution made at the beginning of this section). By induction we also know that $G\{E/X\} \xrightarrow{\gamma} \sum E'$. If $\gamma = \alpha$ then by Lemma 6.5.22, $G\{E/X\}\{\mu Y.G/Y\} \xrightarrow{\alpha} \sum E'\{\mu Y.G/Y\}$ follows. But since $Y \notin \mathsf{fv}(E)$ and hence also $Y \notin \mathsf{fv}(E')$, then this is simply $\sum E'$. Alternatively, if $\gamma = \sigma$ then since we know that $\sigma \notin \mathcal{T}_H \cup \Sigma_H$, we can apply 6.5.23 to achieve the same end.

Each case is proven and thus the inductive proof is complete. □

## 6.6 Equivalence Theory

As with CCS and CaSE before it CaSE$^{\text{ip}}$ needs an equivalence theory so that the behaviour of syntactically distinct processes can be identified. This is important so that processes can be decomposed into constituent parts, and some of these parts replaced with other, equivalent processes. To quote Milner again, this equivalence theory should ensure that *"two systems are indistinguishable if we cannot tell them apart without pulling*

*them apart"*. I shall be using *weak bisimulation* as a basis as it fits in well with a component based system where only visible actions need be considered in an interface.

The equivalence theory we now develop is somewhat more complicated than that of CaSE, owing to the set theoretic approach taken in CaSE$^{\text{ip}}$. Unlike CaSE, a patient CaSE$^{\text{ip}}$ process does not register a tick, that is $\sigma \notin \Sigma_E$ does not entail $E \xrightarrow{\sigma}$. Therefore it necessary to ensure that the stable derivatives of equivalent processes possess the same instability set if bisimulation is to be a congruence relation. For instance, the standard definition of strong bisimulation would have processes $a.E + \Delta_\sigma$ and $a.E$ equivalent in this calculus because neither has a $\sigma$ transition. The difference is only observed when a context requests a $\sigma$ tick, where the latter will permit it, but the former won't.

Therefore, to begin this investigation of the equivalence theory I redefine *strong bisimulation* to account for this nuance.

**Definition 6.6.1** ***Temporal Strong Bisimulation (with Explicit Urgency)***
*A symmetric relation $\mathcal{R}$ is a Temporal Strong Bisimulation provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{\gamma} E'$ then $\exists F'.F \xrightarrow{\gamma} F'$ and $\langle E', F'\rangle \in \mathcal{R}$*
- $\Sigma_E = \Sigma_F$

*We write $E \sim F$ if there is a $\mathcal{R}$ with $\langle E, F\rangle \in \mathcal{R}$ and $\mathcal{R}$ is a Temporal Strong Bisimulation.*

Aside from insisting that each pair of matched states have the same instability set, another more subtle difference to CaSE's bisimulation relation is present. Since patience no longer generates a transition, a clock tick must be matched by another tick explicitly. So for instance, whereas in CaSE it was true that $\lfloor \mathbf{0} \rfloor \sigma(\mathbf{0}) \sim \mathbf{0}$, the equivalent equality in CaSE$^{\text{ip}}$, $\sigma.\mathbf{0} \sim \mathbf{0}$, does not hold: since clock ticks decide choices whereas patience does not, the left hand process would cause a choice to be resolved upon $\sigma$ ticking, whereas the right would not. Specifically, $\sigma.\mathbf{0} + a.E \xrightarrow{\sigma} \mathbf{0}$, but $\mathbf{0} + a.E$ has no $\sigma$ transition and thus the choice is only resolved by $a$.

We could argue that for the purposes of checking bisimulation, we could gloss over the difference between explicit and implicit clock ticks (patience), thus making $\sigma.\mathbf{0} \sim \mathbf{0}$ and rewriting Definition 6.6.1 to enforce the second condition only in the first state. This might seem to work, as only the initial actions of a process may decide a choice it is composed into. But this overlooks other subtle differences in CaSE$^{\text{ip}}$. For instance, clock hiding only produces a $\tau$ for explicit clock transitions, where CaSE also produces a $\tau$ for patience. Thus CaSE$^{\text{ip}}$ distinguishes $\sigma.\mathbf{0}$ and $\mathbf{0}$ with $\sigma$ hidden. The same distinction is made by parallel composition and clock renaming, and therefore the second condition must be applied in every subsequent state. Therefore, we cannot overlook the fundamental difference between patience and clock ticks, and thus the second condition must be enforced in all states.

Clearly my new definition of strong bisimulation remains an equivalence relation, that is it is reflexive, transitive and symmetric, since the additional second condition is

itself defined in terms of an equivalence relation (set equality). This is proven in the following Theorem.

**Theorem 6.6.2** *Temporal Strong Bisimulation is an equivalence relation*

**Proof.** We need to show that $\sim$ is symmetric, reflexive and transitive.

- **Symmetry** – follows by definition, since the conditions of each pairing are identical for both components.
- **Reflexivity** – it suffices to note that $Id_{\mathcal{E}}$ is a temporal strong bisimulation since every process can match its own transitions and $\Sigma_E = \Sigma_E$.
- **Transitivity** – consider three processes $E$, $F$ and $G$ with $E \sim F$ and $F \sim G$. Then there are bisimulations $\mathcal{S}_1$ and $\mathcal{S}_2$ with $\langle E, F\rangle \in \mathcal{S}_1$ and $\langle F, G\rangle \in \mathcal{S}_2$. This can be represented by the following two diagrams:

$$\begin{array}{ccc} E & S_1 & F \\ {\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\gamma} \\ E' & S_1 & F' \end{array} \qquad \begin{array}{ccc} F & S_2 & G \\ {\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\gamma} \\ F' & S_2 & G \end{array}$$

  Since $\Sigma_E = \Sigma_F = \Sigma_G$ then composing these bisimulations yields the following commutative diagram:

$$\begin{array}{ccc} E & S_1 \cdot S_2 & G \\ {\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\gamma} \\ E' & S_1 \cdot S_2 & G' \end{array}$$

  and since $\Sigma_E = \Sigma_F = \Sigma_G$ it follows that $S_1 \cdot S_2$ is also a temporal strong bisimulation as required, and hence that $\sim$ is transitive.

Therefore it follows that $\sim$ is an equivalence relation. □

It will also be convenient to prove a few useful algebraic rules about some of the constructs of CaSE$^{ip}$ for the proofs later. I first show that + is commutative, associative and idempotent.

**Lemma 6.6.3** *Summation is commutative, associative and idempotent w.r.t.* $\sim$

**Proof.** I demonstrate each property in turn, by considering various temporal strong bisimulations, $\mathcal{R}$. The second condition of Definition 6.6.1 follows easily in all three cases properties (e.g. note that $\Sigma_{E+F} = \Sigma_{F+E}$). Therefore I concentrate only on the first condition.

- **Commutativity**. $\mathcal{R} = \{\langle E + F, F + E\rangle\} \cup \approxeq$. When $E + F \xrightarrow{\gamma} G$ it follows that the transition was induced by a rule in the set {Sum1,Sum2,tSum1,tSum2,tSum3}. If $\gamma = \alpha$ then either $E \xrightarrow{\alpha} E'$ with $G \equiv E'$ or $F \xrightarrow{\alpha} F'$ with $G \equiv F'$, and then $F + E$ can match these transitions exactly. Otherwise if $\gamma = \sigma$ then either $E \xrightarrow{\sigma} E' \equiv H$ and $\sigma \notin \Sigma_F \cup \mathcal{T}_F$ by tSum2, $F \xrightarrow{\sigma} F' \equiv H$ and $\sigma \notin \Sigma_E \cup \mathcal{T}_E$ by tSum3, or else both $E \xrightarrow{\sigma} E'$ and $F \xrightarrow{\sigma} F'$ with $H \equiv E' + F'$ by tSum1. In the former two cases the exact transition can be matched by the converse rule. In the third case the transition is $F + E \xrightarrow{\sigma} F' + E'$ and clearly $\langle E' + F', F' + E'\rangle \in \mathcal{R}$ as required. The converse of this proof gives symmetry, thus we are done.

- **Associativity**. $\mathcal{R} = \{\langle E + (F + G), (E + F) + G\rangle\} \cup \approxeq$. If $E + (F + G) \xrightarrow{\alpha} H$ then clearly one of the three can perform an $\alpha$ transition, thus leading to identical transitions for both compositions. Otherwise, if $E + (F + G) \xrightarrow{\sigma} H$ then any subset of the three can tick. I consider only three cases (the others are similar):

  - $F \xrightarrow{\sigma} F'$ only. It follows that $\sigma \notin \mathcal{T}_G \cup \Sigma_G$ by tSum2 and also $\sigma \notin \mathcal{T}_E \cup \Sigma_E$ by tSum3. Therefore $E + F \xrightarrow{\sigma} F'$ by tSum3 and also $(E + F) + G \xrightarrow{\sigma} F'$ by tSum2. Since $F' \approxeq F'$ we are done.
  - $E \xrightarrow{\sigma} E'$ and $G \xrightarrow{\sigma} G'$ only. It follows that $F + G \xrightarrow{\sigma} G'$ with $\sigma \notin \mathcal{T}_F \cup \Sigma_F$ by tSum3 and $E + (F + G) \xrightarrow{\sigma} E' + G'$ by tSum1. Therefore also $E + F \xrightarrow{\sigma} E'$ by tSum2 and $(E + F) + G \xrightarrow{\sigma} E' + G'$ by tSum1 with $E' + G' \approxeq E' + G'$ as required.
  - $E \xrightarrow{\sigma} E'$, $F \xrightarrow{\sigma} F'$ and $G \xrightarrow{\sigma} G'$. Then clearly $E + F \xrightarrow{\sigma} E' + F'$ by tSum1. Also $(E + F) + G \xrightarrow{\sigma} (E' + F') + G'$ by tSum1 and clearly $\langle E' + (F' + G'), (E' + F') + G'\rangle \in \mathcal{R}$ as required.

  The converse of this proof gives, symmetry so we are done.

- **Idempotence**. $\mathcal{R} = \{\langle E + E, E\rangle\} \cup \approxeq$. If $E + E \xrightarrow{\alpha} H$ then by Sum1 or Sum2 it follows that $E \xrightarrow{\alpha} E'$ with $H = E'$ and clearly $E' \approxeq E'$. Otherwise, if $E + E \xrightarrow{\sigma} H$ then the only matching rule is tSum1, whence $E \xrightarrow{\sigma} E'$ and thus $H \equiv E' + E'$ and $\langle E' + E', E'\rangle \in \mathcal{R}$ as required. The converse of this proof gives symmetry, so we are done.

With all three properties proved, we are done. □

This shows that the order and replication of summed processes can effectively be ignored. The main use of Lemma 6.6.3 is to allow the rearrangement of subexpressions, in fact we can now consider a summation of sequential agents as simply a *set* of those agents, e.g. $E+(F+G+(F+F)) \sim \sum\{E, F, G\}$. For instance, we can now treat Lemma 6.5.24 as if $\sum E' = E'$, which is clearly much more convenient.

Strong bisimulation provides a useful foundation but it is, nevertheless, very limited for reasoning about processes as it does not abstract silent actions and thus a more useful

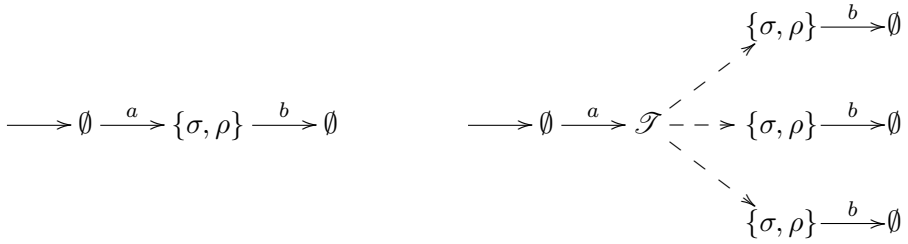



Figure 6.6: Matching weakly accessible stable states (states annotated by $\Sigma_E$)

equivalence theory based on weak bisimulation is sought. I start by directly adapting the regular CCS definition of weak bisimulation, using the standard weak transition relation $\Rightarrow$ as given in Definition 2.3.2 (in Chapter 2).

**Definition 6.6.4** ***Naïve Temporal Weak Bisimulation***
*A symmetric relation $\mathcal{R}$ is a Naïve Temporal Weak Bisimulation provided $\forall \langle E, F \rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{\gamma} E'$ then $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ and $\langle E', F' \rangle \in \mathcal{R}$*

*We write $E \approx_n F$ if $\exists \mathcal{R}.\langle E, F \rangle \in \mathcal{R}$ and $R$ is a n.t.w.b.*

Whilst Naïve Temporal Weak Bisimulation is an equivalence relation as it does not divert from Milner's definition (Milner, 1989*a*), it is not a congruence. In particular, it is not compositional through parallel composition because it does not ensure that stalled clocks are matched. Once again, $a.E + \Delta_\sigma$ and $a.E$ are equivalent in this definition, even though the context $C[\_] = \_ \mid \sigma.F$ distinguishes them. The correction was simple to make in strong bisimulation since every pairing simply had to have the same instability set. We cannot do this in weak bisimulation though, because matching clocks may only be present after a sequence of $\tau$s, i.e. after the process has *stabilised*.

This is illustrated by the two processes in Figure 6.6 (the states are annotated by the stalled clocks set). Performing an $a$ moves the left-hand process directly to a stable state which holds up $\sigma$ and $\rho$. However, the right-hand process move to an unstable state after performing an $a$ with three weakly accessible stable states holding up $\sigma$ and $\rho$ (the dashed arrows represent a $\tau$ sequence). It is not the clocks which are held up in the unstable state which matter (since by definition no clocks can tick at this point), but those held up in each of the *weakly accessible stable states*. If provision is not made for ensuring these match it is possible that a $\tau$ sequence could lead to a stable state where a clock that should be stalled is not. Therefore the clocks stalled after any $\tau$ sequence are defined by the *intersection* of clocks held up in all weakly accessible stable states, or formally:

**Definition 6.6.5** ***Weakly Stalled Clocks***
*The Weakly Stalled Clocks set $\Sigma_E^{\Rightarrow}$ is the set of clocks which are held up by every stable state weakly accessible from $E$. It is defined thus: $\Sigma_E^{\Rightarrow} = \bigcap \{\Sigma_{E'} \mid E \stackrel{\hat{\tau}}{\Rightarrow} E' \text{and } E' \text{ is stable}\}$.*

If a clock is not within the set $\Sigma^{\Rightarrow}_{E}$ then it follows that it is patient in at least one weakly accessible stable state. Hence, a composed process will be allowed to tick this clock should a suitable $\tau$ sequence occur. I also define the equivalent set for active clocks – note the use of union instead of intersection:

**Definition 6.6.6** ***Weakly Active Clocks***
*The Weakly Active Clocks set $\mathcal{T}^{\Rightarrow}_{E}$ is the set of clocks which have explicit transitions in at least one weakly accessible stable state of $E$. It is defined thus: $\mathcal{T}^{\Rightarrow}_{E} = \bigcup\{\mathcal{T}_{E'} \mid E \stackrel{\hat{\tau}}{\Rightarrow} E'\}$*

I now prove the following useful Lemma about weakly stalled clocks:

**Lemma 6.6.7** *If $E \approx_n F$ then $\mathcal{T}_E \cap \Sigma^{\Rightarrow}_{F} = \emptyset$.*

**Proof.** By contradiction. Assume that $E \approx_n F$ and there is a $\sigma$ such that, $\sigma \in \mathcal{T}_E$ and $\sigma \in \Sigma^{\Rightarrow}_{F}$. Then it follows by Lemma 6.5.6 that $\exists E'.E \stackrel{\sigma}{\rightarrow} E'$. Furthermore, since $\sigma \in \Sigma^{\Rightarrow}_{F}$ then by Definition 6.6.5 for any stable $F''$ such that $F \stackrel{\hat{\tau}}{\Rightarrow} F''$ it follows that $\sigma \in \Sigma_{F''}$. Thus by Proposition 6.5.10 it follows that $\sigma \notin \mathcal{T}_{F''}$ and therefore by Lemma 6.5.6 it also follows that $\nexists F'.F'' \stackrel{\sigma}{\rightarrow} F'$, and hence $\nexists F'.F \stackrel{\hat{\sigma}}{\Rightarrow} F'$. Therefore $E \not\approx_n F$ as Definition 6.6.4 cannot be satisfied for $\sigma$, contradicting our assumptions.

Therefore no such $\sigma$ can exist. □

I now use the notion of "weakly stalled clocks" to define a more restricted form of weak bisimulation, which is a smaller relation than the naive definition.

**Definition 6.6.8** ***Temporal Weak Bisimulation (with Explicit Urgency)***
*A symmetric relation $\mathcal{R}$ is a Temporal Weak Bisimulation provided $\forall E\ F.\langle E, F\rangle \in \mathcal{R}$:*

1. *If $E \stackrel{\gamma}{\rightarrow} E'$ then $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$*
2. *$\Sigma^{\Rightarrow}_{F} \subseteq \Sigma_E$*

*We write $E \approxeq F$ if $\exists \mathcal{R}.\langle E, F\rangle \in \mathcal{R}$ and $\mathcal{R}$ is a Temporal Weak Bisimulation.*

The second clause ensures that, for each pairing, every patient clock is matched after an unspecified sequence of $\tau$s. Without this clause $\mathbf{0} \approxeq \Delta$ would follow for instance, which is clearly incorrect as the latter stalls all clocks. In CaSE, since patient clock ticks were regular transitions, all that was required was matching on a theoretically infinite clock universe. This definition overcomes the need for a specific clock context by instead ensuring that pairings only stall equivalent clock sorts. Nevertheless, the second clause seems at first sight unnecessarily complicated and one might try and simplify it to:

$$\textit{If } E \textit{ and } F \textit{ are stable then } \Sigma_E = \Sigma_F$$

thus emulating the definition of strong timed bisimulation; when a stable pairing is made the stalled clocks must match. However, this will not suffice because it assumes that

every process will inevitably stabilise. Processes with $\tau$ loops like $\mu X.\tau.X$ do not, and thus this second clause would have $\mathbf{0} \cong \mu X.\tau.X$ which is incorrect as one permits all clocks whilst the other blocks all clocks indefinitely. (However, if a process doesn't have any infinite $\tau$ loops, this definition would suffice).

With a new definition of equivalence in hand, I now proceed to prove some useful properties about it. Firstly, it is a subset of the Naïve definition:

**Lemma 6.6.9** *Temporal Weak Bisimulation with Explicit Urgency is contained within Naïve Temporal Weak Bisimulation. That is,* $\cong \subseteq \approx_n$.

**Proof.** It suffices to show that every Temporal Weak Bisimulation with Explicit Urgency $\mathcal{R}$ is also a Naïve Temporal Weak Bisimulation. This follows from the fact that the first condition of Definition 6.6.8 is identical to the condition of Definition 6.6.4, and hence every pairing $\langle E, F\rangle$ in $\mathcal{R}$ also satisfies the condition of Naïve Temporal Weak Bisimulation. Therefore it follows that $\mathcal{R}$ is a Naïve Temporal Weak Bisimulation, and hence $\cong \subseteq \approx_n$ as required. $\square$

It also should be the case that $\cong$ is an equivalence relation, like $\sim$ is. It is first convenient to prove the following Lemma which solidifes the idea the weak bisimulation can essentially be seen as bisimulation on a weak transition system.

**Lemma 6.6.10** ***Weak Transition Bisimulation***
*If* $E \cong F$ *then whenever* $E \stackrel{\hat{\gamma}}{\Rightarrow} E'$ *it follows that* $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ *and* $E' \cong F'$.

**Proof.** First of all, if $E \stackrel{\epsilon}{\Rightarrow} E$ then clearly $F \stackrel{\epsilon}{\Rightarrow} F$ with $E \cong F$. Therefore, we need only consider the case when $E \stackrel{\gamma}{\Rightarrow} E'$. In this case, it suffices to fill in the following diagram:

$$\begin{array}{ccc} E & \cong & F \\ {\scriptstyle\gamma}\Downarrow & & {\scriptstyle\hat{\gamma}}\Downarrow \\ E' & \cong & F' \end{array}$$

When $E \stackrel{\gamma}{\Rightarrow} E'$ it follows that $\exists E''\,E'''.E \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} E'' \stackrel{\gamma}{\rightarrow} E''' \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow}$, and since $E \cong F$ it follows that there are corresponding $F$ derivatives as illustrated in the following diagram:

$$\begin{array}{ccccccccc} E & \xrightarrow{\tau} & \cdots\cdots & \xrightarrow{\tau} & E'' & \xrightarrow{\gamma} & E''' & \xrightarrow{\tau} & \cdots\cdots & \xrightarrow{\tau} & E' \\ \cong & & & & \cong & & \cong & & & & \cong \\ F & \stackrel{\hat{\tau}}{\Longrightarrow} & \cdots\cdots & \stackrel{\hat{\tau}}{\Longrightarrow} & F'' & \stackrel{\hat{\gamma}}{\Longrightarrow} & F''' & \stackrel{\hat{\tau}}{\Longrightarrow} & \cdots\cdots & \stackrel{\hat{\tau}}{\Longrightarrow} & F' \end{array}$$

Composing the transitions between $F$ and $F'$ simply gives $F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ as required. $\square$

Now I show that $\cong$ is indeed an equivalence relation.

**Theorem 6.6.11** ***Temporal Weak Bisimulation is an equivalence relation***

**Proof.** We need to show that $\approxeq$ is symmetric, reflexive and transitive.

- **Symmetry** – follows by definition, since the conditions of each pairing are identical for both components.
- **Reflexivity** – it suffices to note that $Id_{\mathcal{E}}$ is a Temporal Strong Bisimulation since whenever a process $E \xrightarrow{\gamma} E'$, it automatically follows that $E \stackrel{\hat{\gamma}}{\Rightarrow} E'$ and $\Sigma_{E'} \subseteq \Sigma_{E'}$.
- **Transitivity** – consider three processes $E$, $F$ and $G$ with $E \approxeq F$ and $F \approxeq G$. Then there are two weak bisimulations $\mathcal{S}_1$ an $\mathcal{S}_2$ with $\langle E, F\rangle \in \mathcal{S}_1$ and $\langle F, G\rangle \in \mathcal{S}_2$. This can be represented by the following two diagrams:

$$\begin{array}{ccc} E & S_1 & F \\ {\scriptstyle\gamma}\downarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ E' & S_1 & F' \end{array} \qquad \begin{array}{ccc} F & S_2 & G \\ {\scriptstyle\gamma}\downarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ F' & S_2 & G \end{array}$$

However, by Lemma 6.6.10 the rightmost diagram can be expressed as:

$$\begin{array}{ccc} F & S_2 & G \\ {\scriptstyle\hat{\gamma}}\Downarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ F' & S_2 & G \end{array}$$

Also, since $\Sigma_E \subseteq \Sigma_F \subseteq \Sigma_G$, and therefore $\Sigma_E \subseteq \Sigma_F$, then composing the two bisimulations yields the following:

$$\begin{array}{ccc} E & S_1 \cdot S_2 & G \\ {\scriptstyle\gamma}\downarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ E' & S_1 \cdot S_2 & G' \end{array}$$

Demonstrating that $S_1 \cdot S_2$ is also a temporal weak bisimulation as required, and hence that $\approxeq$ is transitive.

Therefore it follows that $\approxeq$ is an equivalence relation. □

I now proceed to prove that $\approxeq$ is a congruence with respect to parallel composition and hiding. But first I need to prove a few additional Lemmas. I first formally prove the *maximal progress* property for this calculus:

**Lemma 6.6.12** ***Maximal Progress***
*If $E \xrightarrow{\sigma} E'$ then $\nexists E''.E \xrightarrow{\tau} E''$.*

**Proof.** If $E \xrightarrow{\sigma} E'$ then by Lemma 6.5.6 it follows that $\sigma \in \mathcal{T}_E$. Then, by Proposition 6.5.10 it follows that $\sigma \notin \Sigma_E$, and therefore by Proposition 6.5.9 $\tau \notin \mathscr{A}_E$. Finally, by Lemma 6.5.6 we can conclude that $\nexists E''.E \xrightarrow{\tau} E'$ as required. □

From now I will sometimes use the shorthand $E \xrightarrow{\gamma}$ to refer to $\exists E'.E \xrightarrow{\gamma} E'$ and $E \not\xrightarrow{\gamma}$ to refer to $\nexists E'.E \xrightarrow{\gamma} E'$.

**Lemma 6.6.13** ***Stable processes and Weak Bisimulation***
*When $E \approxeq F$, if $\tau \notin \mathscr{A}_E$ then whenever $F \stackrel{\hat{\tau}}{\Rightarrow} F'$ it follows that $E \approxeq F'$.*

**Proof.** By Lemma 6.5.6 it follows that $E$ can do no $\tau$ transitions. But since $E \approxeq F$ then by Definition 6.6.8 whenever $\exists F'.F \xrightarrow{\tau} F'$ it follows that $E \stackrel{\hat{\tau}}{\Rightarrow} E'$. Thus $E$ must match $F$ by doing an empty sequence, i.e. $E \stackrel{\epsilon}{\Rightarrow} E$ with $E' = E$. Hence for any $\tau$ sequence which $F$ can perform to become $F'$ it follows that $E \approxeq F'$ as required. □

I can now also prove the following Lemma:

**Lemma 6.6.14** *If $E \approxeq F$ and $E$ is patient then $\mathcal{T}_E = \mathcal{T}_F^{\Rightarrow}$.*

**Proof.** I will show that (1) $\mathcal{T}_F^{\Rightarrow} \subseteq \mathcal{T}_E$ and (2) $\mathcal{T}_E \subseteq \mathcal{T}_F^{\Rightarrow}$.

1. It suffices to show for any $\sigma$ that $\sigma \in \mathcal{T}_F^{\Rightarrow}$ entails $\sigma \in \mathcal{T}_E$. If $\sigma \in \mathcal{T}_F^{\Rightarrow}$ then by Definition 6.6.6 it follows that there is an $F'$ such that $F \stackrel{\hat{\tau}}{\Rightarrow} F'$ and $\sigma \in \mathcal{T}_{F'}$. Therefore by 6.5.6 it follows that $\exists F'.F' \xrightarrow{\sigma} F''$. Since $E$ is patient, we know by Proposition 6.5.9 that $E$ is also stable, i.e. that $\tau \notin \mathscr{A}_E$. Thus by Lemma 6.6.13 we know that $E \approxeq F'$ and by Definition 6.6.8 that $\exists E'.E \xrightarrow{\sigma} E'$. Finally by Lemma 6.5.13 it follows that $\sigma \in \mathcal{T}_E$ as required.

2. By contradiction. Assume there is a $\sigma$ with $\sigma \in \mathcal{T}_E$ and $\sigma \notin \mathcal{T}_F^{\Rightarrow}$. Then by Definition 6.6.6 $\exists F'.F \stackrel{\hat{\tau}}{\Rightarrow} F'$ with $\sigma \notin \mathcal{T}_{F'}$ and since $E$ is stable by Lemma 6.6.13 $\langle E, F' \rangle \in \mathcal{R}$. Furthermore, by Lemma 6.5.6 it follows that $\nexists F''.F' \xrightarrow{\sigma} F''$. But therefore $\mathcal{R}$ is not a weak bisimulation, since by Lemma 6.5.12 $\exists E'.E \xrightarrow{\sigma} E'$ but $\nexists F''.F \stackrel{\hat{\sigma}}{\Rightarrow} F''$, meaning Definition 6.6.8 cannot be satisfied. Therefore no such $\sigma$ exists and $\mathcal{T}_E \subseteq \mathcal{T}_F^{\Rightarrow}$.

From the two inclusions I conclude that $\mathcal{T}_E = \mathcal{T}_F^{\Rightarrow}$. □

Finally I can prove that Temporal Weak Bisimulation with Explicit urgency is a congruence relation with respect to all operators other than + and $\mu X$, following Milner's outline (Milner, 1989*a*). However, in order to do this I need an extended version of Definition 6.6.8 which will account for expressions with free process variables.

**Definition 6.6.15** ***Open Temporal Weak Bisimulation (with Explicit Urgency)***
*A symmetric relation $\mathcal{R}$ is an Open Temporal Weak Bisimulation provided $\forall E\ F.\langle E, F \rangle \in \mathcal{R}$:*

1. *If $E \xrightarrow{\gamma} E'$ then $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ and $\langle E', F' \rangle \in \mathcal{R}$*
2. *If $E \rhd X$ then $F \rhd X$*
3. *$\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$*

*We write $E \approxeq F$ if $\exists \mathcal{R}.\langle E, F \rangle \in \mathcal{R}$ and $\mathcal{R}$ is an Open Temporal Weak Bisimulation.*

The second condition ensures that open terms possess the same set of unguarded free variables. This is necessary to ensure, for instance, that $a.E + X \cong a.E + Y$ does not hold, as in the context of $\mu X.\_$ they have different behaviour. Hence from now on I use this redefinition of $\cong$. Now, at last, I proceed to prove compositionality of $\cong$.

**Theorem 6.6.16** ***Compositionality of Temporal Weak Bisimulation***
*Temporal Weak Bisimulation is a congruence with respect to all the operators of* $\mathsf{CaSE}^{ip}$ *other than + and* $\mu X$.

**Proof.** In order to show that this is true I construct a relation $\mathcal{R} = \{\langle C[E], C[F]\rangle \mid E \cong F\} \cup \cong$, where $C$ is a context (e.g. $C[\_] = \_ \mid G$) and show that it is a weak bisimulation. That is, that every element satisfies both conditions of Definition 6.6.8. Specifically, I will demonstrate:

1. Whenever $\exists H.C[E] \xrightarrow{\gamma} H$ it follows that $\exists I.C[F] \stackrel{\hat{\gamma}}{\Rightarrow} I$ with $\langle H, I\rangle \in \mathcal{R}$ (for static operators) or $H \cong I$ (for dynamic operators);
2. Whenever $C[E] \rhd X$ it follows that $C[F] \rhd X$; and
3. $\Sigma^{\Rightarrow}_{C[F]} \subseteq \Sigma_{C[E]}$.

To prove the first of these I consider each rule from Table 6.3 which could have induced the $C[E]$ transition, and prove the case from the premises of the rule. I enumerate the possible constructs to be proved for below and then proceed with the structural induction on these cases for the sub-language of $\mathsf{CaSE}^{\mathrm{ip}}$.

1. $C[E] = \alpha.E$
2. $C[E] = \sigma.E$
3. $C[E] = E \mid G$
4. $C[E] = E \setminus a$
5. $C[E] = E\{\!|a \mapsto b|\!\}$
6. $C[E] = E\{\!|\sigma \mapsto a|\!\}$
7. $C[E] = E/\sigma$

1. $C[E] = \alpha.E$. By rule Act it follows that $\alpha.E \xrightarrow{\alpha} E$. The proof of condition (1) of Definition 6.6.8 follows trivially, specifically whenever $\alpha.E \xrightarrow{\alpha} E$ it follows by rule Act that $\alpha.F \xrightarrow{\alpha} F$, therefore $\alpha.F \stackrel{\hat{\alpha}}{\Rightarrow} F$ and since $E \cong F$, then $\langle E, F\rangle \in \mathcal{R}$. Condition (2) follows trivially since there is no $X$ such that $\alpha.E \rhd X$ or $\alpha.F \rhd X$. Condition (3) holds when $\alpha = a$, in which case $\Sigma_{a.E} = \Sigma^{\Rightarrow}_{a.F} = \emptyset$, and when $\alpha = \tau$, in which case $\Sigma_{\tau.E} = \mathscr{T}$. In both cases clearly $\Sigma^{\Rightarrow}_{\alpha.F} \subseteq \Sigma_{\alpha.E}$ as required.

2. $C[E] = \sigma.E$. Similar to the case for $\alpha.E$.

3. $C[E] = E \mid G$. I first give a proof of condition (1) for each possible $\gamma$:

   (a) Case when $\gamma = \alpha$. This case follows the standard proof for CCS (Milner, 1989*a*).

   (b) Case when $\gamma = \sigma$. There are three possible rules which induce $E \mid G \xrightarrow{\sigma} H$: tCom1, tCom2 and tCom3. I consider all three cases in turn to prove condition (1) (though in a different order):

   - $E \xrightarrow{\sigma} E', \sigma \notin \Sigma_G \cup \mathcal{T}_G$ and $H = E' \mid G$ (matching tCom2)
     - By Definition 6.6.8 it follows that $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ with $E' \approx F'$ (and therefore $\langle E' \mid G, F' \mid G\rangle \in \mathcal{R}$), and $\Sigma^{\Rightarrow}_F \subseteq \Sigma_E$
     - Thus $F \xrightarrow{\tau}{}^* F'' \xrightarrow{\sigma} F''' \xrightarrow{\tau}{}^* F'$
     - By Com1, $F \mid G \xrightarrow{\tau}{}^* F'' \mid G$ and $F''' \mid G \xrightarrow{\tau}{}^* F' \mid G$
     - Since $F'' \xrightarrow{\sigma} F'''$ and $\sigma \notin \Sigma_G \cup \mathcal{T}_G$, by tCom2 it follows that $F'' \mid G \xrightarrow{\sigma} F''' \mid G$
     - Therefore also $F \mid G \stackrel{\hat{\alpha}}{\Rightarrow} F' \mid G$, satisfying condition (1).
   - $G \xrightarrow{\sigma} G', \sigma \notin \Sigma_E \cup \mathcal{T}_E$ and $H = E \mid G'$ (matching tCom3)
     - By Definition 6.6.8 (2) it follows that $\Sigma^{\Rightarrow}_F \subseteq \Sigma_E$
     - Hence according to Definition 6.6.5 there is a $\tau$ derivative $F''$ of $F$ (i.e. $F \xrightarrow{\tau}{}^* F''$) with $\Sigma_{F''} \subseteq \Sigma_E$ and $F''$ is stable
     - By Lemma 6.6.12 (maximal progress) since $E \mid G \xrightarrow{\sigma}$ it follows that $E \mid G \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$
     - Therefore also $E \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$ and by Lemma 6.5.6 it follows that $\tau \notin \mathscr{A}_E$
     - Thus by Lemma 6.6.13, $E \approx F''$
     - Since $F'' \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$ it follows that $\Sigma^{\Rightarrow}_{F''} = \Sigma_{F''}$ and therefore $\Sigma_{F''} \subseteq \Sigma_E$
     - Hence , since $\sigma \notin \Sigma_E$, it follows that $\sigma \notin \Sigma_{F''}$
     - Furthermore, by Lemma 6.6.14 it follows that $\mathcal{T}^{\Rightarrow}_{F''} = \mathcal{T}_E$ and since $F''$ is stable, $\mathcal{T}_{F''} = \mathcal{T}_E$. Therefore $\sigma \notin \mathcal{T}_{F''}$
     - By Com1 $F \mid G \stackrel{\hat{\tau}}{\Rightarrow} F'' \mid G$ and by tCom3 $F'' \mid G \xrightarrow{\sigma} F'' \mid G'$
     - Thus $F \mid G \stackrel{\sigma}{\Rightarrow} F'' \mid G'$ with $\langle E \mid G', F'' \mid G'\rangle \in \mathcal{R}$ satisfying condition (1).
   - $E \xrightarrow{\sigma} E', G \xrightarrow{\sigma} G'$ and $H = E' \mid G'$ (matching tCom1)
     - By Definition 6.6.8 $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ with $E' \approx F'$
     - Hence by Com1 and tCom3, $F \mid G \stackrel{\hat{\sigma}}{\Rightarrow} F' \mid G'$ with $\langle E' \mid G', F' \mid G'\rangle \in \mathcal{R}$ satisfying condition (1).

   Condition (2) holds because $E \rhd X$ implies $F \rhd X$, and hence $E \mid G \rhd X$ implies $F \mid G \rhd X$. To show condition (3) holds, i.e. $\Sigma^{\Rightarrow}_{F \mid G} \subseteq \Sigma_{E \mid G}$, first consider that $E \mid G$ is either patient or impatient. If $E \mid G$ is impatient, i.e. $\Sigma_{E \mid G} = \mathscr{T}$, then

$\Sigma^{\Rightarrow}_{F \mid G} \subseteq \mathscr{T}$ trivially. Otherwise, when $E \mid G$ is patient, consider that $\Sigma_{E \mid G} = \Sigma_E \cup \Sigma_G$. By Proposition 6.5.9 we know that $E \mid G$ is also stable, i.e. $\tau \notin \mathscr{A}_{E \mid G}$. Furthermore since $G$ is stable it follows that $\Sigma^{\Rightarrow}_{G} = \Sigma_G$. Since $E \mid G$ is stable, it follows that $E$ and $G$ don't synchronise, that is either there does not exist an $a$, $E'$ and $G'$ such that $E \xrightarrow{a} E'$ and $G \xrightarrow{\overline{a}} G'$. Furthermore by Lemma 6.6.13 every $\tau$ derivative $F''$ of $F$ is weakly bisimilar with $E$. Therefore, it follows for any such $F''$ that $\nexists F'.F'' \xrightarrow{a} F'$ with $G \xrightarrow{\overline{a}} G'$. Since any such $F''$ and $G$ do not synchronise, the only possible $\tau$ transitions of $F \mid G$ are those already present in $F$. Hence, $\Sigma^{\Rightarrow}_{F \mid G} = \Sigma^{\Rightarrow}_{F} \cap \Sigma^{\Rightarrow}_{G} = \Sigma^{\Rightarrow}_{F} \cap \Sigma_G$, which is clearly a subset of $\Sigma_E \cup \Sigma_G$ since $\Sigma^{\Rightarrow}_{F} \subseteq \Sigma_E$.

4. $C[E] = E \setminus a$. I first give proof of condition (1) for $C[E] = E \setminus a$ for each possible $\gamma$:

   (a) Case when $\gamma = \alpha$. This case follows the standard proof for CCS (Milner, 1989*a*).

   (b) Case when $\gamma = \sigma$. Easily follows, since if $E \setminus a \xrightarrow{\sigma} E' \setminus a$ then by rule Res, $E \xrightarrow{\sigma} E'$. Hence also $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ (since $E \approxeq F$ and therefore by rule Res $F \setminus a \stackrel{\hat{\sigma}}{\Rightarrow} F' \setminus a$ as required.

   Condition (2) follows directly from the knowledge that any subexpression of $E$ is also a subexpression of $E \setminus a$. Condition (3) follows from the fact that we know that $\Sigma^{\Rightarrow}_{F} \subseteq \Sigma_E$. Since for any $P$, $\Sigma_{P \setminus a} = \Sigma_P$ the condition follows easily.

5. $C[E] = E\{\!|a \mapsto b|\!\}$. Proof of condition (1) is standard for CCS (Milner, 1989*a*), except for clock ticks in which case the condition follows since if $E\{\!|a \mapsto b|\!\} \xrightarrow{\sigma} E'\{\!|a \mapsto b|\!\}$ then $E \xrightarrow{\sigma} E'$, and then $F\{\!|a \mapsto b|\!\} \stackrel{\hat{\sigma}}{\Rightarrow} F'\{\!|a \mapsto b|\!\}$ follows easily. Condition (2) follows easily as the above case. Condition (3) follows trivially from the fact that $\Sigma_{E\{\!|a \mapsto b|\!\}} = \Sigma_E$.

6. $C[E] = E\{\!|\sigma \mapsto a|\!\}$. To prove condition (1) for the case when $C[E] = E\{\!|\sigma \mapsto a|\!\}$, I assume $E\{\!|\sigma \mapsto a|\!\} \xrightarrow{\gamma} E'\{\!|\sigma \mapsto a|\!\}$ and prove for every $\gamma$ possibility:

   - Case when $\gamma = \rho$. Whenever $E\{\!|\sigma \mapsto a|\!\} \xrightarrow{\rho} E'\{\!|\sigma \mapsto a|\!\}$ then by rule Rel it follows that $E \xrightarrow{\rho} E'$. Hence, since $E \approxeq F$ it follows that $F \stackrel{\hat{\rho}}{\Rightarrow} F'$ with $E' \approxeq F'$. Therefore, also by rule Rel $F\{\!|\sigma \mapsto a|\!\} \stackrel{\hat{\rho}}{\Rightarrow} F'\{\!|\sigma \mapsto a|\!\}$, and $\langle E'\{\!|\sigma \mapsto a|\!\}, F'\{\!|\sigma \mapsto a|\!\}\rangle \in \mathcal{R}$ as required.
   - Case when $\gamma = b$. If $b \neq a$ then proof of condition (1) follows the same structure as the previous case. Otherwise, if $b = a$ then by Rel it follows that $E \xrightarrow{\sigma} E'$. Therefore also $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ with $E' \approxeq F'$. By Rel $F\{\!|\sigma \mapsto a|\!\} \stackrel{a}{\Rightarrow} F'\{\!|\sigma \mapsto a|\!\}$, and therefore $\langle E'\{\!|\sigma \mapsto a|\!\}, F'\{\!|\sigma \mapsto a|\!\}\rangle \in \mathcal{R}$ as required.

   Condition (2) follows easily as the case for $E\{\!|a \mapsto b|\!\}$. Condition (3) follows easily, since $\Sigma_E = \Sigma_{E\{\!|\sigma \mapsto a|\!\}}$.

7. $C[E] = E/\sigma$. To prove the case when $C[E] = E/\sigma$, we assume $E/\sigma \xrightarrow{\gamma} E'/\sigma$ and prove for every $\gamma$ possibility.

- Case when $\gamma = a$ – follows easily.
- Case when $\gamma = \tau$:
  - $E \xrightarrow{\tau} E', H = E'/\sigma$ – follows easily.
  - $E \xrightarrow{\sigma} E', H = E'/\sigma$
    - By Definition 6.6.8 $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ with $E' \cong F'$
    - Therefore, $\exists F''\ F'''.F \stackrel{\hat{\tau}}{\Rightarrow} F'' \xrightarrow{\sigma} F''' \stackrel{\hat{\tau}}{\Rightarrow} F'$
    - Hence, by rules Hid and tHid1 it follows that $F\{\!|\sigma \mapsto a|\!\} \stackrel{\hat{\tau}}{\Rightarrow} F''\{\!|\sigma \mapsto a|\!\} \xrightarrow{\tau} F'''\{\!|\sigma \mapsto a|\!\} \stackrel{\hat{\tau}}{\Rightarrow} F'$
    - Therefore $F\{\!|\sigma \mapsto a|\!\} \stackrel{\hat{\tau}}{\Rightarrow} F'\{\!|\sigma \mapsto a|\!\}$, and $\rangle E'\{\!|\sigma \mapsto a|\!\}, F'\{\!|\sigma \mapsto a|\!\}\langle \in \mathcal{R}$ as required.
- Case when $\gamma = \rho$:
  - $E \xrightarrow{\rho} E', H = E'/\sigma$. Follows easily through rule tHid2.

Condition (2) follows easily as the above case. To show condition (3) consider that when $\sigma \in \mathcal{T}_E$, $\Sigma_{E/\sigma} = \mathscr{T}$ and thus $\Sigma_F^{\Rightarrow} \subseteq \Sigma_{E/\sigma}$ trivially. Otherwise the proof follows through as for the previous cases.

The converse cases give symmetry. Therefore the proof is complete. □

As is true for CaSE, Temporal Weak Bisimulation is not a congruence with respect to +, for much the same reasons. Firstly there is the standard leading $\tau$ issue present in CCS (Milner, 1989*a*) which can be easily fixed. Secondly, since + can be static with respect to clocks, for instance in the process $\sigma.E + \sigma.F$, it is important that any leading clock tick sequences are unbroken by $\tau$s. For instance the processes $\sigma.\rho.E$ and $\sigma.\tau.\rho.E$ whilst weakly bisimilar can be distinguished by the context $C[\_] = \_ + \sigma.\rho.F$, since the $\tau$ in the former process will cause the choice to be resolved.

Temporal Weak Bisimulation is not a congruence with respect to $\mu X$ as a side-effect of $+$ not being a congruence. In fact it is possible to directly represent the contexts given above as open terms. For example, it is clear that $\tau.c.(\sigma.P + X) \cong c.(\sigma.P + X)$, but placing both processes in the context of $C[\_] = \mu X.\_$ will lead to different behaviours since the substitution for $X$ in the former will stall $\sigma$, but not so in the latter. This represents the two processes $\tau.c.E$ and $c.E$, which are weakly bisimilar, but distinguished by the context $C[\_] = \sigma.P + \_$. Therefore, fixing the problem with $+$ will also fix the problem with $\mu X$.

To fix this issue, CaSE forces initial clock sequences to be strongly matched. I do the same and add the requirement of matching instability sets. This leads to a new definition of *Temporal Observation Congruence*.

**Definition 6.6.17** ***Temporal Observation Congruence (with Explicit Urgency)*** *A symmetric relation $\mathcal{R}$ is a temporal observation congruence provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

1. *If $E \xrightarrow{\alpha} E'$ then $\exists F'.F \stackrel{\alpha}{\Rightarrow} F'$ and $E' \approx F'$*

2. *If* $E \xrightarrow{\sigma} E'$ *then* $\exists F'.F \xrightarrow{\sigma} F'$ *and* $\langle E', F' \rangle \in \mathcal{R}$
3. *If* $E \rhd X$ *then* $F \rhd X$
4. $\Sigma_F \subseteq \Sigma_E$

*We write* $E \approxeq^u F$ *if* $\exists \mathcal{R}.\langle E, F \rangle \in \mathcal{R}$ *and* $\mathcal{R}$ *is a temporal observation congruence.*

I first prove that this relation is compositional with respect to summation:

**Lemma 6.6.18** ***Compositionality wrt.*** $+$
*Temporal Observation Congruence is a congruence with respect to* $+$*.*

**Proof.** To show this we construct a relation $\mathcal{R} = \{\langle E + G, F + G \rangle | E \approxeq^u F\} \cup \approxeq^u$ for any $G$ and show it is a temporal observation congruence. We do this by induction on the possible transitions of $E + G \xrightarrow{\gamma} H$ and showing that the four conditions of Definition 6.6.17 hold.

1. Case when $\gamma = \alpha$. This case follows the standard proof for CCS (Milner, 1989*a*).
2. Case when $\gamma = \sigma$, there are three rules which could induce $E + G \xrightarrow{\gamma} H$:
   - By tSum1, $E \xrightarrow{\sigma} E', G \xrightarrow{\sigma} G'$ with $H \equiv E' + G'$. By Definition 6.6.17, $F \xrightarrow{\sigma} F'$ with $E' \approxeq^u F'$ and thus by tSum1, $F + G \xrightarrow{\sigma} F' + G'$ with $\langle E' + G', F' + G' \rangle \in \mathcal{R}$ as required.
   - By tSum2, $E \xrightarrow{\sigma} E', \sigma \notin \Sigma_G \cup \mathcal{T}_G$ with $H \equiv E'$. By Definition 6.6.17, $F \xrightarrow{\sigma} F'$ with $E' \approxeq^u F'$, and therefore $\langle E', F' \rangle \in \mathcal{R}$ as required.
   - By tSum3, $G \xrightarrow{\sigma} G', \sigma \notin \Sigma_E \cup \mathcal{T}_E$ with $H \equiv G'$. By Definition 6.6.17, $\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$ and therefore $\sigma \notin \Sigma_F$. Furthermore, since $\sigma \notin \mathcal{T}_E$ then $\sigma \notin \mathcal{T}_F$. Therefore it follows that $F + G \xrightarrow{\sigma} G'$ and clearly $\langle G', G' \rangle \in \mathcal{R}$.
3. Condition (2) holds easily, since if $X$ is a subprocess $E + G$ then clearly it is either a subprocess of $E$ or $G$. In the former case it can be matched in $F + G$ by an equivalent subprocess of $F$, or in the latter it can be matched by $G$. To show condition (3) holds, that $\Sigma_{F+G} \subseteq \Sigma_{E+G}$, consider from Definition 6.6.17 that $\Sigma_F \subseteq \Sigma_E$. Therefore, it follows that $\Sigma_{F+G} \subseteq \Sigma_{E+G}$ as required.

The converse cases give symmetry. Thus our proof is complete. □

In order to prove this for the recursion operator I also need to define an *"up to"* relation. This allow a bisimulation to be represented as a smaller relation by only storing pairs which are unique up to bisimilarity, thus avoiding duplication. The following relation is an adaptation of the one found in Sangiorgi and Milner (1992) which corrects the unsound original version found in Milner (1989*a*).

**Definition 6.6.19** ***Asymmetric Temporal Weak Bisimulation up to*** $\approxeq$
*A relation* $\mathcal{R}$ *is an Asymmetric Temporal Weak Bisimulation up to* $\approxeq$ *provided* $\forall E\ F.\langle E, F \rangle \in \mathcal{R}$*:*

1. *If* $E \xrightarrow{\gamma} E'$ *then* $\exists F' F''.F \stackrel{\hat{\gamma}}{\Rightarrow} F'$ *and* $\langle E', F' \rangle \in \sim \mathcal{R} \approxeq$
2. *If* $E \rhd X$ *then* $F \rhd X$
3. $\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$

Using a slight modification of the standard proof, I can show that every Temporal Weak Bisimulation up to $\approxeq$ is contained within $\approxeq$. After Milner, I do this in two parts:

**Lemma 6.6.20** *If* $\mathcal{R}$ *is an Asymmetric Temporal Weak Bisimulation up to* $\approxeq^u$ *then* $\sim \mathcal{R} \approxeq$ *is a Temporal Weak Bisimulation.*

**Proof.** First note that every member of $\mathcal{R}$ satisfies the second condition of Definition 6.6.8 and therefore we need only consider the satisfaction of the first condition. To do this I follow the same procedure as in Milner (1989*a*). To show that $\sim \mathcal{R} \approxeq$ is a Temporal Weak Bisimulation assume that $\langle E, F \rangle \in \sim \mathcal{R} \approxeq$ and that $\exists E'.E \xrightarrow{\alpha} E'$. It then suffices to fill in the following diagram:

$$\begin{array}{ccc} E & \sim \mathcal{R} \approxeq & F \\ {\scriptstyle\gamma}\downarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ E' & \sim \mathcal{R} \approxeq & F' \end{array}$$

First I assume two process $E_1$ and $F_1$ with $E \sim E_1$, $\langle E_1, F_1 \rangle \in \mathcal{R}$ and $F \approxeq F_1$. We can then build the following diagrams from left to right:

$$\begin{array}{ccc} E & \sim & E_1 \\ {\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\gamma} \\ E' & \sim & E_1' \end{array} \qquad \begin{array}{ccccc} E_1 & \mathcal{R} & F_1 \\ {\scriptstyle\gamma}\swarrow & & \Downarrow{\scriptstyle\hat{\gamma}} \\ E_1' & \sim & \mathcal{R} & \approxeq & F_1' \end{array} \qquad \begin{array}{ccc} F_1 & \approxeq & F \\ \Downarrow{\scriptstyle\hat{\gamma}} & & \Downarrow{\scriptstyle\hat{\gamma}} \\ F_1' & \approxeq & F' \end{array}$$

The right-most diagram follows by Lemma 6.6.10 since $F \approxeq F_1$. Composing these three diagrams gives the required result. □

Finally, I can invoke the following standard Lemma from Milner (1989*a*):

**Lemma 6.6.21** *If* $\mathcal{S}$ *is a Temporal Weak Bisimulation up to* $\approxeq$ *then* $\mathcal{S}$ *then* $\mathcal{S} \subseteq \approxeq$.

**Proof.** Since, by Lemma 6.6.20, $\sim \mathcal{S} \approxeq$ is a temporal weak bisimulation then it follows that $\sim \mathcal{S} \approxeq \subseteq \approxeq$. But since $Id_{\mathcal{E}} \subseteq \sim$ and $Id_{\mathcal{E}} \subseteq \approxeq$ and hence $\mathcal{S} \subseteq \sim \mathcal{S} \approxeq$ it follows that $\mathcal{S} \subseteq \approxeq$ as required. □

Therefore, if we wish to show that $E \approxeq F$ it suffices to find a temporal weak bisimulation up to $\approxeq$ containing $\langle E, F \rangle$. Furthermore, we need an equivalent technique for temporal observation congruence.

**Definition 6.6.22** ***Asymmetric Temporal Observation Congruence up to*** $\approxeq^u$
*A relation* $\mathcal{R}$ *is an Asymmetric Temporal Observation Congruence up to* $\approxeq^u$ *provided that* $\forall E\ F.\langle E, F \rangle \in \mathcal{R}$*:*

1. *If* $E \xrightarrow{\alpha} E'$ *then* $\exists F'.F \stackrel{\hat{\alpha}}{\Rightarrow} F'$ *and* $E' \approxeq F'$
2. *If* $E \xrightarrow{\sigma} E'$ *then* $\exists F'.F \xrightarrow{\sigma} F'$ *and* $\langle E', F' \rangle \in \sim\mathcal{R}\approxeq^u$
3. *If* $E \rhd X$ *then* $F \rhd X$
4. $\Sigma_F \subseteq \Sigma_E$

A much simpler diagram chase than the one used for temporal weak bisimulation up to can be used to demonstrate that any pair within a temporal observation congruence up to $\approxeq^u$ are also within $\approxeq^u$. We can now apply these techniques to prove that temporal observation congruence is compositional with respect to recursion. Because every temporal observation congruence $\mathcal{R}$ relies on action derivatives being weakly bisimilar it is first necessary to prove the Lemma below.

**Lemma 6.6.23** ***Temporal weak bisimulation and substitution***
*If* $E \approxeq F$ *then it follows that for any* $G$ *and* $X$, $E\{G/X\} \approxeq F\{G/X\}$.

**Proof.** To prove this I construct a relation $\mathcal{R} = \{\langle E\{G/X\}, F\{G/X\}\rangle \,|\, E \approxeq F\} \cup \approxeq$ and show that it is a temporal weak bisimulation. I do this by induction on the possible transitions of $E\{G/X\} \xrightarrow{\gamma} H$ and show that the corresponding pairing satisfies the conditions of Definition 6.6.15. For reference, this proof will follow a similar line of argument to the proof of Lemma 6.5.16. Conditions (2) and (3) of Definition 6.6.15 follow easily for all such expressions, since substituting for $X$ simply removes this variable from the guarded variables and a substitution for the same variable does not alter the instability sets. Hence I focus on condition (1).

First notice that if $E \rhd X$ then $E\{G/X\}$ contains a collection of $\mu$-guarded subexpressions of the form $\gamma.J$ which together form the $\gamma$ transition, one such expression if $\gamma = \alpha$ or many if $\gamma = \sigma$. Hence, both $G$ and $E$ may contain a collection of these subexpressions. Therefore we know that any transitions are somewhere derived through a summation. For instance $E = a.\mathbf{0} + X$ and $G = b.\mathbf{0}$. Therefore, we can split up the cases for $E\{G/X\} \xrightarrow{\gamma}$ according to how the transition was induced.

- $\gamma = \alpha$, then either:
  - $E \xrightarrow{\alpha} E'$. Then by Lemma 6.5.22 it follows that $E\{G/X\} \xrightarrow{\alpha} E'\{G/X\}$. Furthermore, since $E \approxeq F$ then $F \stackrel{\hat{\alpha}}{\Rightarrow} F'$ and also $F\{G/X\} \stackrel{\hat{\alpha}}{\Rightarrow} F'\{G/X\}$, with $E' \approxeq F'$. Thus $\langle E'\{G/X\}, F'\{G/X\}\rangle \in \mathcal{R}$ as required.
  - $G \xrightarrow{\alpha} G'$ (with $E \rhd X$). Then by Lemma 6.5.24 we know that $E\{G/X\} \xrightarrow{\alpha} G'$. Therefore also $F\{G/X\} \xrightarrow{\alpha} G'$ since $F \rhd X$ and the bound variables of $F$ are likewise independent of those in $G$, and $\langle G', G'\rangle \in \mathcal{R}$ as required.
- $\gamma = \sigma$, then either:

- $E \xrightarrow{\sigma} E'$ and $\sigma \notin \mathcal{T}_G \cup \Sigma_G$. Then by Lemma 6.5.23 it follows that $E\{G/X\} \xrightarrow{\sigma} E'\{G/X\}$. Since $E \approxeq F$ it follows that $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$ with $E' \approxeq F'$. Therefore by Lemmas 6.5.22 and 6.5.23 it follows that $F\{G/X\} \stackrel{\hat{\sigma}}{\Rightarrow} F'\{G/X\}$, and $\langle E'\{G/X\}, F'\{G/X\}\rangle \in \mathcal{R}$ as required.
- $G \xrightarrow{\sigma} G'$ and $\sigma \notin \mathcal{T}_E \cup \Sigma_E$ (with $E \rhd X$). Then by Lemma 6.5.24 we know that $E\{G/X\} \xrightarrow{\sigma} G'$. Now, because $E \approxeq F$ it may follow that $F \xrightarrow{\tau}$ (which could potentially cause problems). However, since $\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$ and $\Sigma_E \neq \mathscr{T}$ we know that there is a stable $\tau$ derivative $F'$ of $F$ by Lemma 6.5.6, i.e. $F \stackrel{\hat{\tau}}{\Rightarrow} F'$ with $\Sigma_{F'} \neq \mathscr{T}$. Furthermore, it follows by Lemma 6.6.13 that $E \approxeq F'$. Therefore, it also follows that $F' \rhd X$ with $\mathcal{T}_E = \mathcal{T}_{F'}$ and $\Sigma_{F'} \subseteq \Sigma_E$. Hence $\sigma \notin \mathcal{T}_{F'} \cup \Sigma_{F'}$ and thus $F'\{G/X\} \xrightarrow{\sigma} G'$ (by Lemma 6.5.24), with $\langle G', G'\rangle \in \mathcal{R}$ as required.
- $E \xrightarrow{\sigma} E'$ and $G \xrightarrow{\sigma} G'$ (with $E \rhd X$). Thus also $F \stackrel{\sigma}{\Rightarrow} F'$ and $F \rhd X$, with $E' \approxeq F'$. We also know that $\exists F''F'''.F \stackrel{\hat{\tau}}{\Rightarrow} F'' \xrightarrow{\sigma} F'''$, $E \approxeq F''$ and thus $F'' \rhd X$. We know that there are $n$ subexpressions of $E$ of the form $\sigma.I_i$ and thus $E' = \sum_{i=0}^{n} I_i\{\vec{K_i}/\vec{Y_i}\}$. There are likewise $m$ subexpressions of $F$ of the form $\sigma.J_j$ and $F' = \sum_{j=1}^{m} J_j\{\vec{L_j}/\vec{Z_j}\}$. Therefore, $E\{G/X\} \xrightarrow{\sigma} \sum I_i\{G/X\}\{\vec{K_i'}/\vec{Y_i}\} + G'$ and $F\{G/X\} \stackrel{\sigma}{\Rightarrow} \sum J_j\{G/X\}\{\vec{L_j'}/\vec{Y_j}\} + G'$, where $K_i'$ and $L_j'$ have $X$ substituted for $G$. By Lemma 6.5.15 we can rewrite these to $\sum I_i\{\vec{K_i}/\vec{Y_i}\}\{G/X\}$ and $\sum J_j\{\vec{K_j}/\vec{Y_j}\}\{G/X\}$. Finally, from the fact that $\langle \sum I_i\{\vec{K_i}/\vec{Y_i}\}\{G/X\}, \sum J_j\{\vec{K_j}/\vec{Y_j}\}\{G/X\}\rangle \in \mathcal{R}$ and are thus bisimilar, we can apply Lemma 6.6.18 to show that $\sum I_i\{\vec{K_i}/\vec{Y_i}\}\{G/X\} + G' \approxeq \sum J_j\{\vec{K_j}/\vec{Y_j}\}\{G/X\} + G'$ as required.

The converse cases give symmetry. Thus the proof is complete. □

To begin showing that temporal observation congruence is compositional with respect to recursion I first create the following schema for temporal weak bisimulation up to.

**Lemma 6.6.24** *Temporal Weak Bisimulation up to schema*
*If $E \approxeq^u F$ then a relation $\mathcal{S}^{\approxeq}_{E,F} = \{\langle G\{\mu X.E/X\}, G\{\mu X.F/X\}\rangle\}$ is an asymmetric temporal weak bisimulation up to $\approxeq$.*

**Proof.** I assume that $G\{\mu X.E/X\} \xrightarrow{\gamma} H$ and show that the conditions of Definition 6.6.19 are satisfied. Condition (2) follows easily since by the fact that $E \approxeq^u F$ we know that the unguarded variables of $\mu X.E$ and $\mu X.F$ are identical. Therefore if we substitute both for the same variable in $G$, then both results must have the same unguarded variables. Condition (3) follows from the fact that $\Sigma_F \subseteq \Sigma_E$. Whenever $F$ is insistent with $\Sigma_F = \mathscr{T}$ it also follows that $\Sigma_E = \mathscr{T}$. Therefore also $\Sigma^{\Rightarrow}_{\mu X.F} \subseteq \Sigma_{\mu X.E}$, since when

$\Sigma_{\mu X.E} \neq \mathscr{T}$ it follows that $\Sigma^{\Rightarrow}_{\mu X.F} = \Sigma_{\mu X.F}$. Therefore, since also $\Sigma^{\Rightarrow}_{G} \subseteq \Sigma_G$, it follows that $\Sigma^{\Rightarrow}_{G\{\mu X.F/X\}} \subseteq \Sigma^{\Rightarrow}_{G\{\mu X.E/X\}}$.

The remainder of this proof focuses on proof of condition (1). Throughout I use $\mathcal{S}$ as a shorthand for $\mathcal{S}^{\cong}_{E,F}$. I split the inductive proof into three possible transition cases after Norton, based on the summation rules. Specifically, if $G\{\mu X.E/X\} \xrightarrow{\gamma} H$ then one the following cases must be true:

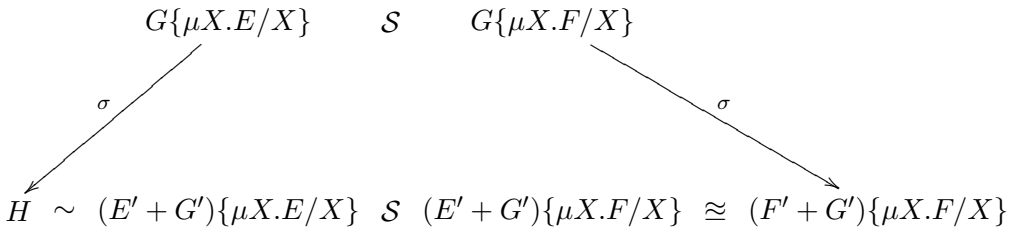
- $G \xrightarrow{\gamma} G'$, and either $\gamma \notin \mathcal{T}_{\mu X.E} \cup \Sigma_{\mu X.E}$ (i.e. the transition is purely from $G$) **or** $G \not\rhd X$, since in the latter none of the $\mu X.E$ transitions are enabled. Either $\gamma = \alpha$ or $\gamma = \sigma$.
  - If $\gamma = \alpha$ then by Lemma 6.5.22 it follows that $G\{\mu X.E/X\} \xrightarrow{\gamma} G'\{\mu X.E/X\}$ with $H \equiv G'\{\mu X.E/X\}$. Therefore also by Lemma 6.5.22 it follows that $G\{\mu X.F/X\} \xrightarrow{\gamma} G'\{\mu X.F/X\}$ and clearly $\langle G'\{\mu X.E/X\}, G'\{\mu X.F/X\}\rangle \in \mathcal{S}$ as required.
  - Alternatively, $\gamma = \sigma$ and either $G \rhd X$ or $G \not\rhd X$. Now, if $G \not\rhd X$ then the proof follows easily through Lemmas 6.5.21 and 6.5.20. Therefore I consider the case when $G \rhd X$. We know that if $X$ is unguarded in $G$ then $\sigma \notin \mathcal{T}_{\mu X.E} \cup \Sigma_{\mu X.E}$. Hence, by Lemma 6.5.23 it follows that $G\{\mu X.E/X\} \xrightarrow{\sigma} G'\{\mu X.E/X\}$. The goal is to show that $G\{\mu X.F/X\} \stackrel{\sigma}{\Rightarrow} G'\{\mu X.F/X\}$. Now, since $E \cong^u F$ and $\sigma \notin \mathcal{T}_E$, it follows that $\sigma \notin \mathcal{T}_F$ by Lemma 6.5.6. Furthermore, since $\sigma \notin \Sigma_E$ and $\Sigma_F \subseteq \Sigma_E$ it follows that $\sigma \notin \Sigma_F$. Therefore, by Lemma 6.5.23 it follows that $G\{\mu X.F/X\} \xrightarrow{\sigma} G'\{\mu X.F/X\}$ and clearly $\langle G'\{\mu X.E/X\}, G'\{\mu X.F/X\}\rangle \in \mathcal{S}$ as required.
- $\mu X.E \xrightarrow{\gamma} J$ (with $G \rhd X$) and $\gamma \notin \mathcal{T}_G \cup \Sigma_G$. Then by Lemma 6.5.16 it follows that $\exists E'.E \xrightarrow{\gamma} E'$ and $J = E'\{\mu X.E/X\}$. Furthermore by Lemma 6.5.24 it follows that $G\{\mu X.E/X\} \xrightarrow{\gamma} J$ and thus $H = J$. Also since $E \cong^u F$, it follows that $\exists F'.F \stackrel{\alpha}{\Rightarrow} F'$ with $E' \cong F'$. Therefore by either Lemma 6.5.22 or 6.5.23 (depending on whether $\gamma = \alpha$ or $\gamma = \sigma$) it follows that $F\{\mu X.F/X\} \stackrel{\gamma}{\Rightarrow} F'\{\mu X.F/X\}$. Furthermore by Lemma 6.5.24 it follows that $G\{\mu X.F/X\} \xrightarrow{\gamma} F'\{\mu X.F/X\}$. By Lemma 6.6.23 it also follows that $F'\{\mu X.F/X\} \cong E'\{\mu X.F/X\}$ (i.e. $E'$ with the same substitution made). Then since $\langle E'\{\mu X.E/X\}, E'\{\mu X.F/X\}\rangle \in \mathcal{S}$ we can fill in the following diagram (illustrating Definition 6.6.19) :

$$\begin{array}{ccccc} G\{\mu X.E/X\} & \mathcal{S} & G\{\mu X.F/X\} & & \\ \downarrow{\scriptstyle\gamma} & & & \searrow\!\!\searrow{\scriptstyle\gamma} & \\ E'\{\mu X.E/X\} & \mathcal{S} & E'\{\mu X.F/X\} & \cong & F'\{\mu X.F/X\} \end{array}$$

This demonstrates that $\langle E'\{\mu X.E/X\},\ F'\{\mu X.F/X\}\rangle \in {\sim}\mathcal{S}{\cong}$ and therefore that

$\langle G\{\mu X.E/X\}, G\{\mu X.F/X\}\rangle \in \mathcal{S}$ as required.

- $G \xrightarrow{\sigma} G'$ and $\mu X.E \xrightarrow{\sigma} J$, with $\gamma = \sigma$ and $G \rhd X$. By Lemma 6.5.16 it follows that $J = E'\{\mu X.E/X\}$ with $E \xrightarrow{\sigma} E'$. Then it follows that $X$ is a subexpression of $G$ and hence $H \sim (F' + G')\{\mu X.E/X\}$ by Lemma 6.6.3 (i.e. modulo the order and recurrence of the resulting sub-terms). Likewise by Definition 6.6.17 it follows that $\mu X.F \xrightarrow{\sigma} K$ with $J \approx^u K$. Then also $G\{\mu X.F/X\} \xrightarrow{\sigma} I$ with $I \sim (F' + G')\{\mu X.F/X\}$. We can now apply a similar logic to the previous case. Firstly notice that $E' \approx^u F'$. Therefore, by Lemma 6.6.18 it follows that $E' + G' \approx^u F' + G'$, and hence also $E' + G' \cong F' + G'$. Therefore also by Lemma 6.6.23 if follows that $(E' + G')\{\mu X.F/X\} \cong (F' + G')\{\mu X.F/X\}$. This allows us to fill in the following diagram:




Which demonstrates that $\langle (E' + G')\{\mu X.E/X\}, (F' + G')\{\mu X.F/X\}\rangle \in \sim\mathcal{S}\cong$ and therefore that $\langle G\{\mu X.E/X\}, G\{\mu X.F/X\}\rangle \in \mathcal{S}$ as required.

The converse cases give symmetry. Thus the proof is complete. □

Finally we can build a schema similar to the weak bisimulation up to schema for temporal observation congruence up to. This will conclude the proof that Temporal Observation Congruence is indeed a congruence relation by demonstrating that if $E \approx^u F$ then $\mu X.E \approx^u \mu X.F$. We do this by taking $G = X$ in the following schema:

**Lemma 6.6.25** ***Temporal Observation Congruence up to schema***
*If $E \approx^u F$ then a relation $\mathcal{S}^{\approx^u}_{E,F} = \{\langle G\{\mu X.E/X\}, G\{\mu X.F/X\}\rangle\}$ is an asymmetric temporal observation congruence up to $\approx^u$.*

**Proof.** I assume that $G\{\mu X.E/X\} \xrightarrow{\gamma} H$ and show that the conditions of Definition 6.6.22 are satisfied. Condition (3) follows easily since by the fact that $E \approx^u F$ we know that the unguarded variables of $\mu X.E$ and $\mu X.F$ are identical. Therefore if we substitute both in the same variable in $G$, then both results must have the same unguarded variables. Condition (4) follows from the fact that $\Sigma_F \subseteq \Sigma_E$. Whenever $F$ is insistent with $\Sigma_F = \mathscr{T}$ it also follows that $\Sigma_E = \mathscr{T}$. Therefore also $\Sigma_{\mu X.F} \subseteq \Sigma_{\mu X.E}$ as required.

The remainder of this proof focuses on proof of conditions (1) and (2). Throughout I use $\mathcal{S}$ as a shorthand for $\mathcal{S}^{\approx^u}_{E,F}$. I split the inductive proof into three possible transition

cases after Norton, based on the summation rules. Specifically, if $G\{\mu X.E/X\} \xrightarrow{\gamma} H$ then one the following cases must be true:

- $G \xrightarrow{\gamma} G'$. Follows an identical proof to this case in 6.6.24.
- $\mu X.E \xrightarrow{\gamma} J$. This follows a very similar proof to the equivalent case in 6.6.24. The difference comes when $\gamma = \alpha$, because at this point we need to show that the resulting expressions are in $\mathcal{S}^{\cong}_{E,F}$. This of course follows easily through the same proof.
- $G \xrightarrow{\sigma} G'$ and $\mu X.E \xrightarrow{\sigma} J$. Again, this follows the same proof as the equivalent case in Lemma 6.6.24.

The converse cases give symmetry. Thus the proof is complete. □

To sum up, I state the Theorem which is one of the main results of this Chapter.

**Theorem 6.6.26** ***Temporal Observation Congruence*** $\approxeq^u$ ***is a congruence relation.***

**Proof.** By combination of Theorem 6.6.16 with Lemmas 6.6.18 and 6.6.25. □

Now that I have shown that $\approxeq^u$ is indeed a congruence with respect to all operators of CaSE$^{\text{ip}}$, it is necessary to show it is the *largest* congruence in $\approxeq$. That is, we must show that $\approxeq^u \equiv \approxeq^+$. I first recall the following fact from universal algebra (borrowed from (Lüttgen, 1998)):

**Proposition 6.6.27** ***Largest Congruence***
*Let $X$ be an equivalence over an algebra $\mathfrak{R}$. Then the largest congruence $X^+$ in $X$ exists and $X^+ = \{\langle P, Q\rangle\} | \forall \mathfrak{R}$-contexts $C[X].\langle C[P], C[Q]\rangle \in X\}$, where a $\mathfrak{R}$-context $C[X]$ is a $\mathfrak{R}$ term with one free occurrence of the variable $X$.*

We can show that $\approxeq^u \equiv \approxeq^+$ by proving the two inclusions $\approxeq^u \subseteq \approxeq^+$ and $\approxeq^+ \subseteq \approxeq^u$. We have already shown in Theorem 6.6.26 that $\approxeq^u \subseteq \approxeq^+$ since we know that $\approxeq^u$ is a congruence and is therefore no larger than the largest congruence. As a result all that remains is to prove the latter inclusion, $\approxeq^+ \subseteq \approxeq^u$. We must first characterise the difference between $\approxeq^+$ and $\approxeq$. Since $\approxeq$ is a congruence for every operator except $+$, we can define an auxiliary relation:

$$\approxeq_a \;\triangleq\; \{\langle E, F\rangle \mid E + \mathcal{G}(E,F) \approxeq F + \mathcal{G}(E,F)\}$$

$$\text{where } \mathcal{G}(E,F) \;\triangleq\; \mu X.\left(\left(\sum_{\sigma \in \mathcal{T}_{E+F}} \sigma.X\right) + c.\mathbf{0}\right) \text{ and } c \text{ is fresh in } E \text{ and } F.$$

This process discriminates the cases when two weakly bisimilar processes are distinguishable. It follows that $\approx^{+} \subseteq \approx_a$ since the definition of $\approx_a$ is less restrictive, it only mandating the equality over a specific class of contexts, whereas $\approx^{+}$ is defined over every context in existence. Therefore, if we can now show that $\approx_a \subseteq \approx^u$ this will prove the remaining inclusion.

**Theorem 6.6.28** *Temporal Observation Congruence is the largest congruence contained in Temporal Weak Bisimulation*

**Proof.** As before, I prove this by showing that each member of the relation $\approx_a$ satisfies the 3 conditions of Definition 6.6.17. In order to demonstrate this it is necessary to show that whenever $E + G \approx F + G$ (where $G$ is defined above) it follows that $E \approx^u F$. We do this by starting with each precondition from the three conditions along with the knowledge that $E + G \approx F + G$, and then form the postcondition.

1. We assume the precondition of the first condition: $E \xrightarrow{\alpha} E'$. Then $E + G \xrightarrow{\alpha} E'$, and therefore since $E + G \approx F + G$ then by Definition 6.6.8, $F + G \stackrel{\hat{\alpha}}{\Rightarrow} H$ with $H \approx E'$. Now, it follows that $H \neq F + G$, since $G \xrightarrow{c}$ which $E'$ cannot match. Because of this, it follows that $F + G \stackrel{\alpha}{\Rightarrow} F'$ with $F \stackrel{\alpha}{\Rightarrow} F'$ and $E' \approx F'$. Thus the first condition is satisfied.

2. Second precondition: $E \xrightarrow{\sigma} E'$. Then $E + G \xrightarrow{\sigma} E' + G$, since $G \xrightarrow{\sigma} G$. Therefore since $E + G \approx F + G$ then by Definition 6.6.8, $F + G \stackrel{\hat{\sigma}}{\Rightarrow} H$ with $H \approx E' + G$. But since $G \xrightarrow{c} G'$ and $c \notin \mathcal{L}(F)$ then $H \equiv F' + G$ with $F \stackrel{\hat{\sigma}}{\Rightarrow} F'$. Therefore, $F \xrightarrow{\sigma} F'$, as otherwise the choice would be resolved. Moreover, since $E' + G \approx F' + G$, it follows that $\langle E', F' \rangle \in \approx_a$, thus satisfying the second condition.

3. Third condition: $\Sigma_F \subseteq \Sigma_E$. If $E$ is not patient the condition holds trivially ($\Sigma_E = \mathscr{T}$). Therefore, we start by assuming $E$ is patient, then $E \stackrel{\tau}{\not\rightarrow}$ and $E + G$ is also patient. Therefore, by Definition 6.6.8, it follows that $\Sigma^{\Rightarrow}_{F+G} \subseteq \Sigma_{E+G}$. But since $\Sigma_G = \emptyset$ it follows that $\Sigma^{\Rightarrow}_{F} = \Sigma^{\Rightarrow}_{F+G}$, since whenever $F + G \stackrel{\tau}{\Rightarrow} H$ then $F \stackrel{\tau}{\Rightarrow} H$. We can also determine that $\nexists F'.F \xrightarrow{\tau} F'$ because then $F + G \xrightarrow{\tau} F'$ and by Definition 6.6.8 it follows that $E + G \approx F'$ which is impossible since $G \xrightarrow{c}$ and $c$ is fresh in $F'$. Since $F \stackrel{\tau}{\not\rightarrow}$ then $\Sigma^{\Rightarrow}_{F} = \Sigma_F$. Furthermore, $\Sigma_{E+G} = \Sigma_E$ and therefore and the postcondition $\Sigma_F \subseteq \Sigma_E$ follows.

The converse follows by symmetry. Thus the proof is complete. □

An alternative characterisation of weak bisimulation which uses the standard CCS definition can be formed by creating an enriched derived transition relation. The standard definition is more convenient to use in many instances since standard partition refinement algorithms can then be used to decide weak bisimulation. This relation makes patient clock ticks explicit again, so that they can simply be matched as usual by weak

bisimulation. It also assumes that $\mathcal{T}_E$ is non-empty, because otherwise $\Delta \approx \mathbf{0}$ holds, which isn't true for our original definition. Regular active clock ticks over $\sigma$ are represented by a $\sigma_1$ transition, whilst patient ticks are represented by a $\sigma_0$, as defined below:

**Definition 6.6.29** ***Explicit Patience Derived Transition Relation***
*Given a CaSE$^{ip}$ LTS $(\mathcal{E}, \mathscr{A} \cup \mathscr{T}, \rightarrow)$, there is an Explicit Patience LTS $(\mathcal{E}, \mathscr{A} \cup \mathscr{T}_0 \cup \mathscr{T}_1, \rightarrow_u)$, where $\mathscr{T}_n = \{\sigma_n | \sigma \in \mathscr{T}\}$. The transition relation $\rightarrow_u$ is defined thus:*

$$\begin{array}{lcl} E \xrightarrow{\sigma_0}_u E & \iff & \sigma \notin (\Sigma_E \cup \mathcal{T}_E) \wedge \tau \notin \mathscr{A}_E \\ E \xrightarrow{\sigma_1}_u E' & \iff & E \xrightarrow{\sigma} E' \\ E \xrightarrow{\alpha}_u E' & \iff & E \xrightarrow{\alpha} E' \end{array}$$

The associated weak transition relation $\Rightarrow_u$ is as given in Definition 2.3.2 (Chapter 2). The standard definition of weak bisimulation can then be applied.

**Definition 6.6.30** ***Temporal Weak Bisimulation (t.w.b.)***
*A symmetric relation $\mathcal{R}$ is a t.w.b. provided $\forall \langle E, F \rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{\gamma}_u E'$ then $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow}_u F'$ and $\langle E', F' \rangle \in \mathcal{R}$*

*We write $E \approx F$ if $\exists R.\langle E, F \rangle \in R$ and $R$ is a t.w.b.*

The derived definition of Temporal Weak Bisimulation can be shown to be equivalent to Temporal Weak Bisimulation with Explicit Urgency:

**Theorem 6.6.31** ***Every Temporal Weak Bisimulation is a t.w.b.e.u.***

**Proof.** Assume the existence of a t.w.b. $\mathcal{R}$. We must show that $\mathcal{R}$ is also a t.w.b.e.u. To this end we choose a pair $\langle E, F \rangle \in \mathcal{R}$, assume the precondition of each condition of Definition 6.6.8 for the given pairing, and finally show that Definitions 6.6.29 and 6.6.30 entail the postcondition.

- Condition (1) – show that if $E \xrightarrow{\gamma} E'$ then $\exists F'.F \stackrel{\hat{\gamma}}{\Rightarrow} F' \wedge \langle E', F' \rangle \in \mathcal{R}$.
  - **Option 1** : $\gamma = \alpha$. Then by Definition 6.6.29, $E \xrightarrow{\alpha}_u E'$ and thus by Definition 6.6.30, $\exists F'.(F \stackrel{\alpha}{\Rightarrow}_u F' \wedge \langle E', F' \rangle \in \mathcal{R})$ as required.
  - **Option 2** : $\gamma = \sigma$. Follows the same proof as $\gamma = \alpha$.
- Condition (2) – Show that if $\Sigma_E \subset \mathscr{T}$ then $\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$
  - Suppose that $\Sigma_E \subset \mathscr{T}$
  - It follows that there is at least one $\sigma$ such that $\sigma \notin \Sigma_E$
  - To show that $\Sigma_F^{\Rightarrow} \subseteq \Sigma_E$ it suffices to show, for any such $\sigma$, if $\sigma \notin \Sigma_E$ then $\sigma \notin \Sigma_F^{\Rightarrow}$

- It follows from $\sigma \notin \Sigma_E$ that by Definition 6.6.29 that $E \xrightarrow{\sigma_0}_u E$
- Therefore by Definition 6.6.30 it follows that $\exists F'.(F \stackrel{\hat{\sigma_0}}{\Rightarrow}_u F' \wedge \langle E, F' \rangle \in \mathcal{R})$
- Furthermore, by Definition 2.3.2, $\exists F'' F'''.F \xrightarrow{\tau}_u{}^* F'' \xrightarrow{\sigma_0}_u F''' \xrightarrow{\tau}_u{}^* F'$
- But by Definition 6.6.29 we know that it must follow that $F'' \equiv F'''$
- Since $F''$ is patient, it follows that $F''' \equiv F'$ and therefore $F \stackrel{\hat{\tau}}{\Rightarrow}_u F'$
- By Definition 6.6.29 it follows that $F \stackrel{\hat{\tau}}{\Rightarrow} F' \wedge \sigma \notin \Sigma_{F'}$
- Therefore it also follows that $\sigma \notin \Sigma_F^{\Rightarrow}$ as required.

The converse cases follow by symmetry. □

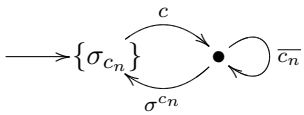
The disadvantage of converting to an explicit patience transition system is that, inevitably, a certain amount of data is lost, since it is necessary to fix $\mathscr{T}$ to a specific finite set to derive the self-transitions. It means composition with this sort of transition system is not possible, since if a $\sigma_0$ transition is not present then that clock is stalled. Therefore, whilst being absolutely necessary for use with standard verification algorithms based on the standard weak bisimulation definition, it cannot replace the set theoretic semantics entirely.

## 6.7 Timed Transition Systems

I highlighted at the end of the previous Section a deficiency in CaSE$^{\text{ip}}$ transitions that follows for multiple-clock timed process calculi generally, particularly those with patience. When a labelled transition system is produced for a CCS process all the data needed for composition is contained within the transitions. Effectively the process labels can be ignored for the purpose of composing the transition system with another CCS process since every CCS LTS is isomorphic to a sequential CCS process. However, this is not the case with CaSE and CaSE$^{\text{ip}}$, as the creation of a clock transition depends upon whether the process syntax stalls any clocks. To reiterate, though the process $\Delta$ has no transitions it does *not* suffice to simply represent it a single-state LTS as this would then be considered equivalence with **0**.

It is possible, as proved in Lemma 6.6.31, to represent patient clocks as self-transitions like CaSE does, but only if the $\mathscr{T}$ sort is collapsed to a finite clock universe. Clearly though for component systems which use as many clocks as there are components, it should be possible to represent a process accurately in a transition system by repeatedly collapsing and expanding $\mathscr{T}$. Although the derived transition relation in the previous section provides a nearest approximation of CaSE$^{\text{ip}}$ in a standard LTS, what is really needed is an expanded form of LTS.

In approaching the same problem, CaSE has the notion of a *symbolic transition system*, which stores the collection of stalled clocks at the states. This is illustrated well in Norton and Fairtlough (2004), where transition systems like the following are utilised:




This is a simple isochronic broadcast agent which inputs a value on $c$ (whilst holding up $\sigma^{c_n}$) and then repeatedly outputs $c_n$ until $\sigma^{c_n}$ ticks. Different classes of states are symbolised differently:

- States holding up several clocks (e.g. $\Delta_\sigma$) are annotated with elements of the set of clocks $\Sigma_P$;
- States holding up all clocks other than those with outgoing transitions are represented by a bullet $\bullet$;
- States patient on all clocks are represented by a bullseye state $\circledcirc$.

The second case is not entirely possible with CaSE$^{\text{ip}}$ as there is no way of representing a globally insistent clock prefix, only one insistent on several clocks. Therefore to represent a transition system in CaSE$^{\text{ip}}$ it is adequate to represent the states as the set of clocks in $\Sigma_P$. If empty, the process is fully patient. Additionally the set must have a "full" or *top* state, which represents that all clocks in $\mathscr{T}$ are held up. By using this form of transition system it is possible to accurately represent the equivalence class of each process. The resulting LTS would therefore be something like $(\mathcal{L}, \mathscr{A} \cup \mathscr{T}, \rightarrow_t)$ with an associated mapping relation $\mathcal{L} \mapsto \tilde{\mathscr{T}}$ which associates the set of stalled clocks with each state label.

Additionally the weak transition relation $\Rightarrow$ gives rise to a weak timed transition system. Each weak state in the weak transition system should be annotated with $\Sigma_P^{\Rightarrow}$, the set of weakly stalled clocks.

This fact will prove vital for producing an efficient implement of CaSE$^{\text{ip}}$ since it is neccessary to represent the state of a process *minimally*. Whilst the process syntax itself provides all the information required, it is inefficient to be constantly regenerating the set $\Sigma_P$ and hence it is important to represent the minimal amount of data needed to represent a timed processes state. This allows a programmer to shortcut the process syntax and instead represent the process in its entirety as a graph.

## 6.8 Refinement Theory

In this final section, my aim is the definition of a preorder which will be used as a refinement or subtyping relation for Web service choreographies. Due to time constraints, this is only proved for CCS so far, but it demonstrates the foundational concept. This is not an immediately important issue, as clocks in the Cashew-A semantics will only be used internally within orchestrations, and not in choreography descriptions. The relation is formulated below:

**Definition 6.8.1** ***Alternating Simulation***
*A relation $\mathcal{R}$ is an Alternating Simulation provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{a} E'$ and $a \in \Lambda$ then $\exists F'.F \stackrel{a}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $F \xrightarrow{\overline{a}} F'$ and $\overline{a} \in \overline{\Lambda}$ then $\exists E'.E \stackrel{\overline{a}}{\Rightarrow} E'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $E \xrightarrow{\tau} E'$ then $\exists F'.F \stackrel{\epsilon}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $F \xrightarrow{\tau} F'$ then $\exists E'.E \stackrel{\epsilon}{\Rightarrow} E'$ and $\langle E', F'\rangle \in \mathcal{R}$.*

*We write $E \lessapprox F$ if $\langle E, F\rangle \in \mathcal{R}$ for any Alternating Simulation $\mathcal{R}$.*

*Alternating Simulation* is adapted from de Alfaro and Henzinger (2001) and Norton and Fairtlough (2004). It is a behavioural relationship which describes when one component adequately satisfies a context. Specifically, it requires that the component provides at least as many inputs required (covariance), and no more outputs than the environment can accept (contravariance). This must be true in every state of a component's interaction. It is useful for ensuring *conformance* between a given template choreography entailed by an orchestration, and a concrete Web service choreography.

I will prove some basic propositions with respect to our basic equivalence, weak bisimulation, namely that Alternating Simulation is a partial order. I first prove the following useful lemma:

**Lemma 6.8.2** ***If $\mathcal{S}_1$ and $\mathcal{S}_2$ are Alternating Simulations then so is $\mathcal{S}_1 \cdot \mathcal{S}_2$***

**Proof.** To show this is true, it suffices to show that for any $\langle P, R\rangle \in \mathcal{S}_1 \cdot \mathcal{S}_2$, it follows that $P \lessapprox R$. Specifically, that both conditions of Definition 6.8.1 hold. It follows that there is a $Q$ such that $\langle P, Q\rangle \in \mathcal{S}_1$ and $\langle Q, R\rangle \in \mathcal{S}_2$. I will use this to demonstrate that the four conditions of Definition 6.8.1 hold.

- If $P \xrightarrow{a} P'$ and $a \in \Lambda$ then $\exists R'.R \stackrel{a}{\Rightarrow} R'$ with $\langle P', R'\rangle \in \mathcal{S}_1 \cdot \mathcal{S}_2$
  - Suppose that $P \xrightarrow{a} P'$ and $a \in \Lambda$
  - By Definition 6.8.1, $Q \stackrel{a}{\Rightarrow} Q'$
  - Specifically, $\exists Q'Q''Q'''.Q \stackrel{\epsilon}{\Rightarrow} Q'' \xrightarrow{a} Q''' \stackrel{\epsilon}{\Rightarrow} Q'$, with $\langle P', Q'\rangle \in \mathcal{S}_1$.
  - Therefore $R \stackrel{\epsilon}{\Rightarrow} R''$ with $\langle Q'', R''\rangle \in \mathcal{S}_2$
  - Furthermore, since $Q'' \xrightarrow{a} Q'''$ and $a \in \Lambda$ then $\exists R'''.R'' \stackrel{a}{\Rightarrow} R'''$ with $\langle Q''', R'''\rangle \in \mathcal{S}_2$
  - And, since $Q''' \stackrel{\epsilon}{\Rightarrow} Q'$ then $R''' \stackrel{\epsilon}{\Rightarrow} R'$ with $\langle Q', R'\rangle \in \mathcal{S}_2$
  - Therefore, $\exists R'.R \stackrel{a}{\Rightarrow} R'$ with $\langle P', R'\rangle \in \mathcal{S}_1 \cdot \mathcal{S}_2$ as required.
- If $R \xrightarrow{\overline{a}} R'$ and $\overline{a} \in \Lambda$ then $\exists P'.P \stackrel{\overline{a}}{\Rightarrow} P'$ with $\langle P', R'\rangle \in \mathcal{S}_1 \cdot \mathcal{S}_2$

  - Similar to the above, but taking the path, $R \xrightarrow{\overline{a}} R'$ to $Q \stackrel{\overline{a}}{\Rightarrow} Q'$ to $R \stackrel{\overline{a}}{\Rightarrow} R'$.

- The two remaining $\tau$ conditions follow the same respective structures as the two above.

Thus it follows that $\mathcal{S}_1 \cdot \mathcal{S}_2$ satisfies the conditions for Alternating Simulation. $\square$

With this lemma, I can now prove that Alternating Simulation is a preorder.

**Lemma 6.8.3** *Alternating Simulation is a preorder*

**Proof.** To show that alternating simulation is a preorder we need to show that it is reflexive and transitive.

- **Reflexivity** – it suffices to note that the identify $Id_{\mathcal{E}}$ is an alternating simulation.

- **Transitivity** – consider three processes $P, Q$ and $R$ with $P \lessapprox Q$ and $Q \lessapprox R$. Then there are two alternating simulations $\mathcal{S}_1$ and $\mathcal{S}_2$ with $\langle P, Q\rangle \in \mathcal{S}_1$ and $\langle Q, R\rangle \in \mathcal{S}_2$. By Lemma 6.8.2 it follows that $\mathcal{S}_1 \cdot \mathcal{S}_2$ is an alternating simulation and therefore transitivity holds.

Thus alternating simulation is a preorder. $\square$

**Proposition 6.8.4** *Alternating Simulation is antisymmetric*

**Proof.** To show that alternating simulation is antisymmetric consider two processes $P$ and $Q$ with $P \lessapprox Q$ and $Q \lessapprox P$. We need to show that it follows that $P \approx Q$. There are two alternating simulations $\mathcal{R}$ and $\mathcal{S}$ with $\langle P, Q\rangle \in \mathcal{R}$ and $\langle Q, P\rangle \in \mathcal{S}$. Therefore, it suffices to show that $\mathcal{R} \cup \mathcal{S}^{-1}$ is a weak bisimulation. We do this by inducting over all possible transitions $P$ may perform to prove our relation satisfies the conditions of weak bisimulation.

- From $P \xrightarrow{\alpha} P'$ we need to show that $\exists Q'.Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ with $\langle P', Q'\rangle \in \mathcal{R} \cup \mathcal{S}^{-1}$. This breaks down into three possibilities:

  - $P \xrightarrow{a} P'$ with $a \in \Lambda$. Using $\mathcal{R}$ with the first condition of Definition 6.8.1 it follows that $\exists Q'.Q \stackrel{a}{\Rightarrow} Q'$ with $\langle P', Q'\rangle \in \mathcal{R}$. Furthermore $\langle P', Q'\rangle \in \mathcal{R} \cup \mathcal{S}^{-1}$, which satisfies the definition of weak bisimulation.
  - $P \xrightarrow{\overline{a}} P'$ with $\overline{a} \in \overline{\Lambda}$. Using $\mathcal{S}$ with the second condition of Definition 6.8.1 it follows that $\exists Q'.Q \stackrel{a}{\Rightarrow} Q'$ with $\langle Q', P'\rangle \in \mathcal{S}$. Furthermore $\langle P', Q'\rangle \in \mathcal{R} \cup \mathcal{S}^{-1}$, which satisfies the definition of weak bisimulation.
  - $P \xrightarrow{\tau} P'$. Using $\mathcal{R}$ with the third condition of Definition 6.8.1 it follows that $\exists Q'.Q \stackrel{\epsilon}{\Rightarrow} Q'$ with $\langle P', Q'\rangle \in \mathcal{R}$. Furthermore $\langle P', Q'\rangle \in \mathcal{R} \cup \mathcal{S}^{-1}$, which satisfies the definition of weak bisimulation.

- The converse case gives symmetry.

Thus we have proved that Alternating Simulation is a antisymmetric. □

Thus, based on the fact that Alternating Simulation is both a preorder and antisymmetric we now have the basis for the following theorem:

**Theorem 6.8.5** ***Alternating Simulation is a partial order*** □

Finally, although Alternating Simulation is a partial order, it is not a precongruence relation, for the same reason that weak bisimulation is not a congruence for CCS. Therefore, I define the following precongruence:

**Definition 6.8.6** ***Alternating Precongruence*** *A relation $\mathcal{R}$ is an Alternating Precongruence provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{a} E'$ and $a \in \Lambda$ then $\exists F'.F \stackrel{a}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $F \xrightarrow{\overline{a}} F'$ and $\overline{a} \in \overline{\Lambda}$ then $\exists E'.E \stackrel{\overline{a}}{\Rightarrow} E'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $E \xrightarrow{\tau} E'$ then $\exists F'.F \stackrel{\tau}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $F \xrightarrow{\tau} F'$ then $\exists E'.E \stackrel{\tau}{\Rightarrow} E'$ and $\langle E', F'\rangle \in \mathcal{R}$.*

*We write $E \leq F$ if $\langle E, F\rangle \in \mathcal{R}$ for some Alternating Simulation $\mathcal{R}$.*

Alternating Precongruence differs from Alternating Simulation in only one respect – once again any initial $\tau$ must be matched by at least one $\tau$.

## 6.9 Conclusion

In this Chapter I have taken an in-depth look into the core theoretical features required for modelling Web service based component systems. I started out from the timed process calculus CaSE and investigated some of the potential problem areas for defining a compositional semantics for service composition. The culmination of this was the definition of three process calculi. The first, an extension of CCS with support for localised interruption, had several theoretical problems, notably an inadequate definition of observation equivalence. This led to the realisation that abstract time itself is an ideal paradigm for expressing interrupting component systems.

I then proceeded to examine timed process calculus in general, and found that CaSE has a number of problems and abnormalities relating to its handling of patient clock ticks. As a result of this I sought a new and more flexible approach. The first attempt at this, $\mathsf{CaSE}^{\mathrm{mt}}$, is very flexible but introduces too much complexity in defining different patterns of patience between clocks. Therefore the successful process calculus, $\mathsf{CaSE}^{\mathrm{ip}}$, was developed using a new style of semantics based on *instability sets*. The new calculus possesses a distinct formulation of patience, which is one of three states a clock can be

in. This allow some of the existing problems, such as patient clock prefix, to be solved in a more elegant fashion, whilst adding new possibilities such as timed choice.

As a further addition to this calculus I added a new *clock renaming* operator, which allows clocks to be realised as standard CCS actions. This provides both a more flexible approach to clock scoping, in that the implicit total order introduced by hiding can be avoided when required, and allows further synchronisation patterns , such as a one-to-many synchronisation between a synchronous agent and a temporal region. All in all, CaSE$^{ip}$ contributes a significant increment from its predecessor, CaSE, whilst still remaining faithful to the latter's concept of abstract time.

The majority of the time in this Chapter has been spent on adapting a weak bisimulation semantics to CaSE$^{ip}$. Although the formulation is very similar to that in CaSE it has provided a number of issues which have been dealt with. In particular CaSE$^{ip}$ formalises the notion of a clock being patient only after a process stabilises. I have proven that my new equivalence theory, *Temporal Observation Congruence*, is the largest equivalence within temporal weak bisimulation, and also that it can be reformulated to standard weak bisimulation on an enriched labelled transition system. As a short addendum, I also considered *alternating simulation* and how it provides an ideal semantic theory for describing compatibility between behavioural component interfaces, such as a Web service choreography.

Although the work in this Chapter is significant, it is still in its infancy. I do not claim that CaSE$^{ip}$ solves all the problems which CaSE can, and indeed it can be argued that CaSE's simplification of patience is a strength in certain circumstances rather than a weakness. For instance, weak bisimulation is more directly adapted in CaSE than in CaSE$^{ip}$ . Nevertheless it is certainly a step forward in the options it provides for representing timed component systems, and as we shall see in the remaining Chapters provides a better foundation for giving a semantics to service composition.

# Chapter 7

# A Compositional Operational Semantics for Cashew-A

*In this Chapter I will give an Operational Semantics to **Cashew-A** in the form of a CaSE$^{ip}$ denotation, the process calculus introduced in Chapter 6. At the core of this denotation are a number of **protocols** which allow components of a workflow to share the same meta-state and negotiate permission to execute from the environment. These protocols provide an approach to semantics which means that different parts of the system are maximally decoupled, and can thus evolve independently. The semantic framework is therefore both **extensible** and **compositional**. I first describe the regular part of the language, in terms of its protocols, data flow and control flow. I then describe an experimental compensable fragment of the language and show that my semantic approach favours a flexible compensatory strategy.*

## 7.1 Overview

HAVING defined the operational calculus CaSE$^{ip}$ in Chapter 6, I now proceed to use it to give an operational semantics to Cashew-A . I will seek to justify the choice of calculi by demonstrating that abstract time is an essential ingredient in describing workflows with explicit preconditions, postconditions, and eventually, compensation.

Cashew-A is given a behavioural semantics though a number of *schedulers*, each of which is composed in parallel with one or more *actor processes*. Each actor process plays the part of a fragment of the workflow and in this way a complete director model can be inductively acquired, an approach adapted from Norton et al. (2005). For instance, the semantics of a Cashew-A process like $[\![P ; Q]\!]$ will be described (broadly) as $[\![;]\!] \mid [\![P]\!] \mid [\![Q]\!]$, that is the semantics of the two processes parallel composed with the $;$ (sequential) scheduler. Each scheduler will coerce the sub-components into a specific *synchronisation pattern* using its various channels, resulting in a particular execution scheme (such as

sequential, choice, parallel etc.). The details of these channels and how they synchronise will be given throughout this chapter.

The semantics will be given formally using a series of semantic relations which will map each structure of Cashew-A to a CaSE$^{\text{ip}}$ process. In general, the signature of this relation is defined thus:

$$\_[\![-]\!]^{\_}_{\_,\_} : \mathsf{Constructor} \times (\mathsf{Name} \times \mathbb{N}) \times \mathsf{Expression} \times (\tilde{\mathbb{A}} \times \tilde{\mathbb{B}}) \times (\tilde{\mathbb{M}} \times \tilde{\mathbb{M}}) \to \mathcal{E}$$

The sorts $\mathbb{W}, \mathbb{A}, \mathbb{B}, \mathbb{M}, \mathbb{W}$ and $\mathbb{P}$ were defined in Chapter 5 Section 5.1. A Constructor is simply a 2 or 3 letter code which distinguishes the context of a particular construct. For instance, the semantics of $P \fatsemi Q$ are different when in a normal workflow to when in a compensable workflow. Thus I distinguish these two cases by the use of constructors **wf** and **cwf**, respectively. A Name is the name of the structure being given a semantics, either a performance or a workflow, thus $\mathsf{Name} = \mathbb{P} \cup \mathbb{W}$. Each name is associated with a granularity given by a natural number, which represents the granularity of the internal RTC in basic units. An Expression is any expression in the Cashew-A language to which we are giving a semantics. Each construct will also possess a set of inputs and a set of outputs (top-right), taken from $\tilde{\mathbb{A}} \times \tilde{\mathbb{B}}$. Finally, each construct also possesses a set of input messages and a set of output messages (bottom-right), taken from $\tilde{\mathbb{M}} \times \tilde{\mathbb{M}}$.

Thus ${}^{\mathbf{wf}}_{w,g}[\![P \fatsemi Q]\!]^{\tilde{a}^P,\tilde{b}^P}_{\tilde{m},\tilde{n}}$ means *the semantics of* $P \fatsemi Q$ *with input set* $\tilde{a}^P$*, output set* $\tilde{b}^P$*, input messages* $\tilde{m}$*, output messages* $\tilde{n}$*, name* $w$ *and granularity* $g$*, under the* **wf** *constructor*. We now proceed to describe the semantics and thus populate this relation, first for normal workflows and then for compensable workflows.

At the core of all of these semantics is a *scheduling protocol* and clock *phase transition system*, which all parts of a component adhere to. The advantage of this approach is that the semantics are directly extensible, and so long as a new construct's semantics adheres to the protocols it will fit into the existing framework. This is important because as I illustrated there are several possibilities for the Cashew-A control flow language, and expansion with additional constructs may be required in the future. The protocol is effectively a conversation between the component and its environment. By contrast, the phase transition system allows all the subcomponents of a workflow to share a common state.

**A note about the syntax of these semantics**. An italicised letter sequence, e.g. $a$, is a *variable*. A bold letter sequence, e.g. $\mathbf{a}$, is a *value*. Variables can have both a subscript and a superscript applied. A superscript means that the given superscript valuation is applied to the underlying value of the variable, so for instance, if $a = \mathbf{input}$ and $w = \mathbf{wfname}$, then $a^w = \mathbf{input}^{\mathbf{wfname}}$. In contrast, a subscript is simply a decoration of an existing variable name, creating a new distinct variable name, for instance $a_{\mathbf{i}}$ is a distinct variable from $a$. A subscript decoration has no effect on the underlying variable valuation. Finally, a tilde over a variable name produces a distinct variable name, but this time it is a set. For instance, if we have $\tilde{a}$, this is a variable distinguished from $a$, with

a set valuation. These rules do not apply to clock names, which are different. Lower case variables like $p$ are names or atomic values of some sort, whereas upper case variables like $P$ range over part of the Cashew-A syntax.

## 7.2 Normal Workflow Semantics

### 7.2.1 Phases

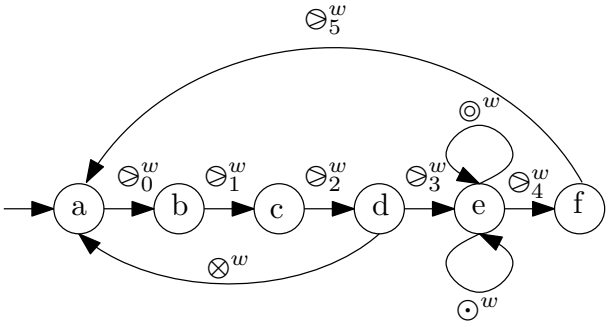



Figure 7.1: Phases of a normal workflow

All of the workflow components follow a common phase transition system built using a number of distinct clock sets per workflow. The phase transition systems enable the components of the workflow to simultaneously share the state. When defining the semantics for the process calculus CaSE$^{\text{ip}}$ in Chapter 6 I used greek symbols ($\sigma, \rho \cdots$) to represent clocks. However, since there are now many different clock classes and using these symbols would be confusing, I use the following clock sort:

$$
\begin{aligned}
\mathscr{T} &= \{\circledcirc_n^w \mid w \in \mathbb{W} \wedge n \in \{0 \cdots 5\}\} \cup \{\odot^w \mid w \in \mathbb{W}\} \\
&\cup \{\circledcirc^w \mid w \in \mathbb{W}\} \cup \{\otimes^w \mid w \in \mathbb{W}\} \cup \{\circledast^w \mid w \in \mathbb{W}\} \\
&\cup \{\sigma^m \mid m \in \mathbb{M}\} \cup \{\circledcirc^p \mid p \in \mathbb{P}\} \cup \{\circledcirc_n^p \mid p \in \mathbb{P} \wedge n \in \{1 \cdots 4\}\} \cup \{\sigma, \rho\} \\
&\cup \left\{ \mathbb{O}_p^b \,\middle|\, p \in \mathbb{P} \wedge b \in \mathbb{B} \right\}
\end{aligned}
$$

The complete phase transition system is illustrated in Figure 7.1. In a normal workflow there are a total of 5 phases (a through e), which indicate what the current action of the workflow is. These phases are delimited by what can be called the "normal" workflow clocks, namely $\circledcirc_0^w \cdots \circledcirc_5^w$. All workflows begin in phase a, which is the inactive phase. Here the workflow can perform literally no behaviour, other than receiving an activation signal. The first workflow clock (0) signals when the workflow has been activated, thus moving it into phase b, the state where it can begin to receive its inputs before executing. Since the workflow will be contained within a performances, these inputs will be fed in via the performance inputs buffers. The second clock (1) ticks when sufficient inputs have been received to execute, and at this point the control flow is activated. This

means that, for instance, any initial messages can be obtained at this point. The third clock (2) signals when the workflow has reached the ready phase (d) – it has satisfied sufficient preconditions so as to enable execution. The fourth clock (3) signals that the environment of the workflow has permitted execution to take place and thus the control flow should start executing. It is also possible that permission to execute may be denied by the environment at this point (if another competing process became ready first), and if so the $\otimes^w$ cancellation clock will tick instead to signal that the sub-components should discard their inputs and return to inactivity. During the execution phase (e) the *yield* clock $\odot^w$ and *real-time clock* (RTC) can also repeatedly tick, I will describe the former in more detail when I explain yielding. The RTC ticks once in between each time interval predefined by the workflow head. The fifth clock (4) ticks when execution of the control flow has completed, moving the workflow into the penultimate phase (f), where the outputs are gathered together and passed to the environment. Finally, the sixth clock (5) indicates that all postconditions have been populated and the workflow returns to inactivity (phase a).

The remaining phase clock type which I have not described is $\circledast^w$, which is not used in this phase transition system as it relates to compensation and so will be dealt with in Section 7.3. However, it is important to emphasise that these workflows enforce a specific priority on the clocks using the hiding operator. For instance, in $P/\sigma/\rho$, if $P$ is capable of ticking both $\sigma$ and $\rho$ it is the $\sigma$ transition which will occur first, since it is converted to a $\tau$ lower down, blocking $\rho$. This demonstrates an important property of the clock phases since it removes the possibility of non-determinism between two phases (unless explicitly required). For instance, if the workflow execution is cancelled in phase c, the ticking of $\otimes^w$ takes precedence over $\circledcirc_3^w$. Furthermore, as we shall see, entering the compensation phase is prioritised above yielding.

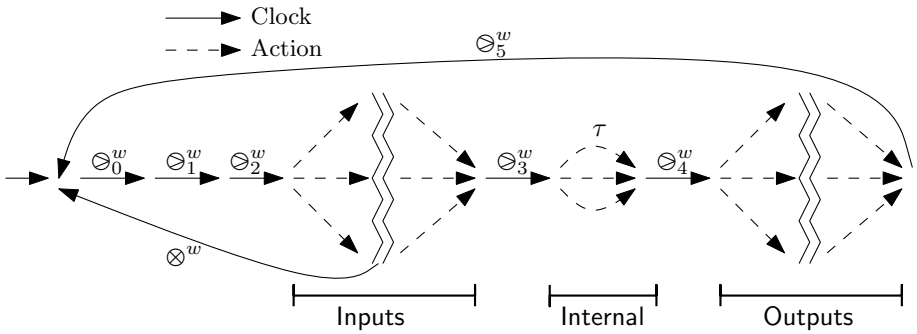



Figure 7.2: Overall workflow timeline (not all clocks and actions shown)

The phase transition system provides a necessary skeleton for the workflow. All agents can tap into it to get information about their peers and act accordingly. It also provides an overall structure which all workflows (and in general all components) adhere to – this is shown in Figure 7.2. In particular, note that for $\circledcirc_3^w$ to tick a path must be found through the input section, signifying the satisfaction of the preconditions.

Nevertheless, the phase transition system does not allow much protocol communication between individual agents, and thus on top of the clock phases there is also a negotiation protocol with which all performances (and thus workflows) must obtain permission to execute from their environment. This is described in the next section.

Before moving on though, there are four other clock types in the sort which are not specifically related to workflow phases. The first $\circledcirc_n^p$ are *performance clocks*, and are used to represent phases in a performance, if required. Since performances are less structured than workflow, these clocks are optional. In addition, there are two *anonymous* clocks $\sigma$ and $\rho$, which are used in contexts where no name is available to decorate them (but see Section 7.2.10 for potential problems). In addition to the performance clocks, there are also the performance data flow clocks, $\circledcirc_p^b$, which are used to broadcast data from perform outputs. These will be explained properly in the performance semantics section.

The remaining clock type is central to message flow. In my semantics, messages are represented intermediately as clocks, e.g. $\sigma^m$. These are then renamed to channels at the workflow head, using CaSE$^{\text{ip}}$'s new clock renaming operator, which I discussed in Chapter 6, Section 6.4. This is to ensure that a message cannot be received whilst there is internal activity taking place in the workflow, which is judged to be faster. If, for instance, there exists a choice between receiving several messages then once one of these messages has been received the remainder must be instantly disabled, so the choreography is correct. If we were to use a regular action this wouldn't be possible as they are not prioritised and therefore simply interleaved if placed in parallel processes. However it is possible to hold up a clock via maximal progress. The *receive* also can only happen if the environment provides a message. Therefore a message receive is a one-to-many decision, it can only occur if both the environment provides the message and all agents in the workflow permit its receipt. Also note that this has the further effect of ensuring that message-less paths through the workflow have higher priority. In other words, if it is possible to complete the workflow without exchanging messages, no messages will ever be exchanged. For instance in (**Receive** m $\emptyset$) $\oplus$ $\epsilon$ it is impossible to receive message $m$, since the skip can always execute immediately thus disallowing $\sigma^m$ from ticking.

### 7.2.2 The Orchestration Protocol

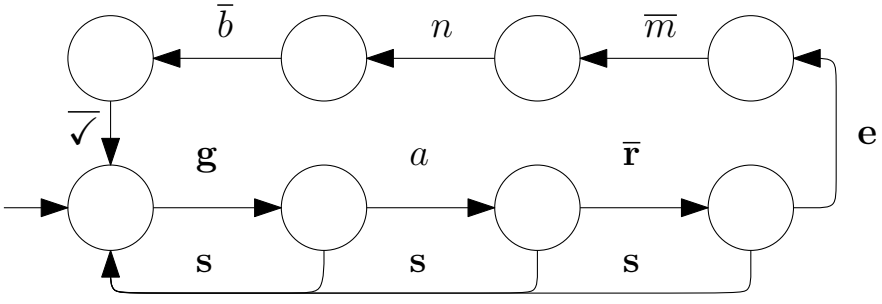



Figure 7.3: A basic Workflow transition system

In addition to adhering to the phase transition system, all components of a workflow follow a common protocol. This ensures that the semantic framework is extensible, in that new workflow constructs can easily be added, provided they adhere to the protocol. The protocol is an extension of the one used in our previous OWL-S semantics (Norton et al., 2005), which is itself an extension of the protocol used by Norton and Fairtlough (2004) for Digital Signal Processor components.

The orchestration protocol uses five channels, $\mathbf{g}, \mathbf{r}, \mathbf{e}, \checkmark$ and $\mathbf{s}$, which are used to instruct the component as to which action it should perform:

- $\mathbf{g}$ stands for *go* and is input at the very beginning of a component's execution. It is essentially an activation signal.
- $\mathbf{r}$ stands for *ready* and is issued by a component once the readiness phase is reached. This is a declaration to the environment that the component can proceed to execute without outside intervention.
- $\mathbf{e}$ stands for *execute* and may be issued to a component by the environment as permission to begin actual execution.
- $\mathbf{s}$ which stands for *stop* and is issued if the environment has alternatively decided the component should not execute, for example if another component has been chosen instead.
- $\mathbf{t}$ stands for *tick* and indicates that the component's internal clock wishes to tick. This is used to mediate between the RTCs of a parent and child component.
- Finally $\checkmark$ is issued by the component at the end of execution, signalling completion and its return to the inactive state.

In the workflow semantics it is usual that these channels will be given superscripts $\mathbf{i}$ and $\mathbf{j}$ to denote whether they refer to the left component or the right component of a binary operator. Furthermore, the $\mathbf{i}$ subscript is sometimes used to refer to the internals, as opposed to the environment. I thus define two renaming sets, $\mathfrak{F}$ and $\mathfrak{G}$, which will be used to distinguish between the channels of internal components being scheduled, and also set $\mathfrak{R}$ which contains all these decorated channels and will be used for restriction. These are defined in terms of $\mathfrak{C}$, which contains all the protocol channels in the language:

$$
\begin{aligned}
\mathfrak{C} &= \{\mathbf{g}, \mathbf{r}, \mathbf{e}, \mathbf{s}, \checkmark, \mathbf{g_c}, \mathbf{r_c}, \mathbf{e_c}, \checkmark_{\mathbf{c}}\} \\
\mathfrak{F} &= \left\{a \mapsto a^{\mathbf{i}} \mid a \in \mathfrak{C}\right\} \\
\mathfrak{G} &= \left\{a \mapsto a^{\mathbf{j}} \mid a \in \mathfrak{C}\right\} \\
\mathfrak{R} &= \left\{a^{\mathbf{i}} \mid a \in \mathfrak{C}\right\} \cup \left\{a^{\mathbf{j}} \mid a \in \mathfrak{C}\right\}
\end{aligned}
$$

This protocol has substantially evolved since our previous work (Norton et al., 2005), where only $\mathbf{r}$ and $\mathbf{e}$ were required. A clock $\sigma$ was used in place of $\checkmark$, which simply used maximal progress to detect completion asynchronously. However, in order to represent the choice and loop constructs the additional $\mathbf{g}$ and $\mathbf{s}$ are required so that precondition satisfaction is separated from execution, as the former does not necessarily precipitate the latter. Furthermore, $\checkmark$ is required as each workflow construct is anonymous and therefore cannot easily be given its own clock. To reiterate, the fundamental purpose of a common protocol is to enable standardised extension, verification, and so on.

An example of this protocol in action is shown by the (weak) LTS in Figure 7.3. Here we have a workflow which requires a single input $a$ to execute, during execution sends a message $m$ and then receives a message $n$, finally returning a single output $b$. Notice also that if the input, output and protocol channels are removed the result is a choreography IO automaton.

### 7.2.3 Derived Syntax

In order to make the process of writing the semantics more convenient, I have created a number of derived operators. First of all, a simple disabling operator $E \rhd_\gamma G$, which proceeds with executing process $E$, but all the while allows $G$ to interrupt it with any of its initial actions. Once $E$ performs action $\gamma$ though, $G$ is disabled and can no longer interrupt. It is useful for describing situations in which an escape route is possible until some "point of no return" is reached, indicated by action $\gamma$. I only derive it for dynamic operators, as it will not be used in a static context. I also define *distributive parallel composition* $\amalg$, which composes a process with a free variable in parallel several times, each time with a substitution from the given set. Below are the derivations:

$$
\begin{aligned}
\delta.E \rhd_\gamma G &\triangleq \begin{cases} \delta.E + G & \text{if } \delta = \gamma \\ \delta.(E \rhd_\gamma G) + G & \text{otherwise} \end{cases} \\
E + F \rhd_\gamma G &\triangleq E \rhd_\gamma G + F \rhd_\gamma G \\
\mathbf{0} \rhd_\gamma G &\triangleq G \\
\Delta \rhd_\gamma G &\triangleq \Delta + G \\
\Delta_\sigma \rhd_\gamma G &\triangleq \Delta_\sigma + G \\
X \rhd_\gamma G &\triangleq X \\
\mu X.E \rhd_\gamma G &\triangleq \mu X.(E \rhd_\gamma G) \\
\coprod_{x \in \{\mathbf{a} \cdots \mathbf{z}\}} E &\triangleq E\{\mathbf{a}/x\} \mid E\{\mathbf{b}/x\} \mid \cdots \mid E\{\mathbf{z}/x\}
\end{aligned}
$$

I also define the following conventions (**Note**: in this context *only* $\sigma$ and decorations thereof are variables):

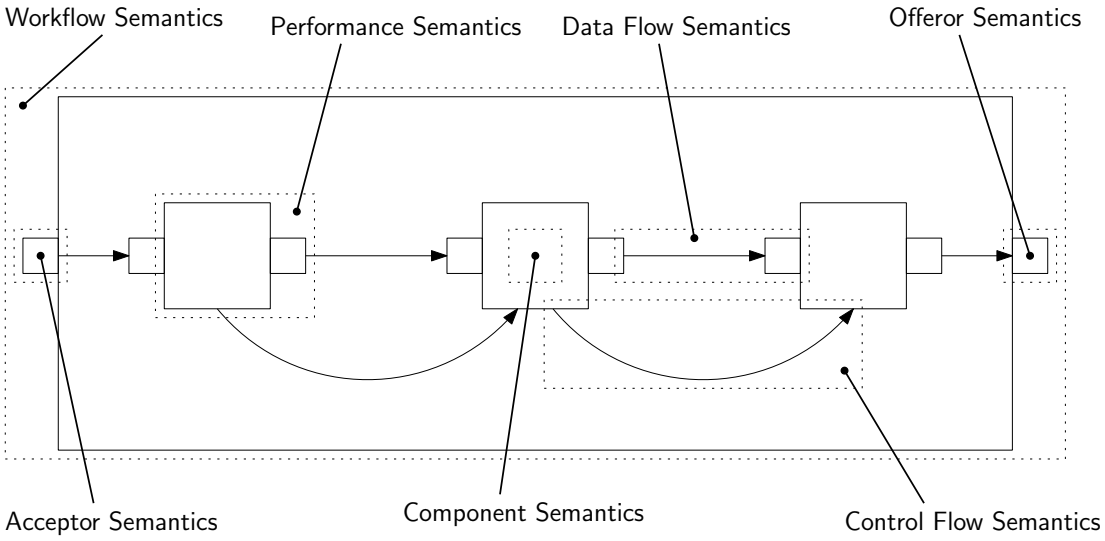



Figure 7.4: Overview of the operational semantics

$$
\begin{aligned}
E \setminus \{a, b, \cdots, z\} &\triangleq E \setminus a \setminus b \setminus \cdots \setminus z \\
E/[\sigma_1, \sigma_2, \cdots, \sigma_n] &\triangleq E/\sigma_1/\sigma_2/\cdots/\sigma_n \\
\lfloor E \rfloor \sigma(F) &\triangleq E + \sigma.F \\
\underline{\gamma}_{\{\sigma_1,\sigma_2,\cdots,\sigma_n\}}.E &\triangleq \gamma.E + \Delta_{\sigma_1} + \Delta_{\sigma_2} + \cdots + \Delta_{\sigma_n} \\
(\gamma)^n.E &\triangleq \begin{cases} \gamma.(\gamma)^{(n-1)}.E & \text{if } n > 0 \\ E & \text{otherwise} \end{cases}
\end{aligned}
$$

I use a vector for clock hiding instead of a set because the order in which clocks are hidden is important as it enforces a total order on the clocks' priority in the enclosure. I also derive the timeout operator from CaSE (though it doesn't behave exactly the same) and insistent prefix $\underline{\gamma}_{\tilde{\sigma}}$, which makes an action insistent over a particular clock sort. Finally I define a simple repetition prefix $(\gamma)^n.E$, with $n \in \mathbb{N}$, which performs the action $\gamma$ $n$ times before enabling $E$.

### 7.2.4 Structure

The semantics are broken down into individual Sections as illustrated by Figure 7.4. This shows a simple sequential workflow with control flow and data flow between a total of three performances. Each key semantic component of the workflow has a dotted line around it and a label indicating its name. These are:

- the *workflow semantics*, defined in Section 7.2.5;
- the *performance semantics*, defined in Section 7.2.6;

- the *control flow semantics*, defined in Section 7.2.7;
- the *data flow semantics*, defined in Section 7.2.8;
- the *acceptor* and *offeror semantics*, also defined in Section 7.2.8; and
- the *component semantics*, defined in Section 7.2.9.

All of these parts semantically composed form a complete behavioural semantics for a workflow. I will now consider each of these in turn.

### 7.2.5 Workflow semantics

The workflow semantics describes the behaviour of an abstract workflow. It gives a semantics to the acceptors, offerors, control and dataflow, and orchestrates them all using the various clock phases.

$$
\begin{array}{l}
{}^{\mathbf{wf}}_{w,g}[\![w[A\{C \times D\}B]g]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \\
\Big( {}^{\mathbf{wac}}_{w}[\![A]\!]^{\tilde{a}} \;\Big|\; {}^{\mathbf{wof}}_{w}[\![B]\!]_{\tilde{b}} \\
\Big| \Big( {}^{\mathbf{wcf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{\tilde{m},\tilde{n}} \;\Big|\; {}^{\mathbf{df}}_{w}[\![D]\!]^{\tilde{a}_{\mathbf{df}}}_{\tilde{b}_{\mathbf{df}}} \Big) \{\!|a \mapsto a^w | a \in \tilde{a}|\!\}\{\!|b \mapsto b^w | b \in \tilde{b}|\!\} \setminus \tilde{b}_{\mathbf{df}} \\
\Big) \setminus \{a^w | a \in \tilde{a}\} \cup \{b^w | b \in \tilde{b}\}\{\!|\otimes^w \mapsto \mathbf{s}|\!\}/[\circledS^w_0 \cdots \circledS^w_5, \odot^w]/[\textcircled{D}^p_a \,|\, a^p \in \tilde{a}_{\mathbf{cf}} \cup \tilde{a}_{\mathbf{df}}] \\
\quad \{\!|\sigma^m \mapsto m | m \in \tilde{m}|\!\}\ \{\!|\sigma^n \mapsto \overline{n} | n \in \tilde{n}|\!\}\ \{\!|\circledcirc^w \mapsto \overline{\mathbf{t}}\,|\!\}
\end{array}
$$

The semantics of a workflow consists of an *acceptor* ${}^{\mathbf{wac}}_{w}[\![A]\!]^{\tilde{a}}$ which handles precondition negotiation, an *offeror* ${}^{\mathbf{wof}}_{w}[\![B]\!]_{\tilde{b}}$ which handles postcondition negotiation, a *control flow* ${}^{\mathbf{wcf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{\tilde{m},\tilde{n}}$ and a *dataflow* ${}^{\mathbf{df}}_{w}[\![D]\!]^{\tilde{a}_{\mathbf{df}}}_{\tilde{b}_{\mathbf{df}}}$. The dataflow is composed with the control flow, and the acceptor and offeror are composed together. The acceptor and offeror define a set of inputs to the workflow $\tilde{a}$ and a set of outputs from the workflow $\tilde{b}$, respectively. Likewise the dataflow, through the workflow input and output connections, entails a set of workflow inputs and outputs, $\tilde{a}_{\mathbf{df}}$ and $\tilde{b}_{\mathbf{df}}$ respectively.

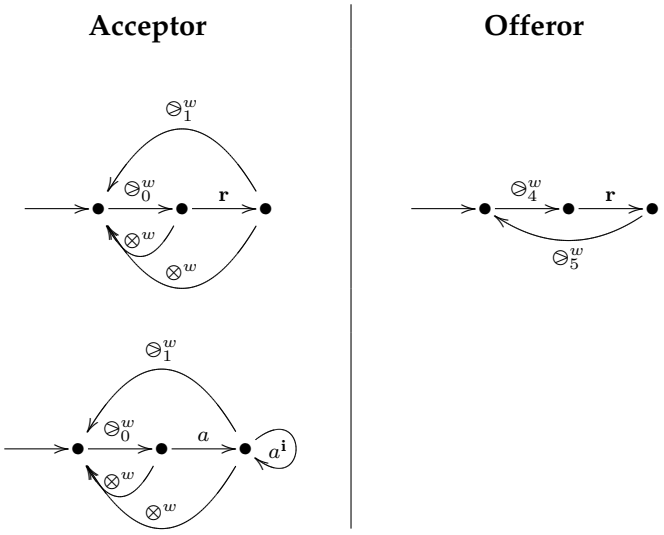
Dataflow inputs and outputs not having corresponding acceptor inputs or offeror outputs, are restricted . The remainder are decorated with a $w$ superscript (i.e. the workflow name), so they don't clash with the actual workflow input and output channels. The acceptor and offeror connect with these decorated channels. For instance a workflow may have an input $a$, which is received by the acceptor. This is then passed onto relevant dataflow connections through a decorated channel $a^w$. These decorated channels are then restricted (so they aren't viewed outside the workflow) and the phase clocks hidden in order. The exception is the cancellation clock $\otimes^w$ which is instead renamed to the stop channel $\mathbf{s}$. This means that if the workflow is instructed to deactivate, this instruction will be converted into a tick of the $\otimes^w$ (which otherwise cannot tick). The input

and output message clocks are then renamed to their respective channels and the RTC clock is also renamed to a channel, so the context can control how fast time passes.

I now proceed to define all the different parts of this workflow. A sequence diagram representing the normal overall timeline of a workflow is shown in Figure 7.5. The thick black horizontal lines within the thick black box represent clock ticks on the labelled clock. The box represents the scope of these clocks, and thus the boundary of the workflow. The presence of a blob on one of the horizontal lines indicates that the process below is a key participant in allowing the clock tick to take place. The other processes observe it, but are not directly involved. Not all possible behaviours are represented, but it provides a rough overall timeline of a workflow when successfully executed, in terms of the interactions between the control flow, dataflow and the workflow's context. This diagram should aid in understanding how all the different parts (which I define below) communicate.

I first define the semantics of the acceptor and offeror processes:

$$
\begin{array}{lcl}
{}^{\mathbf{wac}}_{w}[\![A]\!]^{\tilde{a}} & \triangleq & \Big(\Big({}^{\mathbf{ac}}_{w}[\![A]\!]^{\tilde{a}} \mid \mu X.\, \ominus^w_0 .(\underline{\mathbf{r}}_{\ominus^w_1}.\, \ominus^w_1 .X \rhd_{\ominus^w_1} \otimes^w .X)\Big) \setminus \mathbf{r} \; \{\!|a \mapsto a^{\mathbf{i}} | a \in \tilde{a}|\!\} \\
 & \mid & \prod_{a \in \tilde{a}} \mu X.\, \ominus^w_0 .(a.\mu Y.(\overline{a^{\mathbf{i}}}.Y + \ominus^w_2 .X) \rhd_{\ominus^w_2} \otimes^w .X)\Big) \setminus \{a^{\mathbf{i}} | a \in \tilde{a}\} \\
 & & \\
{}^{\mathbf{wof}}_{w}[\![B]\!]_{\tilde{b}} & \triangleq & \Big({}^{\mathbf{of}}_{w}[\![B]\!]_{\tilde{b}} \mid \mu X.\, \ominus^w_4 .\underline{\mathbf{r}}_{\ominus^w_5}.\, \ominus^w_5 .X\Big) \setminus \mathbf{r}
\end{array}
$$




The acceptors and offerors exist to decide if sufficient inputs have been received to allow the workflow to execute. The acceptor composes the precondition semantics ${}^{\mathbf{ac}}_{w}[\![A]\!]^{\tilde{a}}$ (defined in Section 7.2.8) with an agent to decide when sufficient preconditions have been satisfied. The acceptor only becomes active once the workflow is activated, indicated by $\ominus^w_0$ ticking. The enclosed precondition process accepts requisite inputs and will output an $\mathbf{r}$ once satisfied, which the top-level agent here synchronises with in the first step. Until this happens, the second phase clock $\ominus^w_1$ is held up (indicated by the under-

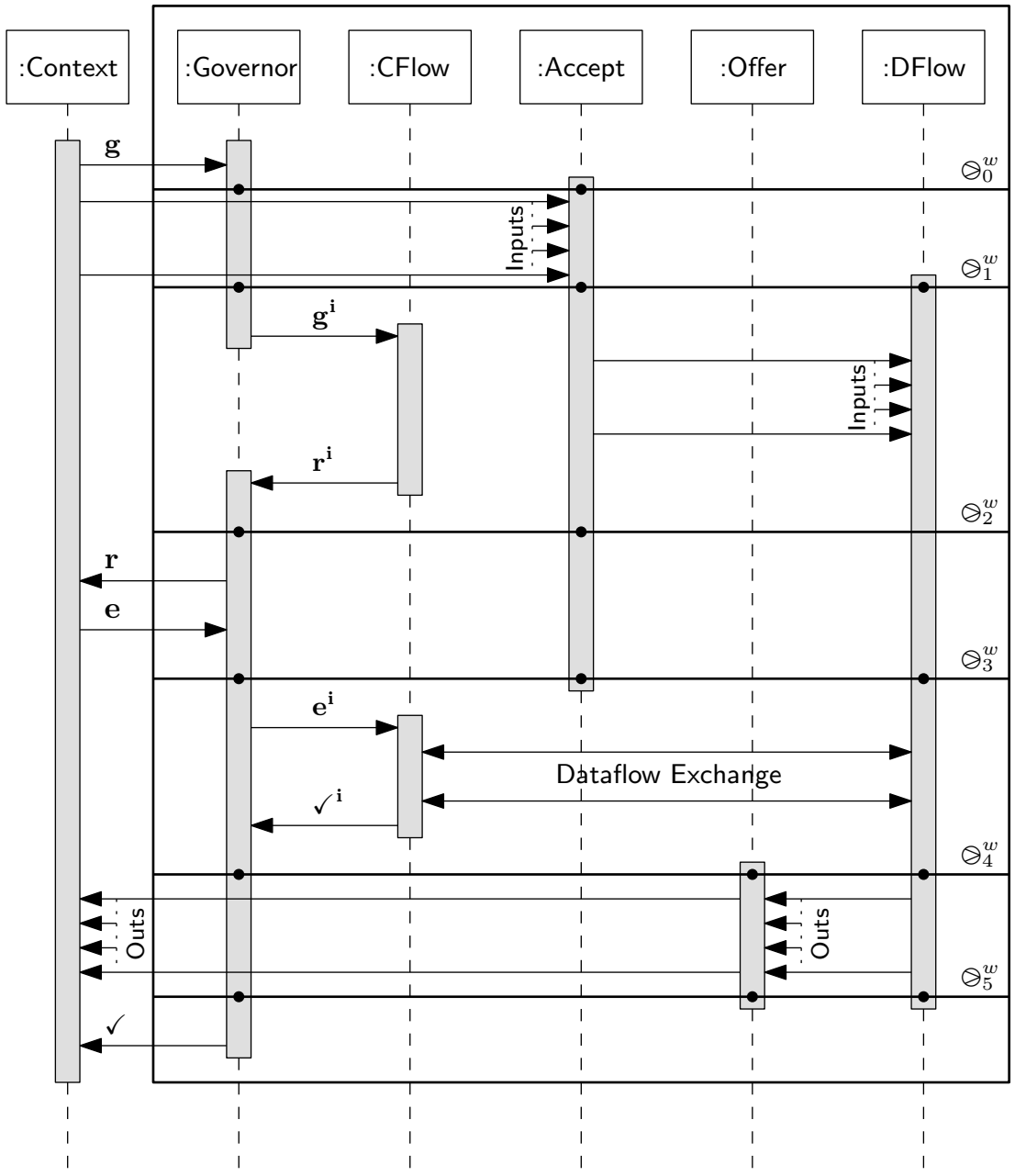



Figure 7.5: The key actors' interaction in a normal workflow timeline

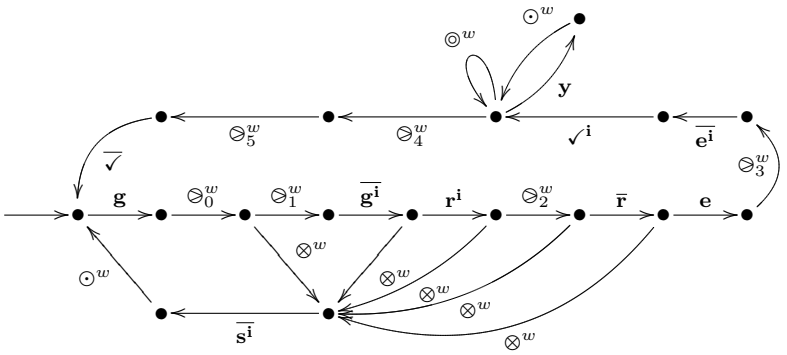
(**Note**: CFlow = Control Flow, DFlow = Dataflow, Accept = Acceptor, Offer = Offeror, and Outs = Outputs)

lining and subscript). Afterwards, $\circledcirc_1^w$ ticks indicating the workflow is has received all its required inputs, and so the control flow can be activated. Once the control flow is ready to execute $\circledcirc_2^w$ ticks to signal that the workflow is ready to execute. If the environment permits it, this is by followed $\circledcirc_3^w$, signalling the beginning of execution. However, until the latter ticks it is also possible for the cancellation clock $\otimes^w$ tick to reset and disable the workflow. In this case the acceptor will reset. The channel $\mathbf{r}$ is restricted and each input collected in $\tilde{a}$, the set of inputs provided by the acceptor process, is decorated with $\mathbf{i}$.

The acceptor also contains a second process, composed with the one described above, which takes care of input broadcasting. It consists of a collection of agents, each corresponding to the possible inputs of the workflow contained in $\tilde{a}$. Each agent broadcasts their respective input to the precondition process, as the input may appear several times in the precondition expression even though the environment will only supply it once. For instance the semantics of precondition $(a \sqcup b) \sqcap (a \sqcup c)$ would need to receive $a$ twice. Thus the input is broadcast, and the decorated input channels are restricted.

The offeror semantics is much the same as the acceptor composition, it composes the postcondition semantics ${}^{\mathbf{of}}_{w}[\![B]\!]_{\tilde{b}}$ (defined in Section 7.2.8) with a decider agent. The difference is that until the postcondition is satisfied, the workflow clock $\circledcirc_5^w$ may not tick. It also does not require a broadcaster, since each output is only supplied once (the performance which contains the workflow takes care of output broadcasting).

$$
\begin{array}{lcl}
{}^{\mathbf{wcf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{\tilde{m},\tilde{n}} & \triangleq & \left( {}^{\mathbf{cf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{\tilde{m},\tilde{n}} \{\!\{\mathfrak{F}\}\!\} \,\middle|\, \mathsf{Governor}_w \right) \backslash \{\mathbf{g^i}, \mathbf{r^i}, \mathbf{e^i}, \mathbf{s^i}, \checkmark^{\mathbf{i}}, \mathbf{y}\} \\
\text{where} & & \\
\mathsf{Governor}_w & \triangleq & \mu X.\underline{\mathbf{g}}_{\mathcal{T}_w}.\langle \circledcirc_0^w \rangle.\langle \circledcirc_1^w \rangle_{\otimes}.\overline{\underline{\mathbf{g^i}}}_{\mathcal{T}_w}. \\
& & \quad (\underline{\mathbf{r^i}}_{\mathcal{T}_w \backslash \{\otimes^w\}}.\langle \circledcirc_2^w \rangle_{\otimes}.\overline{\mathbf{r}}_{\mathcal{T}_w \backslash \{\otimes^w\}}.\underline{\mathbf{e}}_{\mathcal{T}_w \backslash \{\otimes^w\}}.\langle \circledcirc_3^w \rangle.\overline{\underline{\mathbf{e^i}}}_{\mathcal{T}_w}. \\
& & \quad\quad \mu Y.\left( \underline{\mathbf{y}}_{\mathcal{T}_w \backslash \{\circledcirc^w\}}.\mu Z.(\mathbf{y}.Z + \langle \odot^w \rangle.Y) + \circledcirc^w.Y \right. \\
& & \quad\quad\quad \left. + \checkmark^{\mathbf{i}}.\langle \circledcirc_4^w \rangle.\langle \circledcirc_5^w \rangle.\overline{\checkmark}_{\mathcal{T}_w}.X \right) \\
& & \quad \rhd_{\mathbf{e}} \otimes^w.\overline{\underline{\mathbf{s^i}}}_{\mathcal{T}_w}.\langle \odot^w \rangle.X) \\
\mathcal{T}_w & \triangleq & \{\circledcirc_0^w \cdots \circledcirc_5^w, \odot^w, \circledcirc^w, \otimes^w\} \\
\langle \sigma \rangle.E & \triangleq & \underline{\sigma}_{\mathcal{T}_w \backslash \{\sigma\}} \\
\langle \sigma \rangle_{\otimes}.E & \triangleq & \underline{\sigma}_{\mathcal{T}_w \backslash \{\sigma, \otimes^w\}}
\end{array}
$$




The control flow semantics consists of the encapsulated flow process ${}^{\mathbf{cf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{\tilde{m},\tilde{n}}$ (defined in Section 7.2.7) composed with a process called Governor. The Governor is the central linking point of the workflow, and largely manages the workflow's timeline and negotiation with the environment (hence its name). It acts as a mediator between the scheduling protocol and the phase transition system, in that it defines which clock can tick at each point of negotiation. It also schedules the top-level workflow construct.

The semantics of the control flow results in a set of control flow inputs and outputs, $\tilde{a}_{\mathbf{cf}}$ and $\tilde{b}_{\mathbf{cf}}$, respectively. These represent the inputs and outputs of all the encapsulated performances. Inputs and outputs which are not handled by the dataflow connections, and thus do not appear in $\tilde{a}_{\mathbf{df}}$ and $\tilde{b}_{\mathbf{df}}$, respectively, are restricted. The renaming set $\mathfrak{F}$ is applied to decorate all the scheduling channels of the top-level scheduler with an $\mathbf{i}$ superscript. The resulting control flow process is then composed with the Governor.

The Governor defines exactly which clocks can occur at which point. It does this via *urgency*, which is indicated by underlining the associated action, and with a clock to be stalled in the subscript (see derived syntax in Section 7.2.3). In addition, I enclose some clock ticks with angle brackets $\langle\sigma\rangle$ to indicate that *every* phase clock other than $\sigma$ is stalled, where $\mathcal{T}_w$ is the set of phase clocks in workflow $w$. The Governor is one of the only agents in a workflow to prevent clocks from ticking, and thus effectively defines the phase transition system.

The Governor first requires a $\mathbf{g}$ input before anything can be done in the workflow. All phase clocks are held up at this point. Following $\mathbf{g}$ the Governor ticks $\circledS^w_0$ to indicate activation, which the acceptor process (already described) listens for. Once the requisite workflow inputs have been received $\circledS^w_1$ ticks and the governor then sends a $\mathbf{g}^{\mathbf{i}}$ signal to the top-level scheduler, allowing it to attempt to fulfil its preconditions. The Governor then waits for the top-level scheduler to output an $\mathbf{r}^{\mathbf{i}}$ to indicate it is ready. When this happens, the third clock ticks to indicate that all the preconditions are fulfilled (provided the acceptor also allows this).

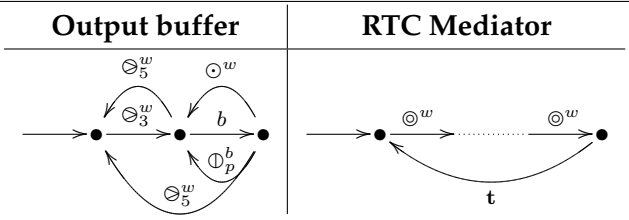
The Governor then informs the environment via $\mathbf{r}$ that the workflow as a whole is ready to execute, to which the environment can either respond with an $\mathbf{e}$ or an $\mathbf{s}$. The $\mathbf{s}$ can occur at any point up until $\mathbf{e}$ is received, and will cause the entire workflow to reset and not execute. This is done via the derived disabling operator at the bottom the process definition. If an $\mathbf{s}$ is received it is converted into a $\otimes^w$ signal which the Governor

detects (as does the acceptor) and sends an $\mathbf{s^i}$ to the top-level scheduler, forcing it to reset and then ticks the yield clock $\odot^w$, which completes cancellation. On the other hand, if $\mathbf{e}$ is received the Governor ticks the third clock $\circledcirc_3^w$, to indicate the execution phase, and passes execution permission onto the top-level scheduler via $\mathbf{e^i}$. From this point on, the workflow must run its course.

During the execution phase, the Governor permits three possibilities. The yield clock can tick if a request is received via $\mathbf{y}$, the RTC can tick, or the top-level scheduler can indicate its completion via $\checkmark^{\mathbf{i}}$. No other phase clocks can occur during execution. Once the top-level scheduler completes, the Governor begins the post-execution commit phase by ticking $\circledcirc_4^w$. In this phase the offeror picks up all the outputs it requires from the body. Once this has done, the sixth and final workflow phase clock $\circledcirc_5^w$ ticks, the Governor passes a $\checkmark$ to its environment to signal the workflow is complete, and the whole workflow resets. All that remains to be done in the control flow semantics is the restriction of the internal scheduling channels used by the Governor. The completes the discussion of the workflow semantics as a whole.

### 7.2.6 Performance Semantics

$$
\begin{aligned}
{}^{\mathbf{cf}}_{w,g}[\![p[P]]\!]^{\{a^p|a\in\tilde{a}\},\{b^p|b\in\tilde{b}\}}_{\tilde{m},\tilde{n}} \triangleq\ & \Big( {}^{\mathbf{pf}}_{p,g_{\mathbf{i}}}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \\
& \mid \prod_{b\in\tilde{b}} \mu X.\ \circledcirc_3^w\ .(\mu Y.\underline{b}_{\circledcirc_b^p}.(\circledcirc_b^p.Y + \circledcirc_5^w.X + \odot^w.Y) + \circledcirc_5^w.X) \\
& \mid \mu X.(\circledcirc^w)^{g_{\mathbf{i}}/g}.\mathbf{t}.X \\
& \Big)\backslash\tilde{b} \setminus \{\mathbf{t},\checkmark^{\mathbf{i}}\}[\![a\mapsto a^p|a\in\tilde{a}]\!][\![\checkmark_{\mathbf{c}}\mapsto\checkmark]\!] \setminus \mathbf{x}
\end{aligned}
$$




The performance wrapper places a component semantics ${}^{\mathbf{pf}}_{p,g_{\mathbf{i}}}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}}$ (defined in Section 7.2.9) into the context of a workflow. It has two functions: mediating data flow and mediating the RTC between the performance and the enclosing workflow. Data flow is mediated by converting the inputs and outputs of the performance into a form which can be relayed to the dataflow connections of the workflow. The inputs are simply renamed to a unique name by decorating them with the performance name $p$ (the actual input buffers are provided by input connections) via $a^p \mapsto a$. In contrast the outputs are buffered and can only be conveyed to respective connections when the respective performance output clock ($\circledcirc_p^b$) ticks, which synchronises with appropriate connections. Alternatively, if either the yield clock or final workflow clock ticks *before* the output can

be conveyed, it is purged. This is necessary so that new data from the encapsulated component can be output should the performance be executed in a loop.

The use of clocks to directly convey data passing is used instead of isochronic broadcast because its use ensures that each recipient receives the input exactly once. The problem with isochronic broadcast is that looping receivers will simply receive the same input again and again. Direct use of a clock avoids this also cuts down on the number of transitions required for the broadcast.

The wrapper also mediates between the workflow's RTC and the performance's RTC using the $t$ channel. Recall that the workflow semantics in Section 7.2.5 renamed the RTC clock to the $t$ channel, so it would communicate with its respective performance wrapper. The $t$ channel outputs every time the performance's RTC ticks, and the workflow must tick its clock several times before allowing the performance clock to tick. The number of workflow RTC ticks is calculated statically by dividing the performance time granularity $g_{\mathbf{i}}$ by workflow granularity $g$. The workflow clock then must tick this number of times, and then a $t$ is input from the performance allowing its clock to tick. This ensures that a form of global synchrony can be ensured without the need for a global clock.

The performance wrapper also restricts the $\mathbf{x}$ channel which is used to force a compensation in an enclosed compensating workflow. This is not relevant for normal workflows. Also, the compensation completion channel $\checkmark_{\mathbf{c}}$ is renamed to a normal $\checkmark$ as normal workflows do not distinguish completion from compensation.

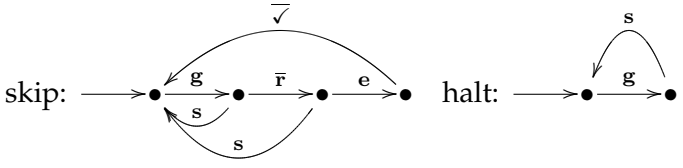
### 7.2.7 Control flow semantics

This section describes the semantics for each of the workflow control flow operators. The basic idea is that every construct in the language has a *scheduler* associated with it, which defines the execution order of its sub-components, an idea we have utilised in the past (Norton et al., 2005; Foster, 2007). In order to add a new construct to the language, all one needs do is define a new scheduler. All of the schedulers utilise the protocol to schedule their encapsulated processes. The main difference between these semantics and our previous work is the use of binary operators, which makes the semantics more flexible and elegant.

#### Skip and Halt

I first consider the two simplest workflow constructs, skip and halt:

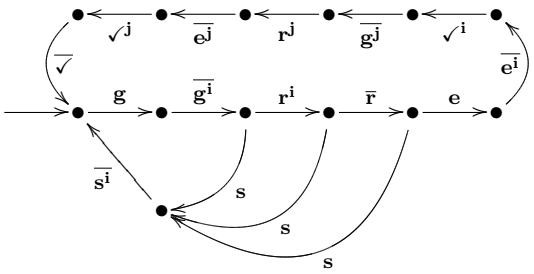
$$\begin{array}{rcl} {}^{\mathbf{cf}}_{w,g}[\![\epsilon]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.(\mathbf{g}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\checkmark}.X \rhd_{\mathbf{e}} \mathbf{s}.X) \\ {}^{\mathbf{cf}}_{w,g}[\![\delta]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.\mathbf{g}.\mathbf{s}.X \end{array}$$



Both of these are naturally very simple. Skip is activated by receiving a $\mathbf{g}$ and immediately becomes ready to execute, conveying an $\mathbf{r}$ to its environment. It can then either be executed via $\mathbf{e}$, which is simply a dummy run returning a ✓ immediately, or it can be stopped via the $\mathbf{s}$ signal. In contrast, halt once activated never can become ready, and will only accept a stop signal. Both have neither inputs, outputs, nor messages and therefore the four associated sets are simply $\emptyset$.

**Sequence**

$$\begin{aligned}{}^{\mathbf{cf}}_{w,g}[\![P \fatsemi Q]\!]^{\tilde{a}_\mathbf{i}\cup\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{i}\cup\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{i}\cup\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{i}\cup\tilde{n}_\mathbf{j}} \quad \triangleq \quad & \Big( {}^{\mathbf{cf}}_{w,g}[\![P]\!]^{\tilde{a}_\mathbf{i},\tilde{b}_\mathbf{i}}_{\tilde{m}_\mathbf{i},\tilde{n}_\mathbf{i}} \{\!\{\mathfrak{F}\}\!\} \;\Big|\; {}^{\mathbf{cf}}_{w,g}[\![Q]\!]^{\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{j}} \{\!\{\mathfrak{G}\}\!\} \\ & \Big| \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.(\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\checkmark^\mathbf{i}.\overline{\mathbf{g^i}}.\mathbf{r^j}.\overline{\mathbf{e^j}}.\checkmark^\mathbf{j}.\overline{\checkmark}.X \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^i}}.X)\Big) \setminus \mathfrak{R}\end{aligned}$$




Sequence, though one of the simplest constructs, illustrates well the protocol and style of composition in the rest of the semantics. As such, I will explain it in more detail than the rest. As is the case with all binary operators, the semantics for both components $P$ and $Q$ have their channels renamed for distinction using $\mathfrak{F}$ and $\mathfrak{G}$, which decorate the protocol channels with $\mathbf{i}$ and $\mathbf{j}$ respectively. They are composed with a *scheduler*, which enforces the execution order using the protocol channels. Furthermore, the input sets, $\tilde{a}_\mathbf{i}$ and $\tilde{a}_\mathbf{j}$ (for processes $P$ and $Q$ respectively), and output sets, $\tilde{b}_\mathbf{i}$ and $\tilde{b}_\mathbf{j}$, of each component are composed to form the overall sets for the composition. The input message and output message sets of both sides are composed in the same way.

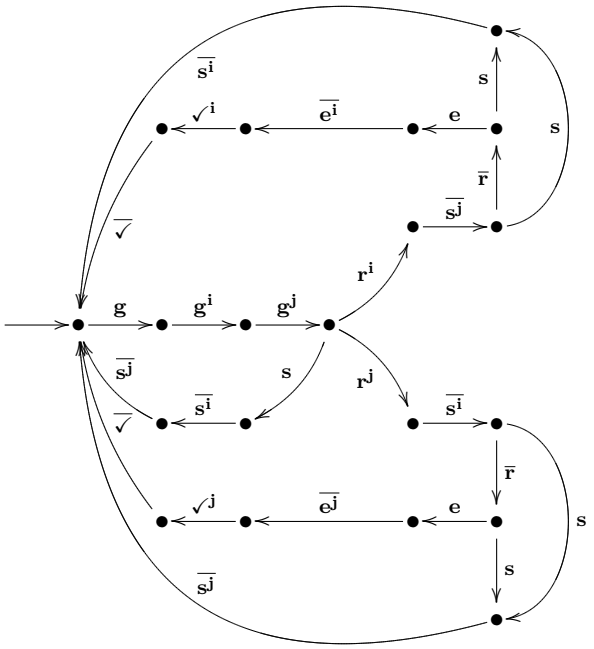
The scheduler for sequence first waits for an activation signal $\mathbf{g}$ from the environment; either the top-level governor or a parent scheduler. Upon receiving this, the scheduler in turn activates the left component and then waits for it to declare readiness to execute via $\mathbf{r^i}$. Upon doing so, the scheduler will declare its readiness to the environment via $\mathbf{r}$, since we make the assumption for sequence that readiness only depends on the readiness of the first element. The environment can then give permission to execute via $\mathbf{e}$ or can order execution to stop via $\mathbf{s}$. Notice the use of the derived disabling operator to do this – between receiving $\mathbf{r^i}$ and $\mathbf{e}$ it is possible for the environment to provide the stop signal. Upon doing so the scheduler orders the left component to stop and then returns to inactive.

Alternatively, if given permission to execute, the scheduler will pass on permission to the left component via $\mathbf{e^i}$ and then wait for a completion signal via $\checkmark^\mathbf{i}$. Then the

right component is executed using the same sequence of actions, although at this point stopping is not an option. Once the second component is complete the scheduler issues a tick ✓ indicating its own completion. Both sets of internal channels are restricted using the $\mathfrak{R}$ set.

### Choice

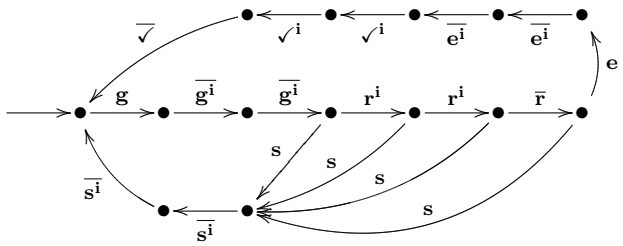
$$\begin{array}{l}\mathop{\mathbf{cf}}_{w,g}[\![P \oplus Q]\!]^{\tilde{a}_\mathbf{i}\cup\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{i}\cup\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{i}\cup\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{i}\cup\tilde{n}_\mathbf{j}} \triangleq \\ \Big(\mathop{\mathbf{cf}}_{w,g}[\![P]\!]^{\tilde{a}_\mathbf{i},\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{i},\tilde{n}_\mathbf{i}} \{\!\{\mathfrak{F}\}\!\} \;\Big|\; \mathop{\mathbf{cf}}_{w,g}[\![Q]\!]^{\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{j}} \{\!\{\mathfrak{G}\}\!\} \;\Big|\; \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.\overline{\mathbf{g^j}}. \\ \quad (\mathbf{r^i}.\overline{\mathbf{s^j}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\checkmark^\mathbf{i}.\overline{\checkmark}.X \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^i}}.X) \\ \quad +\mathbf{r^j}.\overline{\mathbf{s^i}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^j}}.\checkmark^\mathbf{j}.\overline{\checkmark}.X \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^j}}.X) \\ \quad +\mathbf{s}.\overline{\mathbf{s^i}}.\overline{\mathbf{s^j}}.X)) \setminus \mathfrak{R}\end{array}$$




Choice is a little more complicated than sequence, though follows much the same sequence of actions. The main differences are that, since readiness is indicated by one of the two components becoming ready, both sides are activated $(\mathbf{g^i}, \mathbf{g^j})$ and the one declaring readiness first becomes the execution candidate, with the other being stopped. Alternatively, if an s is received, both will end up being stopped.

### Synchronous Parallel Composition

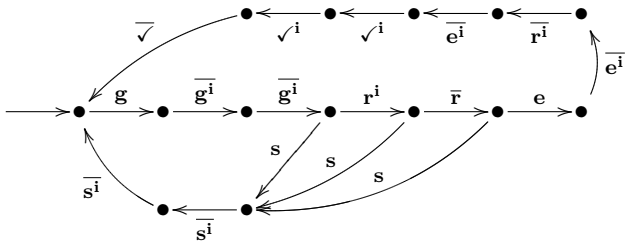
$$\begin{array}{l}\mathop{\mathbf{cf}}_{w,g}[\![P \mid Q]\!]^{\tilde{a}_\mathbf{i}\cup\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{i}\cup\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{i}\cup\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{i}\cup\tilde{n}_\mathbf{j}} \triangleq \\ \Big(\Big(\mathop{\mathbf{cf}}_{w,g}[\![P]\!]^{\tilde{a}_\mathbf{i},\tilde{b}_\mathbf{i}}_{\tilde{m}_\mathbf{i},\tilde{n}_\mathbf{i}} \;\Big|\; \mathop{\mathbf{cf}}_{w,g}[\![Q]\!]^{\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{j}}\Big) \{\!\{\mathfrak{F}\}\!\} \;\Big|\; \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.\overline{\mathbf{g^i}}. \quad (\mathbf{r^i}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\overline{\mathbf{e^i}}.\checkmark^\mathbf{i}.\checkmark^\mathbf{i}.\overline{\checkmark}.X \\ \quad \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^i}}.\overline{\mathbf{s^i}}.X)) \setminus \mathfrak{R}\end{array}$$




Synchronous parallel composition is similar to choice, although much simpler as both sides are executed and thus must both be ready. Notice that I do not distinguish between the two sides with separate channels ($\mathfrak{F}$ is used to rename both sides), this is because parallel composition is symmetric and thus as long as both become ready and execute it doesn't matter in which order. All that is required for readiness is that two $\mathbf{r}^{\mathbf{i}}$ signals are received.

**Asynchronous Parallel Composition**

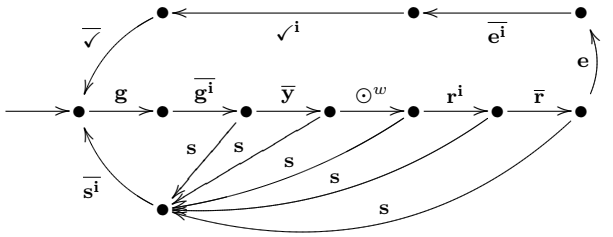
$$\begin{array}{l}{}^{\mathbf{cf}}_{w,g}[\![P \parallel Q]\!]^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\ \quad \left( \left( {}^{\mathbf{cf}}_{w,g}[\![P]\!]^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \;\middle|\; {}^{\mathbf{cf}}_{w,g}[\![Q]\!]^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \right) \{\!|\mathfrak{F}|\!\} \;\middle|\; \mu X.\mathbf{g}.\overline{\mathbf{g}^{\mathbf{i}}}.\overline{\mathbf{g}^{\mathbf{i}}}. \quad (\mathbf{r}^{\mathbf{i}}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{i}}}.\mathbf{r}^{\mathbf{i}}.\overline{\mathbf{e}^{\mathbf{i}}}.\checkmark^{\mathbf{i}}.\checkmark^{\mathbf{i}}.\overline{\checkmark}.X \right. \\ \qquad\qquad \left. \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s}^{\mathbf{i}}}.\overline{\mathbf{s}^{\mathbf{i}}}.X)) \setminus \mathfrak{R} \right)\end{array}$$




Asynchronous parallel execution is similar to synchronous parallel composition except that it needs only one side to be ready before the whole construct is ready. However, it does require both sides to complete execution for it to complete.

**Yield**

$$\begin{array}{l}{}^{\mathbf{cf}}_{w,g}[\![\uparrow P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \left( {}^{\mathbf{cf}}_{w,g}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \{\!|\mathfrak{F}|\!\} \;\middle|\; \mu X.\mathbf{g}.\overline{\mathbf{g}^{\mathbf{i}}}.(\overline{\mathbf{y}}.\odot^{w}.\mathbf{r}^{\mathbf{i}}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{i}}}.\checkmark^{\mathbf{i}}.\overline{\checkmark}.X \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s}^{\mathbf{i}}}.X) \right) \setminus \mathfrak{R}\end{array}$$

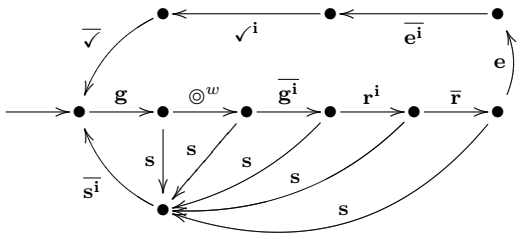



The yield operator's semantics is, naturally, based around the yield clock $\odot^{w}$ which must tick before the encapsulated process can execute. This forces all other behaviour

within the workflow to progress as much as possible, either to completion or to a yield, and only then will the process be executed. It thus ensures that all possible preconditions are satisfied by other parallel processes executing.

**Wait**

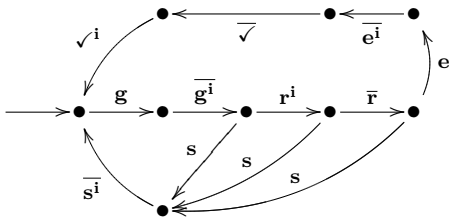
$$\llbracket \circ P \rrbracket^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \left( \llbracket P \rrbracket^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \{\!|\mathfrak{F}|\!\} \;\middle|\; \mu X.\mathbf{g}.(\circledcirc^{w}.\overline{\mathbf{g^i}}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\checkmark^{\mathbf{i}}.\overline{\checkmark}.X \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s^i}}.X) \right) \setminus \mathfrak{R}$$




The wait operator's semantics is virtually identical to that of yield, the main difference being the use of the RTC $\circledcirc^{w}$. In addition, the time operator only activates the inner workflow when the time unit has passed, this is to ensure that if another time unit is inside it can only pass afterwards.

**Fork**

$$\llbracket \curvearrowleft P \rrbracket^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \left( \llbracket P \rrbracket^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \{\!|\mathfrak{F}|\!\} \;\middle|\; \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.(\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\overline{\checkmark}.\checkmark^{\mathbf{i}}.X \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s^i}}.X) \right) \setminus \mathfrak{R}$$

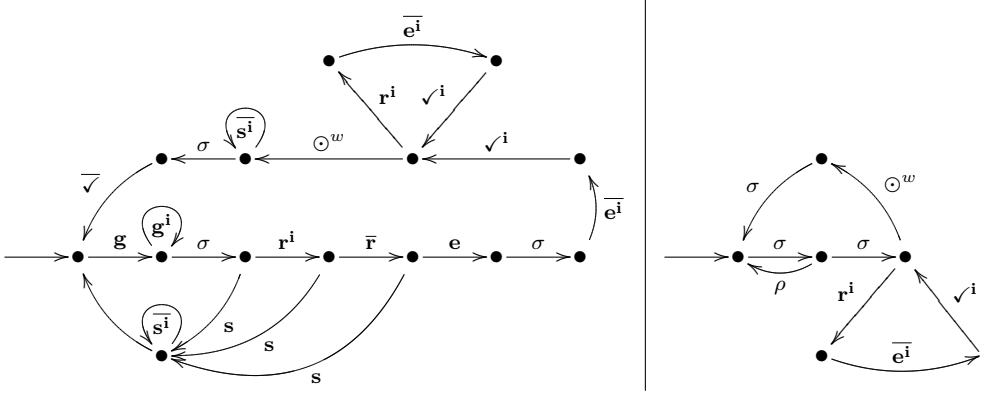



The semantics of fork is very simple, largely because it relies on the structure of the workflow to function correctly. It performs the readiness cycle in the usual way, but when coming to completion it sends out ✓ *before* it has received the completion signal from the enclosed process. This means that relative to the control flow context, the normal progress can continue. However, the workflow as a unit cannot finish until all forked processes have completed, as for $\ominus^{w}_{4}$ to tick no internal $\tau$ actions must remain. If placed in a loop, the process will not become ready to execute again until the internal $\checkmark^{\mathbf{i}}$ has been received.

**Interleaving**

$$\begin{array}{l}
{\scriptstyle \mathbf{cf}\atop\scriptstyle w,g}\Big[\!\Big[ |||_k \vec{P} \Big]\!\Big]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \Big(\mu X.\mathbf{g}.\mu Y.(\overline{\mathbf{g^i}}.Y + \sigma.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\sigma.\overline{\mathbf{e^i}}.\checkmark^{\mathbf{i}}.\mu Z. \quad (\mathbf{r^i}.\overline{\mathbf{e^i}}.\checkmark^{\mathbf{i}}.Z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \odot^w .\mu Y.(\overline{\mathbf{s^i}}.Y + \sigma.\overline{\checkmark}.X)) \\
\qquad\qquad\qquad\qquad \rhd_{\mathbf{e}}\, \mathbf{s}.\mu Z.(\overline{\mathbf{s^i}}.Z + \rho.X)) \\
\qquad\qquad \Big| \prod_{n=1}^{k-1} \mu X.\sigma.(\sigma.\mu Y.(\mathbf{r^i}.\overline{\mathbf{e^i}}.\checkmark^{\mathbf{i}}.Y + \odot^w.\sigma.X) + \rho.X) \\
\qquad\qquad \Big| \ {\scriptstyle \mathbf{bag}\atop\scriptstyle w,g}\Big[\!\Big[ \vec{P} \Big]\!\Big]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \{\!\{\mathfrak{F}\}\!\}\Big) \setminus \mathfrak{R}/\sigma/\rho \\
\\
{\scriptstyle \mathbf{bag}\atop\scriptstyle w,g}\Big[\!\Big[ P : \vec{Q} \Big]\!\Big]^{\tilde{a}_{\mathbf{i}}\cup\tilde{a}_{\mathbf{j}},\tilde{b}_{\mathbf{i}}\cup\tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}}\cup\tilde{m}_{\mathbf{j}},\tilde{n}_{\mathbf{i}}\cup\tilde{n}_{\mathbf{j}}} \triangleq {\scriptstyle \mathbf{cf}\atop\scriptstyle w,g}[\![P]\!]^{\tilde{a}_{\mathbf{i}},\tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}},\tilde{n}_{\mathbf{i}}} \ \Big|\ {\scriptstyle \mathbf{bag}\atop\scriptstyle w,g}\Big[\!\Big[ \vec{Q} \Big]\!\Big]^{\tilde{a}_{\mathbf{j}},\tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}},\tilde{n}_{\mathbf{j}}} \\
\\
{\scriptstyle \mathbf{bag}\atop\scriptstyle w,g}[\![\mathbf{Nil}]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} \triangleq \mathbf{0}
\end{array}$$

**Main Scheduler** | **Mini Scheduler**




The semantics for interleaving is the most complicated in the non-compensating fragment, as well as the most versatile. To reiterate, interleaving is a construct which runs up to $k$ of the bag of processes in parallel, in an arbitrary order. Usually all of the bag of processes would execute, but if some cannot for some reason they are left out. The actual order of execution would usually be resolved by the dataflow and/or message flow. The interleaving scheduler is composed of a main scheduler which acts as a kind of mini Governor agent and $k-1$ under-scheduler agents, making a total of $k$ agents. The job of the main scheduler is ensure that at least one sub-process in the bag is ready to execute and negotiate with the environment. It also detects when all internal activity of the construct has ceased following execution of the bag, and outputs a completion signal. The remaining $k-1$ schedulers run the other parallel processes, alternatively if $k = 1$ only the main scheduler will exist (the bag is sequentially executed).

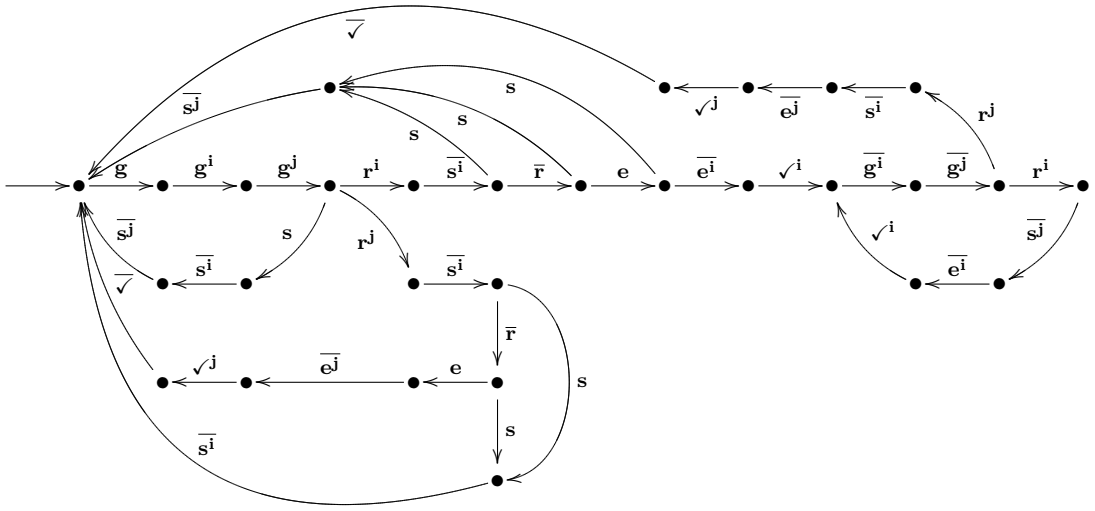
The main scheduler, in addition to using the yield clock to tell when all its subprocesses have completed, uses private clocks $\sigma$ and $\rho$ to broadcast the activate signal and stop signal if need be (they are similar to workflow phase clocks). The yield clock cannot be used for this purpose because stopping may be required before the enclosing workflow has entered the execute phase (if an interleave is unguarded in the control flow).

The interleaving scheduler first activates all its sub-processes, using $\sigma$ to help perform this broadcast. It then waits for any one of the processes to become ready via $\mathbf{r^i}$, and passes the request on to its environment. If successful, execution of the first process proceeds as normal (otherwise all processes are stopped, and $\rho$ ticks). Once completed, the scheduler enters an inner loop which readies and then executes each of the sub-processes in turn. Furthermore, once $\sigma$ has ticked twice, indicating that the main scheduler has been given permission to execute, the other $k-1$ schedulers are activated and begin executing other processes in parallel.

Once the yield clock can tick it is assumed that all possible internal behaviour has completed and thus the scheduler terminates. One could question if the RTC could be used for this purpose, but the RTC only detects completion of immediate dataflow, and not all performances. If other performances exist in the workflow which can satisfy some of the preconditions of processes in the interleaving construct then they must be permitted to execute before the interleaving completes. Before terminating the scheduler broadcasts a stop signal to any internal workflows which did not execute for whatever reason. Thus the semantics allows the maximum number of workflows to execute which are able to, but requires at least one to do so. In addition the semantics invokes an auxiliary semantic constructor **bag** which parallel composes all the internal processes and gathers their inputs, outputs and messages together. The bag is composed with the schedulers and, as for parallel composition, all processes are given the same proxy channels because they needn't be distinguished by the schedulers.

**Kleene Star**

$$
\begin{aligned}
&\underset{w,g}{\mathbf{cf}}[\![P^*Q]\!]^{\tilde{a}_\mathbf{i}\cup\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{i}\cup\tilde{b}_\mathbf{j}}_{\tilde{m}_\mathbf{i}\cup\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{i}\cup\tilde{n}_\mathbf{j}} = \\
&\quad\Big( \quad \underset{w,g}{\mathbf{cf}}[\![P]\!]^{\tilde{a}_\mathbf{i},\tilde{b}_\mathbf{i}}_{\tilde{m}_\mathbf{i},\tilde{n}_\mathbf{i}}\,\{\!|\mathfrak{F}|\!\} \;\Big|\; \underset{w,g}{\mathbf{cf}}[\![Q]\!]^{\tilde{a}_\mathbf{j},\tilde{b}_\mathbf{i}}_{\tilde{m}_\mathbf{j},\tilde{n}_\mathbf{j}}\,\{\!|\mathfrak{G}|\!\} \\
&\quad| \quad \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.\overline{\mathbf{g^j}}. \quad ( \quad \mathbf{r^i}.\overline{\mathbf{s^j}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\surd^\mathbf{i}.\mathsf{Iterate} \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^i}}.\overline{\mathbf{s^j}}.X) \\
&\qquad\qquad\qquad + \quad \mathbf{r^j}.\overline{\mathbf{s^i}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^j}}.\surd^\mathbf{j}.\overline{\surd}.X \rhd_\mathbf{e} \mathbf{s}.\overline{\mathbf{s^j}}.X) \\
&\qquad\qquad\qquad + \quad \mathbf{s}.\overline{\mathbf{s^i}}.\overline{\mathbf{s^j}}.X)\Big) \setminus \mathfrak{R} \\
&\mathbf{where}\ \mathsf{Iterate} = \mu Y.\overline{\mathbf{g^i}}.\overline{\mathbf{g^j}}.(\mathbf{r^i}.\overline{\mathbf{s^j}}.\overline{\mathbf{e^i}}.\surd^\mathbf{i}.Y + \mathbf{r^j}.\overline{\mathbf{s^i}}.\overline{\mathbf{e^j}}.\surd^\mathbf{j}.\overline{\surd}.X)
\end{aligned}
$$



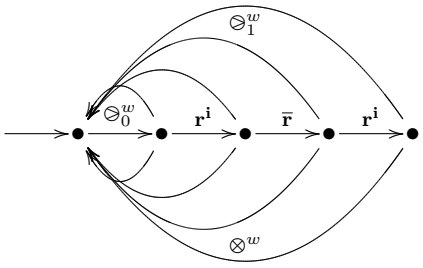

Kleene star represents a loop with a body $P$ and exit condition $Q$, specifically a type of "until" loop. The semantics are very similar to that of the choice operator, the main difference being that $P$ (i.e. the loop body), if permitted to execute will perform the normal scheduling protocol, execute the body and then activate the Iterate process. This process executes the body over and over until the condition is picked. The choice between the body and the condition is entirely non-deterministic, although as I showed in Chapter 5, Section 5.5.2 by using yields prioritisation of either the body or condition can be achieved. This would then turn it into a proper iteration construct similar to those found in programming languages – in particular it is normal for the condition evaluated to have priority over the body. Alternatively, if the preconditions of body and exit condition are naturally exclusive then a yield may not be needed.

### 7.2.8 Dataflow semantics

The dataflow semantics take care of moving data around the workflow and ensuring the preconditions and postconditions are fulfilled before allowing the next workflow phase. The precondition and postcondition agents, also known as *acceptors* and *offerors* use a much simplified scheduling protocol to decide when sufficient conditions for execution have been met. There is only one channel, $r$, which indicates when a condition has been satisfied. Another two points to be reiterated are that $\ominus_2^w$ is the workflow ready phase and $\ominus_4^w$ is the workflow finished phase. Thus the acceptors and offerors hook into these clocks in particular to perform their duties. Additionally if $\otimes^w$ ticks to indicate deactivation of the workflow, all acceptor agents are terminated, usually via use of the derived disabling operator.

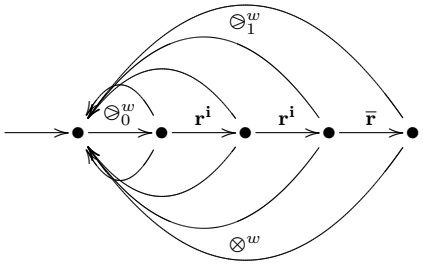
#### Preconditions

$$\begin{aligned}{}^{\mathbf{ac}}_{w}[\![A_1 \sqcup A_2]\!]^{\tilde{a_1}\cup\tilde{a_2}} \;=\; & (({}^{\mathbf{ac}}_{w}[\![A_1]\!]^{\tilde{a_1}} \mid {}^{\mathbf{ac}}_{w}[\![A_2]\!]^{\tilde{a_2}})\{\!|\mathbf{r}\mapsto\mathbf{r^i}|\!\} \\ & \mid \mu X.\, \ominus_0^w .(\mathbf{r^i}.\bar{\mathbf{r}}.\mathbf{r^i}.\mathbf{0} \rhd (\ominus_1^w.X + \otimes^w.X)) \setminus \mathbf{r^i}\end{aligned}$$




Choice between two conditions $A_1 \sqcup A_2$ is satisfied, naturally, when one of the two sub-agents declares readiness. It is also possible, of course, that both may become ready and so this is allowed for as well. Like all of the acceptor agents, this choice may only become active once the workflow itself is active, indicated by $\ominus_1^w$ ticking. The agent then waits for an $\mathbf{r^i}$, which may come from either of the sub-conditions. It then passes readiness up to the agent above and waits for $\ominus_2^w$ to tick, also allowing the other side to indicate readiness via $\mathbf{r^i}$ at this time.

$$\begin{aligned} {}^{\mathbf{ac}}_{w}[\![A_1 \sqcap A_2]\!]^{\tilde{a_1} \cup \tilde{a_2}} = \; & (({}^{\mathbf{ac}}_{w}[\![A_1]\!]^{\tilde{a_1}} \mid {}^{\mathbf{ac}}_{w}[\![A_2]\!]^{\tilde{a_2}})\{\!|\mathbf{r} \mapsto \mathbf{r^i}|\!\} \\ & \mid \mu X.\, \ominus_0^w .(\mathbf{r^i}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{0} \rhd (\ominus_1^w .X + \otimes^w .X))) \setminus \mathbf{r^i} \end{aligned}$$

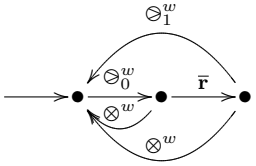



Composition of two conditions $A_1 \sqcap A_2$ works in much the same way as choice, except of course it requires both sub-conditions to declare their readiness before passing $\mathbf{r}$ upwards.

$$ {}^{\mathbf{ac}}_{w}[\![\mathbf{0}]\!]^{\emptyset} = \mathbf{0} $$

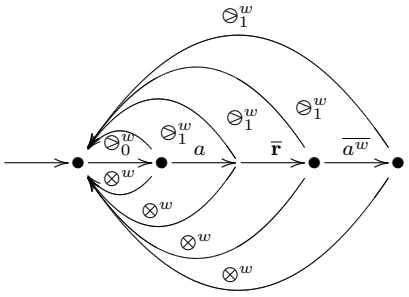
False (**0**) is the condition which is never satisfied and thus does not return an $\mathbf{r}$ and has no behaviour.

$$ {}^{\mathbf{ac}}_{w}[\![\mathbf{1}]\!]^{\emptyset} = \mu X.\, \ominus_0^w .(\overline{\mathbf{r}}.\, \ominus_1^w .X \rhd \otimes^w .X) $$




True (**1**) is the condition which is always satisfied, and thus passes $\mathbf{r}$ upwards immediately.

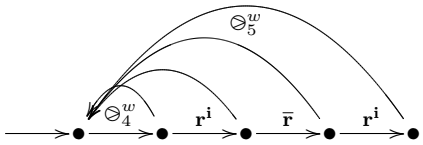
$$\overset{\mathbf{ac}}{w}[\![\mathsf{a}]\!]^{\{a\}} = \mu X.\, \circledcirc_0^w\, .(a.\overline{\mathbf{r}}.\overline{a^w}.\mathbf{0} \rhd (\circledcirc_1^w.X + \otimes^w.X))$$




Input (a) works in much the same way as other acceptor agents, except it also requires that an input is passed to it by the environment before it can declare readiness. In addition, once it has passed on the readiness condition it will broadcast its input on an internal channel $a^w$ to the relevant input connectors, whose semantics we shall see shortly.

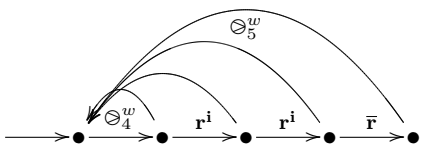
**Postconditions**

$$\begin{aligned}\overset{\mathbf{of}}{w}[\![B_1 \sqcup B_2]\!]_{\tilde{b_1} \cup \tilde{b_2}} \;=\; & ((\overset{\mathbf{of}}{w}[\![B_1]\!]_{\tilde{b_1}} \mid \overset{\mathbf{of}}{w}[\![B_2]\!]_{\tilde{b_2}})\{\!\{\mathbf{r} \mapsto \mathbf{r^i}\}\!\} \\ & \mid \mu X.\, \circledcirc_4^w\, .(\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{r^i}.\, \circledcirc_5^w\, .X \rhd_{\overline{\mathbf{r}}} \circledcirc_5^w.X)) \setminus \mathbf{r^i}\end{aligned}$$



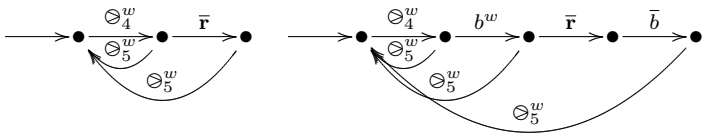

Choice between two postconditions is virtually identical to the analogue for preconditions, the only difference being that its behaviour occurs between the clocks 4 and 5, i.e. in the commit phase of the workflow.

$$\begin{aligned}\overset{\mathbf{of}}{w}[\![B_1 \sqcap B_2]\!]_{\tilde{b_1} \cup \tilde{b_2}} \;=\; & ((\overset{\mathbf{of}}{w}[\![B_1]\!]_{\tilde{b_1}} \mid \overset{\mathbf{of}}{w}[\![B_2]\!]_{\tilde{b_2}})\{\!\{\mathbf{r} \mapsto \mathbf{r^i}\}\!\} \\ & \mid \mu X.\, \circledcirc_4^w\, .(\mathbf{r^i}.\mathbf{r^i}.\overline{\mathbf{r}}.\, \circledcirc_5^w\, .X \rhd_{\overline{\mathbf{r}}} \circledcirc_5^w.X)) \setminus \mathbf{r^i}\end{aligned}$$




Composition for postconditions, as is the case with choice, is virtually identical to its precondition counterpart, and the same goes for the next two agents as well.

$$
\begin{array}{lcl}
{}^{\mathbf{of}}_{w}[\![\mathbf{0}]\!]_{\emptyset} & \triangleq & \mathbf{0} \\
{}^{\mathbf{of}}_{w}[\![\mathbf{1}]\!]_{\emptyset} & \triangleq & \mu X.\, \circledS^{w}_{4}.(\overline{\mathbf{r}}.\, \circledS^{w}_{5}.X + \circledS^{w}_{5}.X) \\
{}^{\mathbf{of}}_{w}[\![\mathsf{b}]\!]_{\{b\}} & \triangleq & \mu X.\, \circledS^{w}_{4}.(b^{w}.\overline{\mathbf{r}}.\overline{b}_{\circledS^{w}_{5}}.\mathbf{0} \rhd \circledS^{w}_{5}.X)
\end{array}
$$




Finally, the actual **output** agents are very similar to the input agents also. The difference is that the output agent's condition is satisfied when the respective output is communicated to it on the internal channel $b^w$. This output is then relayed exactly once from the workflow (it is picked up from here by the performance wrapper and broadcast to the environment).

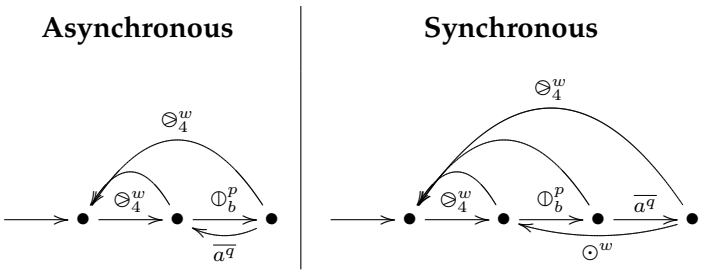
### Connections

Dataflow connections $\rightarrowtail$ are effectively wires which pass data from one part of a workflow to another. There are three types of connection:

- **Passthrough**, which connect a performance output to a performance input (either asynchronous or synchronous);
- **Workflow input**, which connect an input of the workflow to a performance input;
- **Workflow output**, which connect a performance output to a workflow output.

Unlike other semantics, the connection semantics return only two sets, one for workflow inputs and one for workflow outputs. These are used in the workflow semantics to ensure internal dataflow channels are restricted (see Section 7.2.5).

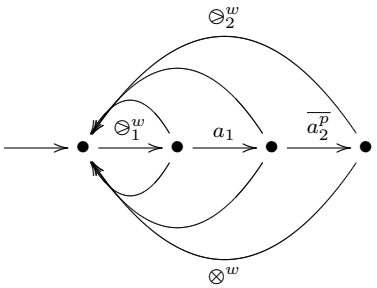
$$
\begin{array}{lcl}
{}^{\mathbf{df}}_{w}[\![p.b \rightarrowtail q.a]\!]^{\{a^q\}}_{\{b^p\}} & = & \mu X.\, \circledS^{w}_{3}.\mu Y.(\textcircled{\scriptsize D}^{p}_{b}.\overline{a^q}.Y \rhd \circledS^{w}_{4}.X) \\
{}^{\mathbf{df}}_{w}[\![p.b \twoheadrightarrow q.a]\!]^{\{a^q\}}_{\{b^p\}} & = & \mu X.\, \circledS^{w}_{3}.\mu Y.(\textcircled{\scriptsize D}^{p}_{b}.\overline{a^q}.\odot^{w}.Y \rhd \circledS^{w}_{4}.X)
\end{array}
$$




The simplest type of connection is the *passthrough*, which simply inputs from the output of a performance via clock $\circledS^{p}_{b}$, and then passes this on to the input of another performance via $\overline{a^q}$. A passthrough is therefore simply a one-place buffer, indeed they provide

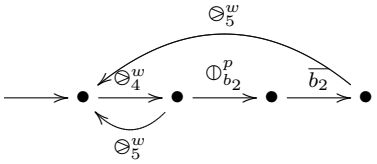
the input buffer for their respective performances. As I described back in Chapter 5 there are two variants of the passthrough, the synchronous and the asynchronous. The only difference between them is that the latter waits for a yield before accepting another input to be passed onto its receiver. This forces the synchronous connection to only pass data on once per phase period, whilst the asynchronous variant will repeatedly pass on. Naturally, passthroughs are only enabled during the execution phase. The input and output sets are populated with the respective performance inputs and outputs.

$$\mathop{\mathbf{df}}_{w}[\![a_1 \rightarrowtail p.a_2]\!]^{\{a_1\}}_{\{a_2^p\}} = \mu X.\, \circledcirc_1^w .(a_1.\overline{a_2^p}.\mathbf{0} \rhd (\circledcirc_2^w.X + \otimes^w.X))$$




Workflow input connections are similarly simple, they take a workflow input from an acceptor agent and pass this onto a performance input. This is stopped either by cancellation, or by the execution phase beginning. The workflow input $a_1$ is returned in the input set.

$$\mathop{\mathbf{df}}_{w}[\![p.b_1 \rightarrowtail b_2]\!]^{\{b_1^p\}}_{\{b_2\}} = \mu X.\, \circledcirc_4^w .(\mathbb{O}^p_{b_1}.\overline{b_2}.\, \circledcirc_5^w .X + \circledcirc_5^w.X)$$




Finally, workflow output connections convey data from performance outputs to workflow outputs. They simply wait for the penultimate workflow clock to tick ($\circledcirc_4^w$) and then either input from the performance via $\mathbb{O}^p_{b_1}$ and in turn pass this on to the workflow output via $b_2$, or else simply exit with $\circledcirc_5^w$ if there is no output available.

$$\begin{array}{rcl} \mathop{\mathbf{df}}_{w}[\![D \cdot D']\!]^{\tilde{a}\cup\tilde{a'}}_{\tilde{b}\cup\tilde{b'}} & \triangleq & \mathop{\mathbf{df}}_{w}[\![D]\!]^{\tilde{a}}_{\tilde{b}} \mid \mathop{\mathbf{df}}_{w}[\![D']\!]^{\tilde{a'}}_{\tilde{b'}} \\ \mathop{\mathbf{df}}_{w}[\![\mathbf{1}]\!]^{\emptyset}_{\emptyset} & \triangleq & \mathbf{0} \end{array}$$

This last part simply joins all the dataflow wires together by parallel composing them and joining together the workflow input and output sets.

#### Dataflow caveats

The Boolean algebra I have adopted for preconditions is heavily dependent on the workflow's abstract time system, be it logical (yield) or physical (RTC). In CCS there is no way of ensuring that, if a possible synchronisation exists in a process, it will be conveyed between the two agents, since all communications are interleaved. As a result optional inputs cannot be implemented, since they would be populated non-deterministically with the readiness signal. However in CaSE$^{ip}$ , and abstract time process calculi in general, signal presence can be detected by maximal progress. Thus if we require a clock tick in a dataflow scope (i.e. a workflow), we are guaranteed that all signals are indeed conveyed.

However, this does mean that we need to seek a trade-off in determining whether or not an optional input will appear. If we force a yield before a given performance executes, it guarantees that all performances unguarded by a yield in the workflow are complete. Normally, this should be adequate since it is unusual for a performance to execute in parallel with other performances it depends on. Unusual that is, until we consider graph based control-flow like *Interleaved Routing* which relies on this. In these circumstances more care is needed when crafting a workflow. For instance, instead of using yield a workflow designer could use time wait, which ensures that all data available at a particular time can be picked up (dataflow is instantaneous, relatively speaking). If we don't force any kind of synchronisation, we end up with the CCS position where optional inputs may not be transmitted, even if they exist. This is the case even in a simple sequential workflow, e.g. $P \fatsemi Q$ – if $P$ outputs an optional input for $Q$ the decision whether it is consumed or not is not deterministic.

It is therefore perhaps better to force an RTC synchronisation in this situation, e.g. $P \fatsemi \circ Q$. Since the RTC can only tick when all possible synchronisations have taken place, this would force all dataflow currently available to be transmitted *before* $Q$ is activated. This is better than a yield, as the data will still be transmitted whilst performances are part way through, whereas yield forces all internal activity to cease. Therefore, this mitigates the problems of optional inputs, though it does drop a unit of time and forces a synchronisation with the parent workflow. Nevertheless this demonstrates why having two forms of time in workflows is so important (besides its use in speculative parallelism).

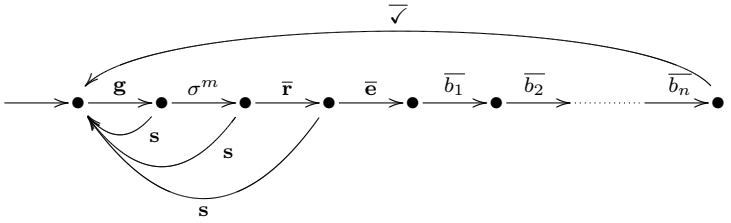
### 7.2.9 Component semantics

Having defined semantics for workflow, control flow and dataflow, I now proceed to give a semantics to the different types of component in Cashew-A . These semantics likewise follow the standard protocol.

$$\begin{array}{lcl} {\scriptstyle \mathbf{pf}}_{p,g}[\![\mathbf{Wf}\ W]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} & \triangleq & {\scriptstyle \mathbf{wf}}_{w,g}[\![W]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \\ {\scriptstyle \mathbf{pf}}_{p,g}[\![\mathbf{CWf}\ W]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} & \triangleq & {\scriptstyle \mathbf{cwf}}_{w,g}[\![W]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \end{array}$$

A workflow performance is a straightforward link to the standard workflow semantics. A compensating workflow performance is the same.

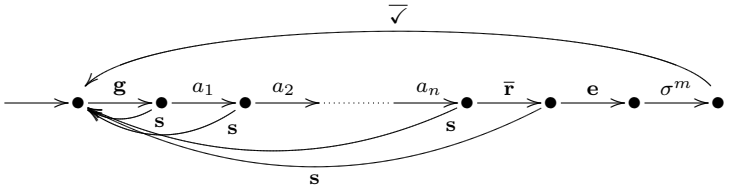
$$\begin{array}{rcl} {\scriptstyle \mathbf{pf}}_{p}[\![\mathbf{Receive}\ m\ \tilde{B}]\!]^{\emptyset,\tilde{b}}_{\{m\},\emptyset} & \triangleq & \mu X.\mathbf{g}.(\sigma^m.\overline{\mathbf{r}}.\mathbf{e}.\mathsf{seq}(\tilde{b}).\overline{\checkmark}.X \rhd_{\mathbf{e}} \mathbf{s}.\mathbf{0}) \\ \text{where } \mathsf{seq}(\tilde{b}).E & = & \begin{cases} E & \text{if } \tilde{b} = \emptyset \\ \overline{b}.\mathsf{seq}(\tilde{b} \setminus \{b\}).E & \text{otherwise} \end{cases} \end{array}$$




A message receive process is activated and then waits for its associated message clock $\sigma^m$ to tick signifying that a message has been received. Once the message arrives the process declares readiness – to reiterate, the message is received *prior* to readiness so that message choice decisions are possible. Once given permission to execute, the receive simply releases all the values contained in the message as dataflow and then terminates. The message parts are propagated via the $\mathsf{seq}$ process constructor, which simply outputs each of the values in turn.

The message clock $\sigma^m$ is renamed to channel $m$ at the top of the workflow (see the workflow semantics in Section 7.2.5). The semantics returns a set of all the outputs the message contains, and the message name in the message input set.

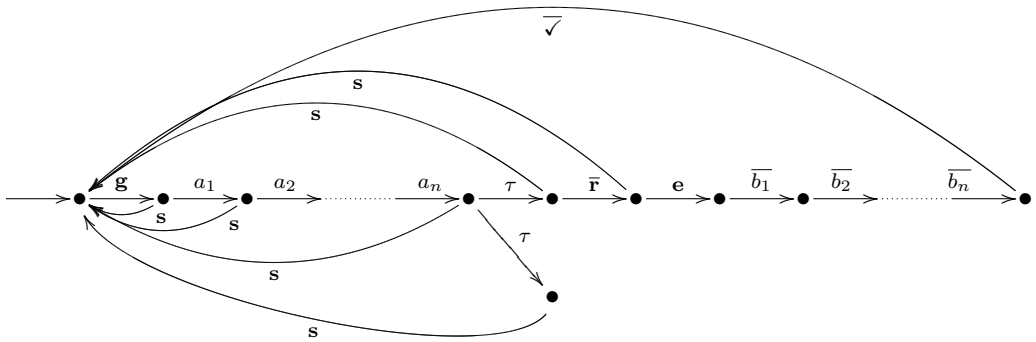
$$\begin{array}{rcl} {\scriptstyle \mathbf{pf}}_{p}[\![\mathbf{Send}\ m\ \tilde{A}]\!]^{\tilde{a},\emptyset}_{\emptyset,\{m\}} & \triangleq & (\mu X.\mathbf{g}.\mathsf{seq}(\tilde{a}).\overline{\mathbf{r}}.\mathbf{e}.\sigma^m.\overline{\checkmark}.X \rhd_{\mathbf{e}} \mathbf{s}.X \\ \text{where } \mathsf{seq}(\tilde{a}).E & = & \begin{cases} E & \text{if } \tilde{a} = \emptyset \\ a.\mathsf{seq}(\tilde{a} \setminus \{a\}).E & \text{otherwise} \end{cases} \end{array}$$




The message send performance is different to a receive in that the message is only sent *after* it has permission to execute. A send process first waits for all the needed data items to populate the message, again using the $\mathsf{seq}$ process constructor. Once every item

is input the send is free to declare readiness as normal. The performance then ticks the message clock, signifying the message transmission and then terminates.

$$
\begin{array}{lcl}
{}^{\mathbf{pf}}_{p}[\![\mathbf{Eval}\ x\ \tilde{A}\ \tilde{B}]\!]^{\tilde{a},\tilde{b}}_{\emptyset,\emptyset} & \triangleq & \mu X.\mathbf{g}.(\mathsf{seqI}(\tilde{a}).(\tau.\overline{\mathbf{r}}.\mathbf{e}.\mathsf{seqO}(\tilde{b}).\overline{\checkmark}.X + \tau.\mathbf{0}) \rhd_{\mathbf{e}} \mathbf{s}.X) \\
\text{where } \mathsf{seqI}(\tilde{a}).E & = & \begin{cases} E & \text{if } \tilde{a} = \emptyset \\ a.\mathsf{seqI}(\tilde{a} \setminus \{a\}).E & \text{otherwise} \end{cases} \\
\mathsf{seqO}(\tilde{b}).E & = & \begin{cases} E & \text{if } \tilde{b} = \emptyset \\ \overline{b}.\mathsf{seqO}(\tilde{b} \setminus \{b\}).E & \text{otherwise} \end{cases}
\end{array}
$$




An evaluation take a list of inputs, performs side-effect free evaluation using them and either produces a list of outputs or stalls. It can be considered analogous to a Haskell function $(\mathtt{a_1} \cdots \mathtt{a_j}) \rightarrow \mathsf{Maybe}\ (\mathtt{b_1} \cdots \mathtt{b_k})$ which may or may not produce a value. It thus consists of a collection of input and output agents, similar to the message send and receive processes. The process is activated and then waits for receipt of the required inputs. An internal choice is then executed which represents the evaluation of the expression producing either a value or nothing. If nothing is produced then the process only accepts s from that point – it has halted. Otherwise, the process proceeds with the readiness protocol as normal and then outputs all the values produced.

$$
\begin{array}{l}
{}^{\mathbf{pf}}_{g,p}[\![\mathbf{GoalTemplate}\ g\ A\ B]\!]^{\tilde{a},\tilde{b}}_{\emptyset,\emptyset} \triangleq \\
\quad (\mu X.\mathbf{g}.\ \otimes^p_1.\lfloor \mathbf{s}.\ \odot^p.X \rfloor \otimes^p_2 (\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\mu Z.\ (\ \tau.\ \odot^p.\otimes^p_3.\mathbf{r^j}.\otimes^p_4.\overline{\checkmark}.X \\
\qquad\qquad\qquad\qquad\qquad\qquad + \tau.\overline{\mathbf{t}}.Z) \\
\qquad\qquad\qquad\qquad\qquad\qquad \rhd_{\mathbf{e}} \mathbf{s}.\ \odot^p.X) \\
\quad \left| \ {}^{\mathbf{gi}}_{p}[\![A]\!]^{\tilde{a}}_{\emptyset}\{\![\mathbf{r} \mapsto \mathbf{r^i}]\!\} \ \right| \ {}^{\mathbf{go}}_{p}[\![B]\!]^{\emptyset}_{\tilde{b}}\{\![\mathbf{r} \mapsto \mathbf{r^j}]\!\}\Big) / [\otimes^p_1, \otimes^p_2, \otimes^p_3, \otimes^p_4, \odot^p] \setminus \{\mathbf{r^i}, \mathbf{r^j}\}
\end{array}
$$

The **GoalTemplate** semantics is similar to an expression evaluation, except it will not halt once its preconditions are satisfied and it uses a full Boolean algebra for preconditions and postconditions (like workflows). Furthermore, since Goals are not instantaneous, time can pass while they execute and thus the Goal execution is represented

as an internal choice of completing the goal or allowing a further RTC tick via the $\mathbf{t}$ channel. The semantics consists of a scheduler and two processes which handle preconditions and postconditions. The scheduler agent follows the normal protocol, and is in many ways similar to the Governor agent of Section 7.2.5, using four clocks to schedule preconditions and postconditions (to a workflow's six clocks). The first clock indicates activation, and allows the preconditions to be fulfilled. The second clock ticks when all the preconditions are fulfilled. The scheduler then requests permission to execute from the environment (the $\mathbf{r^i}$ is just house-keeping), and if this is given it enters a loop.

The loop makes a non-deterministic choice between finishing or sending a $\mathbf{t}$, to indicate the passage of time. This $\mathbf{t}$ is turned into an RTC clock tick in the encapsulated workflow by the performance wrapper (see Section 7.2.6). When the loop exits, $\odot^p$ ticks to cause the precondition process to exit and $\circledcirc_3^p$ ticks to enable the postcondition process. The scheduler then waits for an $\mathbf{r^j}$ to indicate the postconditions are satisfied, ticks the final clock, causing the postcondition process to exit, and sends out ✓.

$$
\begin{array}{lcl}
{}^{\mathbf{gi}}_{p}[\![A_1 \sqcup A_2]\!]^{\tilde{a_1}\cup\tilde{a_2}} & \triangleq & (({}^{\mathbf{gi}}_{p}[\![A_1]\!]^{\tilde{a_1}} \mid {}^{\mathbf{gi}}_{p}[\![A_2]\!]^{\tilde{a_2}})\{\!|\mathbf{r}\mapsto\mathbf{r^i}|\!\} \\
 & & \quad \mid \mu X.\, \circledcirc_1^p.(\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{r^i}.\mathbf{0} \rhd (\circledcirc_2^p.X + \odot^p.X)) \setminus \mathbf{r^i} \\
{}^{\mathbf{gi}}_{p}[\![A_1 \sqcap A_2]\!]^{\tilde{a_1}\cup\tilde{a_2}} & \triangleq & (({}^{\mathbf{gi}}_{p}[\![A_1]\!]^{\tilde{a_1}} \mid {}^{\mathbf{gi}}_{p}[\![A_2]\!]^{\tilde{a_2}})\{\!|\mathbf{r}\mapsto\mathbf{r^i}|\!\} \\
 & & \quad \mid \mu X.\, \circledcirc_1^p.(\mathbf{r^i}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{0} \rhd (\circledcirc_2^p.X + \odot^p.X))) \setminus \mathbf{r^i} \\
{}^{\mathbf{gi}}_{p}[\![\mathbf{0}]\!]^{\emptyset} & \triangleq & \mathbf{0} \\
{}^{\mathbf{gi}}_{p}[\![\mathbf{1}]\!]^{\emptyset} & \triangleq & \mu X.\, \circledcirc_1^p.(\overline{\mathbf{r}}.\, \circledcirc_2^p.X \rhd \odot^p.X) \\
{}^{\mathbf{gi}}_{p}[\![\mathsf{a}]\!]^{\{a\}} & \triangleq & \mu X.\, \circledcirc_1^p.(a.\overline{\mathbf{r}}.\, \circledcirc_2^p.X \rhd \odot^p.X))
\end{array}
$$

$$
\begin{array}{lcl}
{}^{\mathbf{go}}_{p}[\![B_1 \sqcup B_2]\!]_{\tilde{b_1}\cup\tilde{b_2}} & \triangleq & (({}^{\mathbf{go}}_{p}[\![B_1]\!]_{\tilde{b_1}} \mid {}^{\mathbf{go}}_{p}[\![B_2]\!]_{\tilde{b_2}})\{\!|\mathbf{r}\mapsto\mathbf{r^i}|\!\} \\
 & & \quad \mid \mu X.\, \circledcirc_3^p.\mathbf{r^i}.\overline{\mathbf{r}}.(\tau.(\lfloor \mathbf{r^i}.\, \circledcirc_4^p.X \rfloor \circledcirc_4^p(X)) + \tau.\, \circledcirc_4^p.X) \setminus \mathbf{r^i} \\
{}^{\mathbf{go}}_{p}[\![B_1 \sqcap B_2]\!]_{\tilde{b_1}\cup\tilde{b_2}} & \triangleq & (({}^{\mathbf{go}}_{p}[\![B_1]\!]_{\tilde{b_1}} \mid {}^{\mathbf{go}}_{p}[\![B_2]\!]_{\tilde{b_2}})\{\!|\mathbf{r}\mapsto\mathbf{r^i}|\!\} \\
 & & \quad \mid \mu X.\, \circledcirc_3^p.(\mathbf{r^i}.\mathbf{r^i}.\overline{\mathbf{r}}.\, \circledcirc_4^p.X \rhd_{\overline{\mathbf{r}}} \circledcirc_4^p.X)) \setminus \mathbf{r^i} \\
{}^{\mathbf{go}}_{p}[\![\mathbf{0}]\!]_{\emptyset} & \triangleq & \mathbf{0} \\
{}^{\mathbf{go}}_{p}[\![\mathbf{1}]\!]_{\emptyset} & \triangleq & \mu X.\, \circledcirc_3^p.\overline{\mathbf{r}}.\, \circledcirc_4^p.X \\
{}^{\mathbf{go}}_{p}[\![\mathsf{b}]\!]_{\{b\}} & \triangleq & \mu X.\, \circledcirc_3^p.(\overline{\mathbf{r}}.\underline{\overline{b}}_{\circledcirc_4^p}.\, \circledcirc_4^p.X + \circledcirc_4^p.X)
\end{array}
$$

The semantics for Goal preconditions and postconditions are essentially the same as for workflows (see Section 7.2.8), but with a change to the postcondition semantics. Goals are essentially abstract entities, and therefore I assume that all output possibilities are equally likely. Therefore it is necessary to allow a non-deterministic valuation of postconditions. The semantics of $B_1 \sqcup B_2$, after receiving readiness from one side (which is mandatory), makes a non-deterministic choice between allowing readiness from the other (optional) side or not. This means that all output possibilities become possible.

### 7.2.10 Compositionality Problems

The majority of the semantics specified in this Section are by nature compositional, they are given in terms of the semantics of the parts together which some scheduling device which combines them. There are, however, a number of exceptions which make compositionality more difficult to prove. In particular, the semantics of yield and wait are only compositional in the context of a specific workflow context. The reason for this is a long-standing theoretical issue with CaSE(and to a degree CaSE$^{ip}$ ) – clocks names must be globally unique.

$$\frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma} \qquad \frac{E \xrightarrow{\alpha} E'}{E/\sigma \xrightarrow{\alpha} E'/\sigma} \qquad \frac{E \xrightarrow{\rho} E' \quad E \xrightarrow{\sigma}\!\!\!\!\!\rightarrow}{E/\sigma \xrightarrow{\rho} E'/\sigma}\ \rho \neq \sigma$$

In the original CaSE semantics, when a clock is hidden it also forcibly removed from the clock context permanently. Specifically, a clock may not be reused by a process which has already hidden a clock of the same name in a sub-process. The operational rules only allow clocks other than $\sigma$ to tick over $E/\sigma$, even when $E \xrightarrow{\sigma}\!\!\!\!\!\rightarrow$. I am uncertain of the reason for this, but conversations with the author[1] lead me to believe that parts of the CaSE theory would not work without this restriction. Certainly there is a problem with $\Delta$, as if we *were* to allow $\sigma$ to tick over $E/\sigma$, then it would be necessary to establish if there are any $\Delta$s present in $E$. Normally this is done simply by checking if $\sigma$ can also tick in the subprocesses, but clearly this wouldn't work as $\sigma$ ticking in $E$ would prevent $\sigma$ ticking in $E/\sigma$. A possible alternative may be to create a "distinguished" clock $\delta$, which may not be used in the process syntax, but is used to ensure $E$ is not insistent on all clocks. I would treat parametrised $\Delta_\sigma$ as localised to the $\sigma$ hiding boundary in which it exists, thus having no effect on a clock outside.

$$\frac{E \xrightarrow{\delta} \quad E \xrightarrow{\sigma}\!\!\!\!\!\rightarrow}{E/\sigma \xrightarrow{\sigma} E/\sigma}$$

This would potentially fix the problem in CaSE. Of course, I'm not using CaSE, I'm using CaSE$^{ip}$ which doesn't have the same problem. The way I fix it is through the set $\Sigma$, which defines the set of clocks a process stalls explicitly (see Chapter 6, Section 6.3). Since $\Sigma_{E/\sigma}$ does not contain $\sigma$, any parallel process which wishes to tick $\sigma$ will be

[1] Barry Norton

permitted to by the parallel composition rules.

Nevertheless, even though this is true, I am still apprehensive about having missed another theoretical problem. Therefore, for the time being I err on the side of caution and avoid re-using hidden clocks. This is not possible in some instances, for example the semantics of interleaving uses private clocks to decide when all sub-processes have been activated. If it *does* prove to be the case that globally unique clock names really aren't necessary then they can easily be removed from the semantics by removing the name superscripts from all clocks (e.g. $\circledcirc_1^w \mapsto \circledcirc_1$).

Having globally a unique clock name does, unfortunately, lead to implementation problems. For instance it isn't possible to compose two arbitrary control flows and place them in a workflow, as the workflow name must be known beforehand. Therefore their removal, or partial removal, would be of potential benefit.

## 7.3 Compensation Semantics

Having completed the denotation for the normal workflow section of Cashew-A , I now proceed to define the semantics for the compensable fragment from Chapter 5 Section 5.5.3. The original structure for compensation was documented in my earlier paper (Foster, 2007), and abstract time is once again central to the approach. Much of the basic protocol remains unchanged from normal workflows, and many parts of the existing semantics will be reused. The brevity of the section demonstrates that, having given a basic framework for Cashew-A workflows, expanding it with additional features requires minimal work. In particular the dataflow model is largely unchanged. Nevertheless the work in this Section is much less developed and as such is not central to my Thesis. As such I provide only the symbolic semantics and a relatively brief explanation. The uninterested reader can safely skip to Chapter 8. The main purpose of this Section is to show that my approach can easily be extended to handle compensation.

### 7.3.1 Protocol

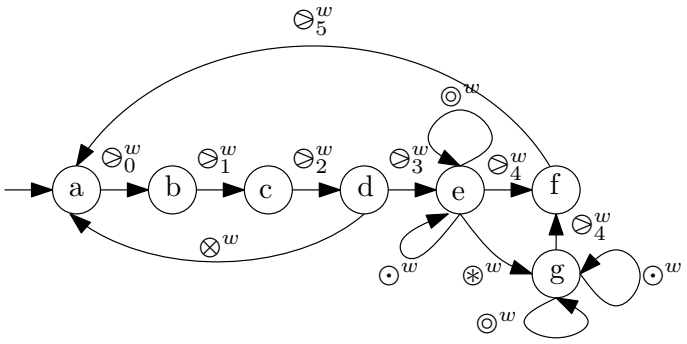



Figure 7.6: Phases of a compensating workflow

The behaviour of the protocol for compensation in Figure 7.6 contains all the standard behaviour for a non-compensating workflow (specifically, the compensable protocol simulates the normal protocol), but adds the ability for a compensation to be scheduled during and after execution of each process. During the execution of a workflow, i.e. the phase between $\circledcirc_3^w$ and $\circledcirc_4^w$, a compensation may be triggered by the raising of an exception indicated by the new clock $\circledast^w$ signalling, moving the workflow into a compensation phase (g). This will only occur once the entire workflow has yielded – a subprocess cannot be broken midway unless it explicitly permits this by use of the compensation yield operator $\Uparrow$. Compensation is orchestrated using the channels $\mathbf{g_c}, \mathbf{r_c}, \mathbf{e_c}$ and $\checkmark_\mathbf{c}$, which are analogous to their non-compensatory counterparts. There is no equivalent of the $\mathbf{s_c}$ channel because the compensation flow cannot contain choices, only atomic performances. A choice in the workflow is compensated for by running the compensations of the branch which executed. Therefore a compensation need never be stopped via $\mathbf{s_c}$. Compensation can only be initiated one time in a workflow.

Unlike in the normal workflow semantics, processes in the compensable semantics remain "live" even after completion. A process only becomes inactive and thus accepting $\mathbf{g}$ again *after* the workflow as a whole signals completion via $\circledcirc_5^w$. Until this point it is possible for compensation to be scheduled via $\mathbf{g_c}$, since compensation of the workflow may begin after the process executes. This is the main reason why a loop cannot be represented in a compensable workflow, as it would be impossible to keep track of its complete status in a static process topology.

As we saw in Section 5.5.3, each sub-process can be composed with a compensation performance. For example $P \div Q$ indicates that $P$ is compensated by performing $Q$. When execution initially begins $Q$ is inactive, it can only be executed once $P$ has successfully completed, thus being "installed". If $P$ has not executed then obviously no compensation at all is needed. On the other hand, if $P$ has partially executed – i.e. it has received the $\mathbf{e}$ command but has yet to output $\checkmark$ – then responsibility for compensation must be taken by the individual components of $P$, to which $\mathbf{r_c}, \mathbf{e_c}$ and $\checkmark_\mathbf{c}$ are simply passed. The only way $P$ can partially complete is if it contains at least one compensation yield. If $P$ has completed then the compensation is installed, and therefore the compensation performance is executed if need-be.

In addition to the compensation readiness protocol, I also supply a method of forcing a transaction to compensate. In the new semantics each workflow (though not all performances) exposes an $\mathbf{x}$ channel, which will force the workflow into the compensation phase, similar to if a $\lightning$ process had been present inside. This exists primarily because of speculative parallelism (see Section 7.3.6), but could potentially also be used to link a workflow with its parent as a single transaction.

In a sense, the compensation semantics are similar in style to that of cCSP (Butler, Hoare and Ferreira, 2005) (see Chapter 2, Section 2.4.1), not only in terms of the compensation strategy, but also in terms of processes used. Calculi like StAC (Chessell et al., 2002) represent the compensation stack by encapsulating it in the transition system, stor-

ing a separate compensation context outside the process being executed. By contrast, in cCSP compensations are contained in the process syntax itself. In the Cashew-A semantics a compensation is represented as an agent which can either be active, in which case it will schedule its associated performance when called, or inactive. This is also the reason why loops may not be represented in the transaction control-flow language $\mathcal{T}$, as compensations must be represented statically.

In terms of the four compensation strategies presented by Bruni, Butler, Ferreira, Hoare, Melgratti and Montanari (2005), the strategy here is broadly *centralised with interruption*, since the entirety of an individual workflow must yield to compensation. Nevertheless, a certain amount of distribution may be achieved by hierarchically arranging compensating workflows and using dataflow to testify whether an enclosed workflow successfully completed or not.

Having described the overall idea behind the semantics, we can now proceed to give them formally.

### 7.3.2 Compensable workflow semantics

The processes in this section follow a very similar timeline to that of a normal workflow. The main difference is the addition of the compensation phases, and machinery needed to deal with this. The timeline of a compensation can be seen in Figure 7.7. It is represents what happens when the control-flow raises an exception somewhere during execution.

$$
\begin{aligned}
&{}^{\mathbf{cwf}}_{w,g}[\![w[A\{C \times D\}B]g]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \\
&\Big( {}^{\mathbf{wac}}_{w}[\![A]\!]^{\tilde{a}} \;\Big|\; {}^{\mathbf{wof}}_{w}[\![B]\!]_{\tilde{b}} \\
&\Big| \Big( {}^{\mathbf{cwcf}}_{w,g}[\![C]\!]^{\tilde{a}_{\mathbf{df}},\tilde{b}_{\mathbf{df}}}_{\tilde{m},\tilde{n}} \;\Big|\; {}^{\mathbf{cdf}}_{w}[\![D]\!]^{\tilde{a}_{\mathbf{df}}}_{\tilde{b}_{\mathbf{df}}} \Big) \{\!\{a \mapsto a^w | a \in \tilde{a}\}\!\}\{\!\{b \mapsto b^w | b \in \tilde{b}\}\!\} \setminus \tilde{b}_{\mathbf{df}} \\
&\Big) \setminus \{a^w | a \in \tilde{a}\} \cup \{b^w | b \in \tilde{b}\}\{\!\{\otimes^w \mapsto \mathbf{s}\}\!\}/[\circledcirc^w_0 \cdots \circledcirc^w_5, \odot^w, \circledast^w]/[\mathbin{\textcircled{\scriptsize D}}^p_a \,|\, a^p \in \tilde{a}_{\mathbf{cf}} \cup \tilde{a}_{\mathbf{df}}] \\
&\quad \{\!\{\sigma^m \mapsto m | m \in \tilde{m}\}\!\}\ \{\!\{\sigma^n \mapsto \overline{n} | n \in \tilde{n}\}\!\}\ \{\!\{\circledcirc^w \mapsto \overline{\mathbf{t}}\}\!\}
\end{aligned}
$$

The top-level workflow semantics of a transaction block is very similar to that of a normal workflow (see Section 7.2.5). The acceptor and offeror semantics is identical, since compensation can only occur when neither process is active. The control flow and dataflow processes are composed in the same way, but have a different semantics. The only other difference is that the compensation clock $\circledast^w$ is hidden in addition to the other workflow clocks. It is hidden after every other clock, as compensation should only occur if it is the only option and therefore has the lowest priority.

$$\llbracket C \rrbracket^{\tilde{a}_{\mathbf{df}},\tilde{b}_{\mathbf{df}}}_{w,g,\tilde{m},\tilde{n}} \triangleq \Big(\llbracket C \rrbracket^{\tilde{a}_{\mathbf{cf}},\tilde{b}_{\mathbf{cf}}}_{w,g,\tilde{m},\tilde{n}} \setminus \{a | a \in \tilde{a}_{\mathbf{cf}} \wedge a \notin \tilde{a}_{\mathbf{df}}\} \cup \{b | b \in \tilde{b}_{\mathbf{cf}} \wedge b \notin \tilde{b}_{\mathbf{df}}\} \{\!|\mathfrak{F}|\!\} \mid \mathsf{CGovernor}_w \Big) \setminus \{\mathbf{g^i}, \mathbf{r^i}, \mathbf{e^i}, \mathbf{s^i}, \checkmark^{\mathbf{i}}, g^{\mathbf{i}}_{\mathbf{c}}, \mathbf{r}^{\mathbf{i}}_{\mathbf{c}}, \mathbf{e}^{\mathbf{i}}_{\mathbf{c}}, \mathbf{s}^{\mathbf{i}}_{\mathbf{c}}, \checkmark^{\mathbf{i}}_{\mathbf{c}}\}$$

The compensable control flow semantics is also similar to normal workflow semantics, the difference being the new governor process, CGovernor which is composed with the compensable control flow. The new compensable channels are restricted to this composition in addition to the normal channels.

$$\begin{aligned}
\mathsf{CGovernor}_w \triangleq\ & \mu X.\underline{\mathbf{g}}_{\mathcal{T}_w}.\langle \circledcirc^w_0 \rangle.\langle \circledcirc^w_1 \rangle_{\otimes}.\overline{\mathbf{g^i}}.\underline{\mathbf{r^i}}_{\mathcal{T}_w}.\langle \circledcirc^w_2 \rangle_{\otimes}.\overline{\mathbf{r}}_{\mathcal{T}_w}.\underline{\mathbf{e}}_{\mathcal{T}_w}.\langle \circledcirc^w_3 \rangle.\overline{\mathbf{e^i}}_{\mathcal{T}_w}. \\
& \mu Y.\ \Big( \underline{\odot^w}_{\mathcal{T}_w \setminus \{\odot^w, \circledast^w\}}.Y \\
& + \checkmark^{\mathbf{i}}.\ (\ \langle \circledcirc^w_4 \rangle.\langle \circledcirc^w_5 \rangle.\overline{\underline{\checkmark}}_{\mathcal{T}_w}.X \\
& \qquad + \mathbf{x}.\langle \circledast^w \rangle.\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{e^i_c}}.\underline{\checkmark^{\mathbf{i}}_{\mathbf{c}}}_{\mathcal{T}_w}.\langle \circledcirc^w_4 \rangle.\langle \circledcirc^w_5 \rangle.\overline{\underline{\checkmark_{\mathbf{c}}}}_{\mathcal{T}_w}.X) \\
& + \circledast^w\ .\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{e^i_c}}.\underline{\checkmark^{\mathbf{i}}_{\mathbf{c}}}_{\mathcal{T}_w}.\langle \circledcirc^w_4 \rangle.\langle \circledcirc^w_5 \rangle.\overline{\underline{\checkmark_{\mathbf{c}}}}_{\mathcal{T}_w}.X \\
& + \mathbf{x}.\langle \circledast^w \rangle.\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{e^i_c}}.\underline{\checkmark^{\mathbf{i}}_{\mathbf{c}}}_{\mathcal{T}_w}.\langle \circledcirc^w_4 \rangle.\langle \circledcirc^w_5 \rangle.\overline{\underline{\checkmark_{\mathbf{c}}}}_{\mathcal{T}_w}.X \\
& \Big)\ \triangleright_{\mathbf{e}}\ \mathbf{s}.\overline{\mathbf{s^i}}_{\mathcal{T}_w}.\langle \otimes^w \rangle.X
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{T}_w &\triangleq \{\circledcirc^w_0 \cdots \circledcirc^w_5, \odot^w, \circledcirc^w, \otimes^w, \circledast^w\} \\
\langle \sigma \rangle.E &\triangleq \underline{\sigma}_{\mathcal{T}_w \setminus \{\sigma\}} \\
\langle \sigma \rangle_{\otimes}.E &\triangleq \underline{\sigma}_{\mathcal{T}_w \setminus \{\sigma, \otimes^w\}}
\end{aligned}$$

The CGovernor process is extended with the ability to orchestrate a compensation. There are two ways compensation can be triggered: by $\circledast^w$ ticking directly (i.e. originating from a throw $\lightning$ within the control flow), or by receipt of an $\mathbf{x}$ from the environment, which causes the CGovernor to force $\circledast^w$. Compensation always begins when $\circledast^w$ is forced during what would normally be a yield by stalling $\odot^w$. When $\circledast^w$ ticks, the CGovernor begins compensation scheduling by first passing $\mathbf{g_c}$ onto the top-level scheduler, and then following the compensation protocol. It ends when $\checkmark^{\mathbf{i}}_{\mathbf{c}}$ is received, indicating the control flow has finished compensating. When this occurs the CGovernor proceeds to tick the two remaining normal phase clocks $\circledcirc^w_4$ and $\circledcirc^w_5$ to allow the offeror to fulfil the workflow's postconditions.

When compensation begins, all workflow output data collected so far is purged, leaving the compensation action to populate the workflow outputs. All the schedulers act in

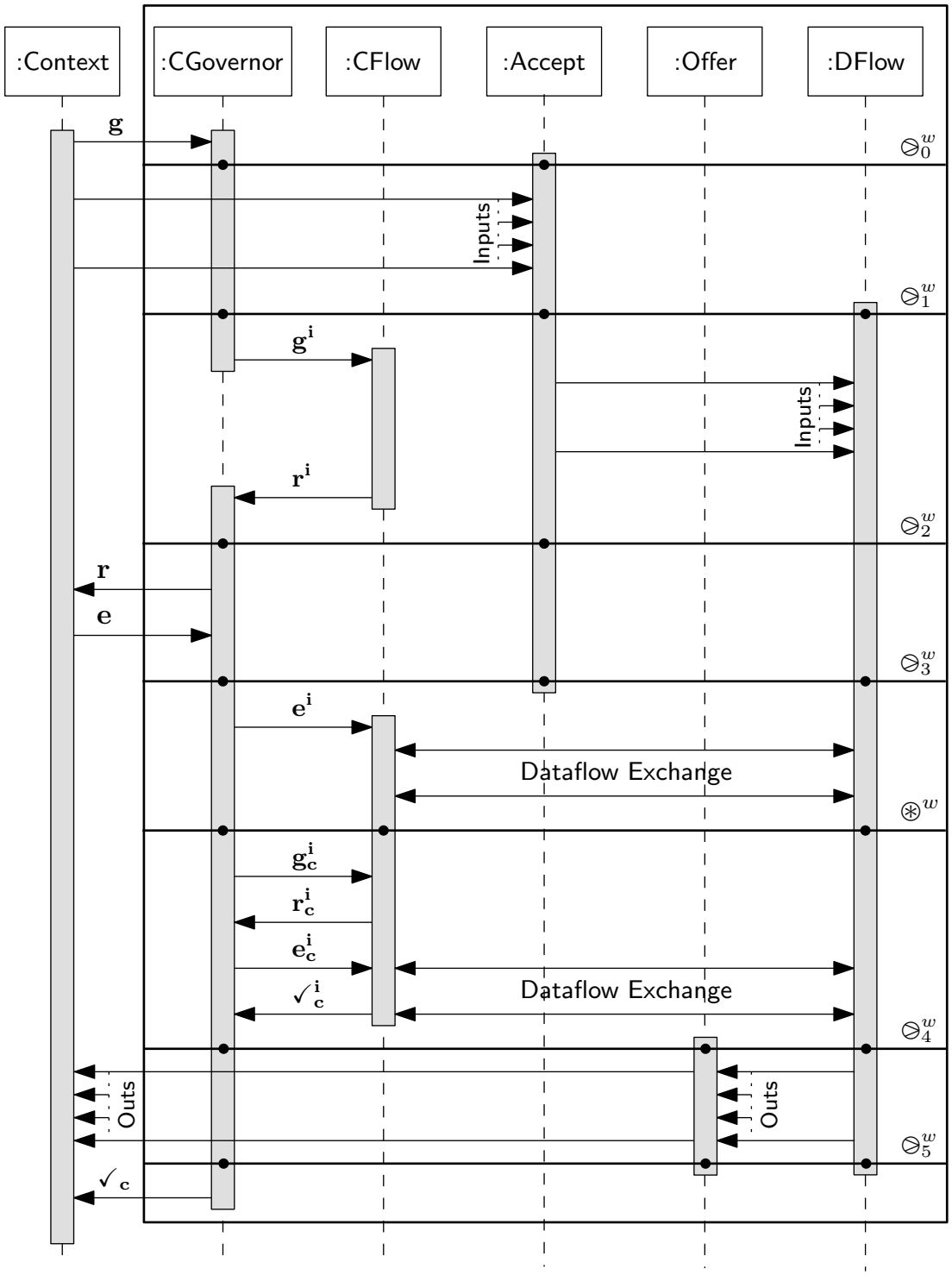



Figure 7.7: The key actors' interaction in a workflow compensation timeline

(**Note**: CFlow = Control Flow, DFlow = Dataflow, Accept = Acceptor, Offer = Offeror, and Outs = Outputs)

reverse, using the compensation channels to run the compensations of their respective workflows. Once compensation has completed, the normal workflow commit phases are begun with the clocks $\circledS_4^w$ and $\circledS_5^w$ guiding the workflow using the new data.

### 7.3.3 Performance Semantics

$$
\begin{aligned}
{}^{\mathbf{cwf}}_{w,g}[\![p[P]]\!]^{\{a^p|a\in\tilde{a}\},\{b^p|b\in\tilde{b}\}}_{\tilde{m},\tilde{n}} \triangleq\ & \left({}^{\mathbf{pf}}_{p,g_{\mathbf{i}}}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}}\right. \\
& \mid \prod_{b\in\tilde{b}} \mu X.\ \circledS_3^w\ .(\mu Y.\underline{b}_{\mathbb{O}_b^p}.(\mathbb{O}_b^p.Y + \circledS_5^w.X + \odot^w.Y) + \circledS_5^w.X) \\
& \mid \mu X.(\circledcirc^w)^{g_{\mathbf{i}}/g}.\mathbf{t}.X \\
& \mid \mu X.\mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\checkmark_{\mathbf{c}}}.X \\
& \left.\right)\backslash\tilde{b}\setminus\{\mathbf{t},\checkmark^{\mathbf{i}}\}\{\!\{a\mapsto a^p|a\in\tilde{a}\}\!\}\{\!\{\checkmark_{\mathbf{c}}\mapsto\checkmark\}\!\}\setminus x
\end{aligned}
$$

The performance wrapper in the context of a compensable workflow is very similar to the performance wrapper in a normal workflow. Since the enclosed performance cannot be compensated by the enclosing workflow a dummy compensation agent is added, which ensures the process follows the protocol. Therefore, a compensable workflow in a performance has no effect on and is not affected by the workflow in which it operators. However, if the enclosed workflow does compensate it can indicate this by its dataflow outputs.

$$
\begin{aligned}
{}^{\mathbf{cwf}}_{w,g}[\![p[P]_{\mathbf{T}}]\!]^{\{a^p|a\in\tilde{a}\},\{b^p|b\in\tilde{b}\}}_{\tilde{m},\tilde{n}} \triangleq\ & \left({}^{\mathbf{pf}}_{p,g_{\mathbf{i}}}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}}\right. \\
& \mid \prod_{b\in\tilde{b}} \mu X.\ \circledS_3^w\ .(\mu Y.\underline{b}_{\mathbb{O}_b^p}.(\mathbb{O}_b^p.Y + \circledS_5^w.X + \odot^w.Y) + \circledS_5^w.X) \\
& \mid \mu X.(\circledcirc^w)^{g_{\mathbf{i}}/g}.\mathbf{t}.X \\
& \mid \mu X.\mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\checkmark_{\mathbf{c}}}.X \\
& \left.\right)\backslash\tilde{b}\setminus\{\mathbf{t},\checkmark^{\mathbf{i}}\}\{\!\{a\mapsto a^p|a\in\tilde{a}\}\!\}
\end{aligned}
$$

A compensable performance wrapper is identical to a normal performance wrapper, except the $\mathbf{x}$ channel isn't restricted. It is used exclusively in the context of a compensable workflow, where the $\mathbf{x}$ channel may be required to force cancellation of an encapsulated performance. Also the $\checkmark_{\mathbf{c}}$ channel is not renamed, as it is used to distinguish whether compensation took place or not.

### 7.3.4 Dataflow semantics

$$
\begin{array}{rcl}
{\scriptstyle w}^{\mathbf{cdf}}[\![p.b \twoheadrightarrow q.a]\!]^{\{a^q\}}_{\{b^p\}} & \triangleq & \mu X.\, \circledS^w_3 .\mu Y.(\textcircled{\scriptsize !}^p_b.\overline{a^q}.Y) \rhd \circledS^w_4.X) \\
{\scriptstyle w}^{\mathbf{cdf}}[\![p.b \twoheadrightarrow q.a]\!]^{\{a^q\}}_{\{b^p\}} & \triangleq & \mu X.\, \circledS^w_3 .\mu Y.(\textcircled{\scriptsize !}^p_b.\overline{a^q}.(\odot^w.Y + \circledast^w.Y) \rhd \circledS^w_4.X) \\
{\scriptstyle w}^{\mathbf{cdf}}[\![a \rightarrowtail p.a']\!]^{\{a\}}_{\emptyset} & \triangleq & \mu X.\, \circledS^w_0 .a.(\overline{a'^p}.\mathbf{0} \rhd (\circledS^w_2.X + \otimes^w.X)) \\
{\scriptstyle w}^{\mathbf{cdf}}[\![p.b \rightarrowtail b']\!]_{\{b\}} & \triangleq & \mu X.\, \circledS^w_3 .(b'^p.\mu Y.(\odot^w.b'^p.Y + \circledast^w.b'^p.Y) \\
 & & \quad \rhd \circledS^w_4 .(\overline{b}.\, \circledS^w_5 .X + \circledS^w_5.X)) \\
{\scriptstyle w}^{\mathbf{cdf}}[\![D \cdot D']\!]^{\tilde{a} \cup \tilde{a'}}_{\tilde{b} \cup \tilde{b'}} & \triangleq & {\scriptstyle w}^{\mathbf{cdf}}[\![D]\!]^{\tilde{a}}_{\tilde{b}} \mid {\scriptstyle w}^{\mathbf{cdf}}[\![D']\!]^{\tilde{a'}}_{\tilde{b'}} \\
{\scriptstyle w}^{\mathbf{cdf}}[\![\mathbf{1}]\!]^{\emptyset}_{\emptyset} & \triangleq & \mathbf{0}
\end{array}
$$

The dataflow model is virtually unchanged from Section 7.2.8. The only difference is that connections, in addition to clearing on yield, also clear on the compensation clock to ensure that only new data is using during compensation. The precondition and post-condition semantics are identical to those in Section 7.2.8 and thus are not reproduced.

### 7.3.5 Control Flow

Most of the control flow semantics are very similar to their non-compensating counterparts, but as with the CGovernor they are augmented with facilities for compensation. We start by considering the simplest four, skip, halt, compensation yield and throw.

#### Basic Agents

$$
\begin{array}{rcl}
{\scriptstyle w,g}^{\mathbf{ccf}}[\![\epsilon]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.(\mathbf{g}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\checkmark}.(\circledS^w_4.X + \mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\checkmark_{\mathbf{c}}}.X) \rhd_{\mathbf{e}} \mathbf{s}.X) \\
{\scriptstyle w,g}^{\mathbf{ccf}}[\![\delta]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.\mathbf{g}.\mathbf{s}.X \\
{\scriptstyle w,g}^{\mathbf{ccf}}[\![\Uparrow]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.(\mathbf{g}.\overline{\mathbf{r}}.\mathbf{e}.(\odot^w.\overline{\checkmark}.(\circledS^w_4.X + \mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\checkmark_{\mathbf{c}}}.X) + \circledast^w.\mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\checkmark_{\mathbf{c}}}.X) \\
 & & \quad \rhd_{\mathbf{e}} \mathbf{s}.X) \\
{\scriptstyle w,g}^{\mathbf{ccf}}[\![\lightning]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} & \triangleq & \mu X.\mathbf{g}.\overline{\mathbf{r}}.\mathbf{e}.\underline{\circledast^w}_{\odot^w}.\mathbf{g_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\checkmark_{\mathbf{c}}.X \rhd_{\mathbf{e}} \mathbf{s}.X \\
{\scriptstyle w,g}^{\mathbf{ccf}}[\![\circ P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} & \triangleq & \Big({\scriptstyle w,g}^{\mathbf{ccf}}[\![P]\!]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \{\!\{\mathfrak{F}\}\!\} \\
 & & \quad \mid \mu X.\mathbf{g}.(\circledcirc^w.\overline{\mathbf{g^i}}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.(\checkmark^{\mathbf{i}}.(\circledS^w_4.X \\
 & & \qquad + \mathbf{g_c}.\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e^i_c}}.\checkmark^{\mathbf{i}}_{\mathbf{c}}.\overline{\checkmark_{\mathbf{c}}}.X) \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s^i}}.X \\
 & & \qquad + \mathbf{g_c}.\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e^i_c}}.\checkmark^{\mathbf{i}}_{\mathbf{c}}.\overline{\checkmark_{\mathbf{c}}}.X\Big) \setminus \mathfrak{R}
\end{array}
$$

The semantics of the basic agents are changed to allow compensation at the appropriate places. Skip $\epsilon$, after issuing the completion signal via $\overline{\checkmark}$, does not immediately

return to the initial state, but waits for the workflow as a whole to finish (signalled by $\circledcirc_4^w$). Until then, the process may begin compensation via communication of $\mathbf{g_c}$. Skip then executes the compensation protocol, which of course is a dummy run like the forward flow, ending with a $\checkmark_{\mathbf{c}}$ which returns it to the initial state. Halt $\delta$ is identical to the original semantics as it can never reach a state where compensation is possible.

Compensation yield ($\Uparrow$) is similar to skip, except that for it to complete the yield clock must tick, between $\mathbf{e}$ and $\checkmark$. It is thus different to normal yield as the yield clock ticks after the process execution has started and not before. If the compensation clock ticks instead of the yield clock, the process executes compensation, the same as skip. Also in common with skip, after executing it must wait for the commit clock before sending $\checkmark$, should compensation be triggered.

Throw $\lightning$ is likewise similar to compensation yield, the difference being that it *forces* $\circledast^w$ to tick after receiving $\mathbf{e}$ by holding up the yield clock $\odot^w$. This means the compensation phase must start next time the workflow offers to yield. When it does, throw simply performs a dummy compensation, as for skip.

Finally, the wait operator as in a normal workflow enables its parameter when a unit of time has passed, indicated by $\circledcirc^w$ ticking. It also provides additional machinery for passing compensation scheduling communications onto the enclosed performance.

**Sequence**

$$
\begin{array}{l}
{\scriptstyle \mathbf{ccf}}\llbracket P \fatsemi Q \rrbracket^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\
\quad \left( {\scriptstyle \mathbf{ccf}}\llbracket P \rrbracket^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \{\!\{\mathfrak{F}\}\!\} \,\middle|\, {\scriptstyle \mathbf{ccf}}\llbracket Q \rrbracket^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \{\!\{\mathfrak{G}\}\!\} \right| \\
\quad \mu X.\mathbf{g}.\overline{\mathbf{g^i}}.(\mathbf{r^i}.\bar{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}.\ (\checkmark^{\mathbf{i}}.\overline{\mathbf{g^j}}.\mathbf{r^j}.\overline{\mathbf{e^j}}.\ (\checkmark^{\mathbf{j}}.\overline{\checkmark}.\ (\circledcirc_4^w.X \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \mathsf{CompensateQthenP}_w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \mathsf{CompensateQthenP}_w) \\
\qquad\qquad\qquad\qquad + \mathsf{CompensateP}_w) \\
\quad \rhd_{\mathbf{e}} s.\overline{\mathbf{s^i}}.X)) \setminus \mathfrak{R} \\
\qquad \mathsf{CompensateP}_w \triangleq \circledast^w.\mathbf{g_c}.\overline{\mathbf{g_c^i}}.\mathbf{r_c^i}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e_c^i}}.\checkmark_{\mathbf{c}}^{\mathbf{i}}.\overline{\checkmark_{\mathbf{c}}}.X \\
\mathsf{CompensateQthenP}_w \triangleq \circledast^w.\mathbf{g_c}.\overline{\mathbf{g_c^j}}.\mathbf{r_c^j}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e_c^j}}.\checkmark_{\mathbf{c}}^{\mathbf{j}}.\overline{\mathbf{g_c^i}}.\mathbf{r_c^i}.\overline{\mathbf{e_c^i}}.\checkmark_{\mathbf{c}}^{\mathbf{i}}.\overline{\checkmark_{\mathbf{c}}}.X
\end{array}
$$

There are two possible compensations which sequence may need to do, compensation of $P$, or compensation of $Q$ followed by compensation of $P$, depending on when compensation was triggered. These two strategies are encapsulated in the definitions of $\mathsf{CompensateP}_w$ and $\mathsf{CompensateQthenP}_w$. If execution has begun, but $\checkmark^{\mathbf{i}}$ has not yet been received then the former is run, otherwise the latter is run.

Compensable sequence proceeds to follow the normal scheduling protocol, but when waiting for the process completion signal $\checkmark^{\mathbf{i}}$ there is the additional option of performing $\mathsf{CompensateP}_w$. This process is activated by $\circledast^w$ ticking, and follows the compensating protocol, sending compensation signals to $P$ only. If $P$ successfully completes, the next possibility for compensation is when the scheduler is waiting for $\checkmark^{\mathbf{j}}$, during the execution of $Q$. If compensation is raised here then $\mathsf{CompensateQthenP}_w$ is activated, which also follows the compensation protocol, but applies compensation to $Q$ followed by $P$. Finally, if both $P$ and $Q$ complete, but the workflow as a whole hasn't signalled completion via $\circledcirc_4^w$, then if compensation is triggered $\mathsf{CompensateQthenP}_w$ is also activated, since both must be compensated.

### Choice

$$
\begin{array}{l}
{\scriptstyle \mathbf{ccf}}_{w,g}[\![P \oplus Q]\!]^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\
\quad \Big( {\scriptstyle \mathbf{ccf}}_{w,g}[\![P]\!]^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \{\!\{\mathfrak{F}\}\!\} \;\Big|\; {\scriptstyle \mathbf{ccf}}_{w,g}[\![Q]\!]^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \{\!\{\mathfrak{G}\}\!\} \;\Big|\; \mu X.\mathbf{g}.\overline{\mathbf{g}^{\mathbf{i}}}.\overline{\mathbf{g}^{\mathbf{j}}}. \\
\quad (\quad \mathbf{r}^{\mathbf{i}}.\overline{\mathbf{s}^{\mathbf{j}}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{i}}}.(\checkmark^{\mathbf{i}}.\overline{\checkmark}.\circledcirc_5^w.X \rhd_{\circledcirc_5^w} \circledast^w.\mathbf{g_c}.\overline{\mathbf{g}^{\mathbf{i}}_{\mathbf{c}}}.\mathbf{r}^{\mathbf{i}}_{\mathbf{c}}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{i}}_{\mathbf{c}}}.\checkmark^{\mathbf{i}}_{\mathbf{c}}.\overline{\checkmark_{\mathbf{c}}}.X) \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s}^{\mathbf{i}}}.X) \\
\quad + \quad \mathbf{r}^{\mathbf{j}}.\overline{\mathbf{s}^{\mathbf{i}}}.(\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{j}}}.(\checkmark^{\mathbf{j}}.\overline{\checkmark}.\circledcirc_5^w.X \rhd_{\circledcirc_5^w} \circledast^w.\mathbf{g_c}.\overline{\mathbf{g}^{\mathbf{j}}_{\mathbf{c}}}.\mathbf{r}^{\mathbf{j}}_{\mathbf{c}}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{j}}_{\mathbf{c}}}.\checkmark^{\mathbf{j}}_{\mathbf{c}}.\overline{\checkmark_{\mathbf{c}}}.X) \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s}^{\mathbf{j}}}.X) \\
\quad + \quad \mathbf{s}.\overline{\mathbf{s}^{\mathbf{i}}}.\overline{\mathbf{s}^{\mathbf{j}}}.X)\Big) \setminus \mathfrak{R}
\end{array}
$$

Choice compensates only the branch which has (partially) executed. As in sequence, a compensation can be triggered between the sending of $\mathbf{e}^{\mathbf{i}}/\mathbf{e}^{\mathbf{j}}$ and receiving of $\checkmark^{\mathbf{i}}/\checkmark^{\mathbf{j}}$, or afterwards. Compensation is triggered, as before, by $\circledast^w$ ticking at this point, which causes the scheduler to compensate the side which was executing. Obviously the other side wasn't started, so it is left alone.

### Synchronous Parallel Composition

$$
\begin{array}{l}
{\scriptstyle \mathbf{ccf}}_{w,g}[\![P \mid Q]\!]^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\
\quad \Big( \Big( {\scriptstyle \mathbf{ccf}}_{w,g}[\![P]\!]^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \;\Big|\; {\scriptstyle \mathbf{ccf}}_{w,g}[\![Q]\!]^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \Big) \{\!\{\mathfrak{F}\}\!\} \\
\quad \mid \mu X.\mathbf{g}.\overline{\mathbf{g}^{\mathbf{i}}}.\overline{\mathbf{g}^{\mathbf{i}}}.\quad (\mathbf{r}^{\mathbf{i}}.\mathbf{r}^{\mathbf{i}}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e}^{\mathbf{i}}}.\overline{\mathbf{e}^{\mathbf{i}}}.\quad (\checkmark^{\mathbf{i}}.\checkmark^{\mathbf{i}}.\overline{\checkmark}.\circledcirc_5^w.X \\
\qquad\qquad\qquad\qquad\qquad\qquad \rhd_{\circledcirc_5^w} \circledast^w.\mathbf{g_c}.\overline{\mathbf{g}^{\mathbf{i}}_{\mathbf{c}}}.\overline{\mathbf{g}^{\mathbf{i}}_{\mathbf{c}}}.\mathbf{r}^{\mathbf{i}}_{\mathbf{c}}.\mathbf{r}^{\mathbf{i}}_{\mathbf{c}}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e}^{\mathbf{i}}_{\mathbf{c}}}.\overline{\mathbf{e}^{\mathbf{i}}_{\mathbf{c}}}.\checkmark^{\mathbf{i}}_{\mathbf{c}}.\checkmark^{\mathbf{i}}_{\mathbf{c}}.\checkmark_{\mathbf{c}}.X) \\
\qquad\qquad\qquad \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s}^{\mathbf{i}}}.\overline{\mathbf{s}^{\mathbf{i}}}.X)\Big) \setminus \mathfrak{R}
\end{array}
$$

The compensation semantics for synchronous parallel composition follows the same pattern, but both sides are compensated in parallel.

**Compensation**

$$
\begin{array}{l}
{\scriptstyle \mathbf{cwf}}_{w,g}[\![P \div Q]\!]^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\
\quad \Big( {\scriptstyle \mathbf{ccf}}_{w,g}[\![P]\!]^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \{\!\{\mathfrak{F}\}\!\} \;\Big|\; {\scriptstyle \mathbf{ccf}}_{w,g}[\![Q]\!]^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \{\!\{\mathfrak{G}\}\!\} \;\Big| \\
\quad \mu X.\mathbf{g}.(\overline{\mathbf{g^i}}.\mathbf{r^i}.\overline{\mathbf{r}}.\mathbf{e}.\overline{\mathbf{e^i}}. \;\; (\checkmark^{\mathbf{i}}.\overline{\checkmark}.(\ominus_5^w.X + \circledast^w.(\mathbf{g_c}.\overline{\mathbf{g^j}}.\mathbf{r^j}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e^j}}.\checkmark^{\mathbf{j}}.\overline{\checkmark_{\mathbf{c}}}.X + \ominus_5^w.X))) \\
\qquad\qquad + \circledast^w .\mathbf{g_c}.\overline{\mathbf{g^i_c}}.\mathbf{r^i_c}.\overline{\mathbf{r_c}}.\mathbf{e_c}.\overline{\mathbf{e^i_c}}.\overline{\checkmark^{\mathbf{i}}_{\mathbf{c}}}.X \\
\qquad \rhd_{\mathbf{e}} \mathbf{s}.\overline{\mathbf{s^i}}.X)\Big) \setminus \mathfrak{R}
\end{array}
$$

Apart from compensation yield, compensation composition is the only new operator in the language. It composes a compensable process $P$ on the left-hand side with a non-compensable performance $Q$ on the right-hand side, which acts as the compensation action. The compensation performance $Q$ is only executed if $P$ has executed and completed, as is normal for this kind of operator. If execution of $P$ is part-way through, the compensation has not been "installed" and therefore it is assumed that $P$ contains other compensations to handle it internally. For instance in $(A \div A' \fatsemi B \div B') \div C'$, compensation performance $C'$ is only executed if both $A$ and $B$ have finished, otherwise the encapsulated compensation performances are executed.

The compensation agent therefore passes the readiness protocol signals onto $P$ as normal. If compensation is raised between $\mathbf{e^i}$ being sent and $\checkmark^{\mathbf{i}}$ being received back, then $\mathbf{g^i_c}$ and the other compensation signals are passed on to $P$ as before. If, however, $\checkmark^{\mathbf{i}}$ has been received, the compensation is installed and therefore requests to compensate will be turned into regular protocol signals ($\mathbf{r^j}$, etc.) and communicated to $Q$ which will cause the compensation action to execute.

### 7.3.6 Speculative Parallelism

The consideration of an abstract time semantics for speculative parallelism raises a number of interesting and important questions, not least with regard to my choice of calculus. As a result I devote an entire section to discussing it.

The basic idea of speculative parallelism in this setting is this: a number of compensable workflows are run in parallel, all of which achieve the same goal. One of them will complete first, and thus will be the successful method, or the "winner". The remaining, partially complete threads are compensated and the outer workflow continues. The problem is, how do we give this a semantics, or more specifically how do we, in an abstract time setting, decide which of the executed threads completes first? The answer is to use the RTC which I have taken pains to include. In fact, speculative parallelism was my original motivation for including a real-time timing system.

It is important that each thread receives equal opportunity to execute to completion. If I were simply to allow use of a similar semantics to the standard parallel operator this

wouldn't be directly possible, as the receipt of the final ✓ from one branch or another is entirely non-deterministic. Quite simply, without some form of time there is no way of measuring which branch completed first. It is necessary therefore to have a notion of "faster than" in order to ensure the winner is picked. It is possible to do this without an RTC using the yield clock, but yield is only meaningful within the context of a specific workflow – it cannot be used as an absolute measure. Thus, the only way to do this fairly is via a real-time model, so that the enclosing workflow can ensure the transactions are lock-stepped using a common measure. In fact, with the addition of the time mediators which we have already seen in Section 7.3.3, speculative parallelism becomes very simple.

$$
\begin{array}{l}
{\scriptstyle \mathbf{ccf}}\!\left[\!\!\left[ |*| \vec{P} \right]\!\!\right]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \triangleq \\
\quad \Big( {\scriptstyle \mathbf{sp}}\!\left[\!\!\left[ \vec{P} \right]\!\!\right]^{\tilde{a},\tilde{b}}_{\tilde{m},\tilde{n}} \;\Big|\; \mu X.\ \ \underline{\mathbf{g}}_{\sigma}.\sigma.\sigma.\underline{\overline{\mathbf{r}}}_{\{\sigma,\rho\}}.\underline{\mathbf{e}}_{\{\sigma,\rho\}}.\sigma. \\
\qquad\qquad \mu Y.\ (\ \checkmark^{\mathbf{i}}_{\mathbf{c}}.Y \\
\qquad\qquad\quad + \checkmark^{\mathbf{i}}.\mu Z.(\overline{\mathbf{x}^{\mathbf{i}}}.Z + \checkmark^{\mathbf{i}}_{\mathbf{c}}.Z + \underline{\rho}_{\sigma}.\underline{\overline{\checkmark}}_{\sigma}.X) \\
\qquad\qquad\quad + \underline{\circledast^{w}_{\odot^{w}}}.\underline{\mathbf{g}_{\mathbf{c}}}_{\sigma}.\underline{\overline{\mathbf{r}_{\mathbf{c}}}}_{\sigma}.\underline{\mathbf{e}_{\mathbf{c}}}_{\sigma}.\mu Z.(\overline{\mathbf{x}^{\mathbf{i}}}.Z + \underline{\rho}_{\sigma}.\underline{\checkmark_{\mathbf{c}}}_{\sigma}.X)) \\
\qquad\qquad \rhd_{\mathbf{e}}\ \mathbf{s}.\underline{\rho}_{\sigma}.X\Big)\ /\sigma/\rho
\end{array}
$$

$$
\begin{array}{l}
{\scriptstyle \mathbf{sp}}\!\left[\!\!\left[ P : \vec{P} \right]\!\!\right]^{\tilde{a}_{\mathbf{i}} \cup \tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{i}} \cup \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{i}} \cup \tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{i}} \cup \tilde{n}_{\mathbf{j}}} \triangleq \\
\quad \Big( {\scriptstyle \mathbf{ccf}}[\![P]\!]^{\tilde{a}_{\mathbf{i}}, \tilde{b}_{\mathbf{i}}}_{\tilde{m}_{\mathbf{i}}, \tilde{n}_{\mathbf{i}}} \{\!\{\mathfrak{F}\}\!\} \;\Big|\; \mu X.\sigma.\overline{\mathbf{g}^{\mathbf{i}}}.(\mathbf{r}^{\mathbf{i}}.\sigma.\sigma.\overline{\mathbf{e}^{\mathbf{i}}}.X \rhd_{\overline{\mathbf{e}^{\mathbf{i}}}} \rho.\overline{\mathbf{s}^{\mathbf{i}}}.X)\Big) \setminus \{\mathbf{g}^{\mathbf{i}}, \mathbf{r}^{\mathbf{i}}, \mathbf{e}^{\mathbf{i}}\} \;\Big|\; {\scriptstyle \mathbf{sp}}\!\left[\!\!\left[ \vec{P} \right]\!\!\right]^{\tilde{a}_{\mathbf{j}}, \tilde{b}_{\mathbf{j}}}_{\tilde{m}_{\mathbf{j}}, \tilde{n}_{\mathbf{j}}} \\
\\
{\scriptstyle \mathbf{sp}}[\![\mathbf{Nil}]\!]^{\emptyset,\emptyset}_{\emptyset,\emptyset} \triangleq \mathbf{0}
\end{array}
$$

The semantics is given is two parts. The first part is the central scheduler which activates and readies the transactions via a pair of private clocks which act as proxies for the various channels. The second part is the scheduler which is composed with each individual transaction and takes care of supplying readiness conditions, but nothing else.

The main scheduler is akin to a mini-governor agent, using two private clocks to synchronise the processes representing the parallel branches. It first receives activation via g and then waits for two $\sigma$ ticks before continuing. In this context the clock $\sigma$ is used to begin and end the process of allowing the parallel branches to fulfil their preconditions (all parallel branches must become ready). The parallel composed branch schedulers each wait for the first $\sigma$ and upon observing it send a $\mathbf{g}^{\mathbf{i}}$ to their respective process, and if $\mathbf{r}^{\mathbf{i}}$ is returned, they allow $\sigma$ to tick the second time. Once this happens, indicating all branches are ready, the main scheduler engages in the normal process of negotiating

permission to execute from the environment. If permission is received, $\sigma$ ticks for the third time which allows the branches to begin execution. Each branch's scheduler sends an $\mathbf{e^i}$ to its respective branch and they start.

The main scheduler waits until one of three things happens:

- One of the branches outputs a $\checkmark^{\mathbf{i}}_{\mathbf{c}}$, meaning the branches attempted to complete but failed and thus compensated. Clearly this should not be treated as a successful completion, and therefore this option simply causing the scheduler to continue waiting for another signal;
- One of the branches outputs a $\checkmark^{\mathbf{i}}$, meaning that the "winner" has been declared. If this happens the scheduler repeatedly sends out $\mathbf{x^i}$ to the other branches, causing them to cancel. It also repeatedly inputs $\checkmark^{\mathbf{i}}_{\mathbf{c}}$ to receive completion signals from the compensated branches. Once all the branches have finished compensating $\rho$ will tick and the scheduler finally outputs $\checkmark$ to indicate completion.
- The compensation clock $\circledast^{w}$ ticks to indicate the workflow has entered the compensation phase. This being the case, basically the same strategy as the previous option is followed, but the compensation protocol is first honoured.

Notice that we do not have to specifically mention the RTC, fairness is automatically enforced by the time mediators and thus we know that whichever transaction responds with $\checkmark$ first is the fastest.

### 7.3.7 Evaluation

The semantics of Cashew-A follows a broadly *centralised* approach to compensation. A participant in a workflow may raise an exception, which will force the workflow into compensation mode when the next yield occurs. Naturally, if no yields are present the parallel processes will continue to completion (or until they too fail) and only then will compensation occur. Thus, referring to the four compensation strategies from Bruni, Butler, Ferreira, Hoare, Melgratti and Montanari (2005) (see also Chapter 2, Section 2.4.1), my semantics are immediately capable of both *No interruption and centralised compensation* and *Coordinated interruption*. Furthermore, if the parallel processes of a transaction are wrapped in performances, then they will act independently of the parent transaction, compensating when needed. If a compensation does occur, this can be indicated to the parent process by the dataflow postcondition, e.g. $\mathsf{success} \sqcup \mathsf{failure}$.

In this context *Distributed interruption* is complicated to implement. The primary reason for this is data flow, as if it were possible to start compensating without synchronisation, there would be data flow relating to both forward and compensation flow in the same scope. Therefore, a better approach may be to allow child workflows to be compensated in a distributed manner. This would require having a semantics which would allow a workflow to send an interrupt signal to all its children. Part of this is already

present, since there are interruptible performances $\mathcal{P}_{\mathcal{T}}$. It is certainly possible within this semantic framework, but time constraints do not permit further extension.

## 7.4 Conclusion

In this Chapter I gave a compositional operational semantics to Cashew-A in the form of a CaSE$^{ip}$ denotation. The aim of this exercise on the one hand was to give Web service composition a semantics. On the other hand I have sought to show that CaSE$^{ip}$ is an ideal calculus for representing this type of component system. Clearly Cashew-A is well featured, and is comparable to modern workflow languages. Therefore, with additional work CaSE$^{ip}$ can be expected to provide a basis for Web service composition. Furthermore through the use of common protocols, Cashew-A's semantic model is highly extensible. Additional language constructs can easily be added, so long as they conform to this protocol.

It is also the case, as I indicated in Section 7.2.10 that there are some compositionality problems related to naming of clocks. If it is possible that clocks can be anonymous in a workflow then this would give a great deal of additional scope to the language.

In the next Chapter I will provide a partial implementation of non-compensable fragment of this semantics in order to demonstrate its viability.

# Part III

# Implementation

# Chapter 8

# Implementation of CaSE$^{ip}$

*In this Chapter I will describe, step-by-step, how the CaSE$^{ip}$ process calculus can be implemented in the purely functional programming language, **Haskell**. The purpose of this exercise is to demonstrate that the formal underpinnings for Cashew-A can be represented and used to form the basis of a service composition engine. I will begin with a basic CCS process calculus model, and describe how the simulation of such processes may be used to drive real-world interactions using Haskell **Monads**. I will then extend this implementation to CaSE$^{ip}$, and show how abstract time fits in with the monadic model. I will also describe my verification framework for CaSE$^{ip}$ processes, and demonstrate my command-line tool **ConCalc**. Finally, I will detail my efforts at implementing Cashew-A itself, using the CaSE$^{ip}$ implementation.*

## 8.1 Introduction

A MAJOR ADVANTAGE OF THE CCS PROCESS ALGEBRA, along with its various timed derivatives, is that it can be faithfully implemented, primarily due the simplicity of its operators. By "faithful" I refer to the implementation providing a true representation of the formal model described in the calculus. Implementation provides at least two benefits:

- Verification of abstract processes, such as bisimulation and refinement checking;
- Execution, for the purposes of testing.

In order to simplify the former I have chosen the purely functional programming language *Haskell* (Peyton-Jones, 2003). Haskell allows a very concise implementation which is also very close to the operational semantics, and thus reduces the possibility of error. Furthermore, recent extensions to Haskell's type-system allow us to make certain guarantees about processes, for instance restricting process to those which are finite state.

Thee contributions in this Chapter come under three main sections:

- **An implementation of CaSE$^{ip}$** in Section 8.2. I will begin building an incremental implementation of a timed process calculus, starting with a basic CCS process algebra and setting some principles for how computation is abstracted in CCS. I will seek to justify my use of Haskell by showing that *Monads* provide a clean way of binding computations to the process syntax. I will then move on to explore how Haskell's type system can be used to make static guarantees about process syntax. Finally, I will add abstract time to the model, resulting in an implementation of CaSE$^{ip}$.

- **An LTS verification framework** in Section 8.3. This general framework will provide a library of tools for studying Labelled Transition Systems. I will describe a number of Haskell classes which provide and LTS interface onto processes and then use these to build verification functions. The main contribution of this Section is a *partition refinement* algorithm, which allows both *bisimulation checking* and LTS *minimisation* , both of which are directly applicable to Cashew-A . I also introduce my verification tool ConCalc, which has been used for extensive testing of the CaSE$^{ip}$ implementation.

- **A partial reference implementation of Cashew-A** in Section 8.4. This draws together all the other implementation work and show the viability of the Cashew-A operational semantics presented in Chapter 7. I will present how I implement the semantic mapping, and show some sample transition graphs. In particular I will show the graph for the calculator example introduced in Chapter 4.

## 8.2 Process Calculus Implementation

### 8.2.1 Background: Computation in CCS

CCS , at its core, is not specifically a model of computation. Unlike CSP it provides no inherent notion of process state beyond the communications a process is willing to make at a particular time. CCS is rather an abstract model of communication which is used to explore different types of models which can be represented purely in terms of abstract synchronous processes. Therefore, if we are to make CCS processes executable it is first necessary to consider exactly where computations occur. Physical computations can then be bound to the process syntax which will allow a CCS process to drive an execution model.

We must consider, therefore, exactly what a CCS process means so that we can understand where computations should occur. On the one hand we must remain faithful to CCS, ensuring that the model cannot do anything that CCS does not permit. On the other hand we must also allow the implementation to "fill in the gaps" which CCS leaves undefined.

An important point to remember about CCS is its use of *interleaving semantics*, that is processes in the calculus are resolved into several sequences of atomic steps. Parallel behaviour is achieved by interleaving the steps of two such processes. An atomic step represents an instantaneous event which occurs at some specific point in the process's time-line. They are never composite tasks which take time to resolve, otherwise a process being executed would have to constantly wait for them to complete. This assumption is vital for concurrency in an interleaving setting, since in a process like $a.\mathbf{0} \mid b.\mathbf{0}$ it then does not matter whether $a$ or $b$ occurs first in the transition system – they both occur at effectively the same instant.

Therefore, a visible action is not a representation of computation, rather it is simply a signal. Any time-consuming computation takes place in the *space* between two observations. For instance, the CCS process $a.\overline{b}.\mathbf{0}$ may, at first sight, seem to refer to a process which immediately performs a $\overline{b}$ after an $a$, but this is naïve. Such a CCS process may be represented, to borrow the illustration from van Glabbeek (2001), by a machine with a screen on it which displays the action which the process is performing[1]. When observing this process we would first see an $a$ on the screen, but straight afterwards the screen goes blank. We don't know what the machine is doing at this point, but we do know at some point in the future it will print out a $\overline{b}$ and then stop. Thus, the . operator in CCS actually refers to an unspecified time period between two observations.

CCS's observation equivalence semantics further solidifies this idea by allowing any two observations to be separated by an unbounded number of silent steps, represented by $\tau$s. So the process $a.\overline{b}.\mathbf{0}$ is equivalent to $a.\tau.\overline{b}.\mathbf{0}$ or $a.\tau.\tau.\overline{b}.\mathbf{0}$, and so on. Thus silent steps represent the presence of computation – some unobservable action which takes an unknown length of time to complete and somehow affects the state of the world. Silent actions are not themselves the computations, as like all steps in an interleaving semantics they must be instantaneous. Rather they represent the explicit presence of internal activity which must be completed before the process can advance down that path. Thus, an unguarded $\tau$ is a sign that invisible work is in progress. When the $\tau$ transition is resolved it means that this activity has completed, and thus the process can continue. It is therefore fair to say that in CCS concurrency is not present in interleaving, but in the silence between observations. It is not the *tasks* themselves which are interleaved, but rather the actions scheduled or used to observe those tasks. These ideas are made even more explicit with the addition of *abstract time*, but for now we stick with basic CCS as a model.

When we come to implement this idea in Haskell, we find an analogous notion of hidden, internal and unobservable activity in the *Monad* . A monad is principally an abstract mathematical object, but mostly they are used to represent *computations* and provide primitives for connecting these computations in different ways. A particularly interesting class of monads are the IO monads, which allow impure computations to be

[1]Ignoring for the moment that these actions are instantaneous and would therefore be invisible to the naked eye.

represented in an otherwise purely functional programming language. The IO monad allows file access, socket communication and any sort of activity for which the result cannot be directly observed or influenced by the program, and is thus *unsafe*. More importantly, when we add in the features of *Concurrent Haskell* (Peyton-Jones et al., 1996), a monadic action can schedule a thread, or see if one has completed.

In this implementation of CCS I will use monads to represent the silent computations taking place in CCS processes. Every operator which can emit a silent action must have some form of monadic computation associated with it, which must be successfully executed *before* the associated $\tau$ step can be taken. We could have it executing after, but this would violate the CCS semantics, since there would still be internal activity occurring. This will become particularly important when we introduce abstract time, as maximal progress requires that all internal computations are complete for the clock to tick. This approach has the advantage that there is a definite separation between the abstract and concrete behaviour of a process, rather than combining the two. Hence, the abstract theory developed thus far remains sounds.

However, the concept of choice in CCS raises another interesting problem. When we consider choices in CCS, the silent steps allow us to fundamentally alter the semantics of a process from one where the choice is resolved by an observer, $a.P + b.Q$, to one which is resolved purely internally, $\tau.P + \tau.Q$. As a result an unguarded silent action is very different to a guarded one, and therefore choices require special handling in any implementation. If we were to stick with the above definition of computations, executing before a $\tau$ path could be taken, choices would involve each branch having its own internal guard. These guards would have to be executed in parallel and the first one to complete would have its branch chosen. This raises a number of problems, not least the fact that the guards would have to be completely independent, when in reality they are rarely so.

As a result we adopt a slightly more conservative notion of choice. Each internal choice is decided by a *single* computation, whose result causes one of the $\tau$ paths to be chosen. This is consistent with our definition of a $\tau$ representing the completion of a computation, but with the addition of some guard-data being passed internally.

The next question to be answered is how parallel composition should be dealt with. Since our processes have underlying abstract computations, how should two processes exchange data at synchronisation? First we need to consider exactly what the CCS operators mean in terms of data transfer. CCS at its core is completely abstract – the terms $a$ and $\overline{a}$ mean nothing more than complementary actions, they are not concretely input and output. However in extensions of CCS such as the *value passing* variant and $\pi$-calculus they are, nevertheless, interpreted concretely as inputs and outputs with associated data. The value passing calculus works purely by substitution, with parallel composition working like this:

$$\frac{E \xrightarrow{\overline{a}v} E', F \xrightarrow{a(x)} F'}{E \mid F \xrightarrow{\tau} E' \mid F'\{v/x\}}$$

The process on the left outputs a value, and the process on the right inputs the value and substitutes it for every instance of $x$. Of course, this is essentially $\pi$-calculus (Milner, 1999), the difference being that the data sort is disjoint with the channel sort. Such an approach provides a useful inspiration for data transfer. Nevertheless, this approach inherits many of the theoretical problems of $\pi$, related to of binding channel names. It also restricts the possibilities for abstraction, since synchronisation cannot be reduced to a $\tau$, as this will remove the flow of data. For instance $(a(x).\overline{b}x.0 \mid b(y).\overline{c}y.0) \setminus b$ is not simply equal to $a(x).\tau.\overline{c}y.0$, where basic CCS would identify the equivalent value-less processes. It also makes separating the LTS from the execution model difficult, since the actual process syntax can be influenced by the impure real-world, which makes finite verification impossible.

Therefore rather than embedding value passing directly into the calculus, CCS retains the standard syntax, but every synchronisation has an *implicit* transfer of data associated with it. Rather than treating these simply as a pure substitution, we treat them as monadic actions to be composed. The reason for this is that data may originate from an impure source, and thus cannot be substituted into the process syntax itself. Instead the computation is kept within the monad, so that the LTS can be built without impure computations. As in CSP, we will make the simplification that each process simplifies to a hierarchy of sequential agents. Each sequential agent will have its own distinct state-space into which data is written, and from which it is read. As a result, synchronisations will only be able to move data from one agent to another and not perform any complex IO operation. This means we don't have to deal with the complexity of sharing two state spaces. All "external" data acquired via interaction with the real world will originate from explicit $\tau$ prefixes.

In the next section I will begin to expand on some of these details.

### 8.2.2 Basic CCS

I begin by looking at standard CCS processes with choices, parallel composition and restriction as defined by Milner (1989*a*). For the time being all computations will be represented using abstract monads. Throughout this chapter I will be using the *Generalised Algebraic Datatype* (GADT) notation explained in Chapter 3 Section 3.5 in order to explicitly define the constructors, which will become convenient later. I adapt the basic syntax found in Chapter 2 Section 2.3.3 to the following algebraic data types.

```
data Process a v m where
    -- The stalled process; 0
    NIL :: Process a v m
    -- Observable action prefix; a.P
    OBS :: Action a v m → Process a v m → Process a v m
    -- (External) Choice / Summation; P + Q
    SUM :: Process a v m → Process a v m → Process a v m
    -- Silent action prefix; τ.P
    TAU :: m Bool → m () → m Bool → Process a v m → Process a v m
    -- Parallel Composition; P | Q
    PAR :: Process a v m → Process a v m → Process a v m
    -- Restriction; P \ a
    RES :: a → Process a v m → Process a v m
    -- Renaming; P {[a ↦ b]}
    REN :: Process a v m → (a, a) → Process a v m

data Action a v m = Input a (v → m ()) | Output a (m v)
```

This initial data type has three type parameters:

- a, representing the action name sort $\Lambda$;
- v, representing the type of data to be passed between parallel processes by synchronisations;
- m, representing the underlying Monad used for the execution model.

At this stage I assume that all processes pass the same type of data for the sake of simplicity. The Action type represents observable actions, i.e. $a$ and $\overline{a}$, specifically the union of names and co-names $\Lambda \cup \overline{\Lambda}$. An output action is associated with a computation in Monad m producing a value of type v. An input action is associated with a function from a value of type v to a computation with an empty output. These two components are composed together with monadic bind $\gg=$ upon synchronisation. The output component produces a value which is then passed to the input component, the result being an internal computation of type m(), which can directly be executed.

An observable action with constructor OBS takes an observable action and a process to evolve into, after performing the action. Silent action prefix, TAU has three monadic computations. The first is the pre-guard, which decides whether the computation has started executing or not, the second is the actual computation and the third is the post-guard, which decides whether the computation has finished or not. If the pre-guard evaluates to False, it means that the computation does not yet have requisite preconditions to execute, and thus cannot be started. If the post-guard evaluates to False, this means the

$\tau$ prefix cannot yet reduce, and the execution semantics will pass over this transition at that point. The pre- and post-guards should be instantaneous actions, in that they shouldn't do any actual work, simply perform a check. The computation should likewise be a non-blocking operation (e.g. spawning a thread), which when finished will set the post-guard to true. When putting together an internal choice, the programmer should use a process like $\tau.(\tau.P + \tau.Q)$, in which the leading $\tau$ holds the actual computation which makes the choice, whilst the others are mutually exclusively guarded and contain a null computation.

Equipped with a data-type to represent the calculus, I now proceed to give it a semantics. The semantics is defined using a function `step :: Process a v m → [Transition a v m]`, which gives a list of transitions which the given process can immediately enact.

```
data Transition a v m      = Tran (TransitionLabel a v m) (Process a v m)
data SilentType a          = External | Synchronize a
data TransitionLabel a v m = Observable (Action a v m)
                           | Silent (SilentType a) (m Bool) (m ()) (m Bool)
```

A Transition is a label with a process, with the label representing either an observable action or a silent action with associated computation. TransitionLabel is equivalent to the CCS action sort $\mathscr{A}$, i.e. $\Lambda \cup \overline{\Lambda} \cup \{\tau\}$. The `Observable` constructor simply links to observable actions, whilst `Silent` consists of the silent action triple of pre-guard, computation, post-guard, together with a SilentType. The SilentType data-type contains information on the origins of the silent action, where in CCS this can either be from a $\tau$ prefix (`External`) or a synchronisation (`Synchronize`).

I now define `step`, which is derived from the CCS transition relation $\rightarrow$, specifically $p \xrightarrow{\alpha} p' \implies \texttt{Tran}\ \alpha\ \texttt{p}' \in \texttt{step p}$. I define `step` recursively as shown below:

```
step NIL              = []
step (OBS l p')       = [Tran (Observable l) p']
step (TAU pr m po p') = [Tran (Silent External pr m po) p']
step (SUM p q)        = step p ++ step q
```

`NIL` has no transitions, so simply returns an empty list. `OBS` produces a single observable action, whilst `TAU` produces a silent action with type `External`. `SUM` simply concatenates the transition lists produced by either side. Notice that each of these definitions (except for the `NIL` case) corresponds to one or more rules in CCS (See Chapter 2, Section 2.3.3). The `OBS` and `TAU` cases together make up the Act rule. The `SUM` case combines the rules Sum1 and Sum2 – it makes two recursive calls to `step`, each corresponding to the respective antecedents. Indeed, a recursive call to `step` is equivalent to a rule antecedent being checked.

The definition for `PAR` combines the two computations, as shown below:

```
step (PAR p q) = let tp = step p; tq = step q in
                 [ Tran a (PAR p' q) | Tran a p' ← tp ] ++
                 [ Tran a (PAR p q') | Tran a q' ← tq ] ++

                 [Tran (Silent (Synchronize a) (return True) (o >>= i)
                               (return True)) (PAR p' q')
                 |(Tran (Observable (Input a i)) p') ← tp
                 , (Tran (Observable (Output b o)) q') ← tq, a == b] ++

                 [Tran (Silent (Synchronize a) (return True) (o >>= i)
                               (return True)) (PAR p' q')
                 |(Tran (Observable (Output b o)) p') ← tp
                 , (Tran (Observable (Input a i)) q') ← tq, a == b]
```

The first line generates the transitions for each process and places them into two variables `tp` and `tq` for convenience. The independent actions from either side of the operator are then calculated by building a transition where one side is changed but not the other. It does this using *list comprehensions*, which mirror a set comprehension by performing an operation on every element of a list. The first two lists correspond to CCS rules Com1 and Com2 respectively. These two list comprehensions simply deconstruct the transition to pull out the label and resultant process. A new transition is then built from the label with the resultant process being a parallel process with one side unchanged.

The remaining two list comprehensions collect the synchronisations by matching an input with an output from either side, corresponding to CCS rule Com3. Both comprehensions are essentially the same, but for Input/Output and Output/Input on the left and right side respectively – this is necessary because it isn't possible to encode the fact that $\overline{\overline{a}} = a$. The transition formed is a silent synchronisation transition containing a computation made by composing the output and input computations. Notice that the pre- and post-conditions are both set to `True` – this is because synchronisations are deemed instantaneous, being simple data-transfer, and thus do not have pre- or post-conditions.

Notice that all the semantics are defined by creating one or more lists for each semantic rule and then concatenating them. This is the way that all the semantics in this Chapter will be defined, with an increasing emphasis on list comprehensions since this mirrors the logical definition in the Structural Operations Semantics (SOS). Next I define the semantics for the restriction operator:

```
step (RES p h) = [Tran (Observable (Input a m)) (RES p' h)
                 |Tran (Observable (Input a m)) p' ← tp, not (a == h)] ++

                 [Tran (Observable (Output a m)) (RES p' h)
                 |Tran (Observable (Output a m)) p' ← tp, not (a == h)] ++

                 [Tran (Silent t pr m po) (RES p' h)
                 |Tran (Silent t pr m po) p' ← tp]

  where
    tp = step p
```

The restriction operator's semantics filters out input and output transitions from the enclosed process possessing the restricted label. This is again done using a simple list comprehension, which checks if a transition's label `a` is the restricted label `h`. All silent transitions are simply passed through.

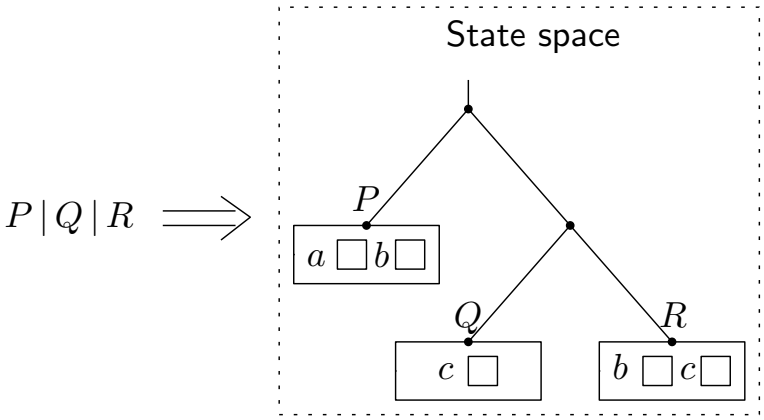
Once `step` has been defined for all operators of the language the process can be executed. By "executed" I mean that all the monadic computations emitted by the process can be sequentially composed and invoked by whatever method is specific to the given monad. The idea is that the programmer will choose a suitable monad for their specific task. I assume that a process to be executed will be fully restricted, only emitting silent transitions and thus will be executable, as observable actions are partial and cannot therefore be executed. Before this can happen though some sort of transition picker function is needed. This is needed because `step` returns a list of possible transitions, and it is necessary therefore to pick one as the next to execute. Thus the `runProcess` function, which executes a process, has three arguments:

- The maximum number of simulation steps (or `Nothing` for unbounded);
- A monadic picker function, returning the chosen transition and the others;
- The actual process to be executed.

```
runProcess :: Maybe Int →
              ([Transition a v m] → m (Transition a v m, [Transition a v m])) →
              Process a v m → m ()
```

In each iteration the algorithm for `runProcess` is as follows:

1. Obtain a list of transitions for the process using `step`;




$$
\begin{aligned}
P &= a.\overline{b}.\mathbf{0} \\
Q &= c.\mathbf{0} \\
R &= b.\overline{c}.\mathbf{0}
\end{aligned}
$$

Figure 8.1: A basic state space BTree

2. If the list is empty or the maximum number of cycles is passed, terminate. Otherwise continue;
3. Use the picker function to ascertain which transition will be executed next;
   - The picker returns the first observable action it encounters, or the first ready silent action. Readiness is based on the silent action's precondition.
4. Based on the next transition:
   - If the transition is an observable action then do nothing;
   - Otherwise execute the associated body of the silent action.
5. Iterate with the new process.

Thus we have accomplished building a framework which can produce code from a basic CCS process annotated with monadic actions. This framework is fairly primitive thus far, and therefore I will now go about enhancing it in the following Sections.

### 8.2.3 Hierarchical State Space

Moving on from the basic CCS model, we now turn our attention to process state. To reiterate, we are modelling processes as a hierarchy of sequential processes, each having its own internal state. Behind this exists the world-state, which is distinct and can obviously be manipulated to a lesser degree. So far all interactions have simply been within an abstract monad which tells us very little about state. This monad, for instance, could simply be IO which would give us access to the real world. However, the IO monad does not provide an explicit state space for each distinct agent. Thus, in this section I

will further expand my CCS implementation to provide a realistic state-space which can be used to write $\tau$ bindings.

An important aspect to consider here is compositionality – even at the level of implementation it should still be the case that the semantics of the whole is a composition of the semantics of the parts. The question is how do we allow each agent to have a distinct state space and still maintain this, all within the context of a monad? The usual way of maintaining internal state is via a variant of the State monad (see Chapter 3, Section 3.3), which is parametrised over the type of state. The more useful relative, StateT, the state transformer monad, allows another monad to be enclosed inside. For example, we might type our computations as StateT MyState IO which would allow us to manipulate MyState via `get` and `put`, but also perform IO operations.

But how should MyState represent a state-space for each agent? The naive way would simply be a list, each element of which is a process state and each agent would be associated with an index of this list. But aside from the obvious problem of allocating numeric indices, this does not allow composition of computations in the hierarchy. Consider composition of two processes, $P$ and $Q$, assuming they are laden with computations working on different parts of a state. How should these computations be composed in such a way that there is no state overlap? If they both work on a flat state space, and indices of the respective computations clash, there will be problems. Clearly then it's not easy, the main problem being that a flat state space is incongruous with the process hierarchy. Therefore, what is needed is a state space which is also hierarchical, therefore mirroring the process hierarchy. We start from a simple binary tree:

```
data BTree a = BBranch (BTree a) (BTree a) | BLeaf a | BStub
```

We view a process as simply a binary composition of processes, ignoring restriction, and use this data-type to associate a state variable with each agent (a leaf). This is exemplified in Figure 8.1 where we have three parallel processes $P, Q$ and $R$, each of which having a state corresponding to its channels, the idea being that data acquired or sent is stored in these variables. Clearly this tree can easily be expanded upwards as other processes are composed in parallel.

The tree grows at run-time, initially starting with a `BStub` and then adding data as computations are executed from different parts of the process. This is done via two functions, `stateLeft` and `stateRight` which are used in conjunction with parallel composition to build a path for the given composition's state-space. For instance, if a $\tau$ originates from the left-hand side of a parallel composition, the `step` function will augment the associated computations with `stateLeft` which will enable the computation to find the correct position for its state variable in the BTree. In the example in Figure 8.1 the path to $P$ is $l$, whilst the path to $R$ if $rr$. Hence, when the synchronisation on $b$ occurs, the computation associated with $\bar{b}$ will be augmented with `stateLeft` once, whilst the com-

putation associated with $b$ will be augmented with `stateRight` twice. The two resulting computations can then be composed and applied to the state tree.

The code for `stateLeft` is shown below; its partner is almost identical, the only difference being it transforms the right rather than left branch.

```
stateLeft :: Monad m ⟹ StateT (BTree s) m a → StateT (BTree s) m a
stateLeft m = do s ← get
                 (l, r) ← case s of
                            BBranch l r → return (l, r)
                            BStub → return (BStub, BStub)
                 (a, l') ← lift $ runStateT m l
                 put (BBranch l' r)
                 return a
```

The function assumes a StateT Monad with a polymorphic BTree as the state parameterised over `s`, the type of the agent variables. It transforms a computation in this monad into another of the same type, shifting computations to use the left subtree. The function deconstructs the tree in the state, pulls out the left branch, runs the computation using that as the state, places the resulting state back into the left branch and finally returns the value produced. If the given tree is empty (i.e. it is a BStub) then the tree is first grown by one-level, with a stub at each branch. If a BLeaf is present then the function is undefined – this should never happen in practice as the tree should only ever mimic the process topology. Using this function I build another pair of functions `actLeft` and `actRight`, which transform an Action by transforming its computation to the left or right subtree:

```
actLeft :: Monad m ⟹ Action a (BTree s) m → Action a (BTree s) m
actLeft a = case a of
              Input a f → Input a (stateLeft . f)
              Output a m → Output a (stateLeft m)

data Action a s m = Input a (Dynamic → StateT s m ())
                  | Output a (StateT s m Dynamic)
```

In addition I have changed the syntax of Action so that all computations are wrapped in the StateT transformer (cf. Chapter 3 for details of transformers). Furthermore I have removed the `v` parameter and replaced it with an `s` to represent the type of state for each process. Each value being exchanged is now a Dynamic rather than an arbitrary type. A Dynamic may contain any value, so long as its type can be reflected at runtime. This allows differently typed values to be exchanged, which is naturally vital to writing realistic $\tau$ bindings. This change is also reflected in the Process type, which has an extra parameter:

```
data Process a s m where
    ...
    -- Observable action prefix; a.P
    OBS :: Action a s m → Process a s m → Process a s m
    -- Silent action prefix; τ.P
    TAU :: StateT s m Bool → StateT s m () → StateT s m Bool → Process a s m
                                                              → Process a s m
    ...
```

Additionally, the TAU constructor is altered to have its associated pre-guard, computation and post-guard running in the StateT monad, but using the s state instead of a BTree. The BTree will be merged in via the semantics. Therefore, I also define a function liftStateM which lifts a computation using the StateT monad with state s (i.e. the state of an individual agent) into one using a BTree state:

```
liftStateM :: (Monoid s, Monad m) ⟹ StateT s m a → StateT (BTree s) m a
liftStateM m = do s ← get
                  sl ← case s of
                           BLeaf l → return l
                           BStub → return mempty
                  (a, s') ← lift $ runStateT m sl
                  put (BLeaf s')
                  return a

actLeaf :: (Monoid s, Monad m) ⟹ Action a s m → Action a (BTree s) m
actLeaf a = case a of
              Input a m → Input a (liftState . m)
              Output a m → Output a (liftState m)
```

This function, liftStateM, takes the current state-space tree, assuming it is either empty or a leaf, and uses the state to run the given computation. It gets the current state, pulls a state value out and uses this with the monad execution function runStateT to run the computation with the given state. The resulting state value is then lifted back into the monad and the output value returned. If the initial state-space is empty (i.e. the tree is BStub) then an empty state is used, the assumption being that the agent will write to the state. Therefore, to ensure an empty state value exists, the state type s must be have an instance of the Monoid class, which provides a zero value (mempty) and a binary composition function (mappend). The second function, actLeaf, converts an Action computation into one running in a leaf-node state-space. Finally, I create an expanded version of step for OBS and PAR (as only these cases change), along with a redefined TransitionLabel which takes account of the hierarchy:

```
data TransitionLabel a s m = Observable (Action a (BTree s) m)
                           | Silent (SilentType a c)(StateT (BTree s) m Bool)
                                                   (StateT (BTree s) m ())
                                                   (StateT (BTree s) m Bool)
-- Transform the computation in l to one working on a leaf node state
step (OBS l p') = [Tran (Observable (actLeaf l)) p']
```

```
-- Transform the computations from p and q to the left and right state tree branches
-- (respectively)
step (PAR p q) =
  let tp = step p; tq = step q in

  [Tran (Observable $ actRight o) (PAR p qq)
  |Tran (Observable o) qq <- tq] ++

  [Tran (Observable $ actLeft o) (PAR pp q)
  | Tran (Observable o) pp <- tp] ++

  [Tran (Silent t (stateLeft pr) (stateLeft m) (stateLeft po)) (PAR p' q)
  | Tran (Silent t pr m po) p' <- r] ++

  [Tran (Silent t (stateRight pr) (stateRight m) (stateRight po)) (PAR p q')
  | Tran (Silent t pr m po) q' <- s] ++

  [Tran (Silent (Synchronize a) (return True) (stateRight o >>= stateLeft.i)
                (return True)) (PAR p' q')
  |(Tran (Observable (Input a i)) p') <- tp
  ,(Tran (Observable (Output b o)) q') <- tq, a == b] ++

  [Tran (Silent (Synchronize a) (return True) (stateLeft o >>= stateRight.i)
                (return True)) (PAR p' q')
  |(Tran (Observable (Output b o)) p') <- tp
  ,(Tran (Observable (Input a i)) q') <- tq, a == b]
```

The new definitions add state-space transformers for all the cases, making sure the state is picked from the correct branch of the tree. The case for OBS transforms the computation into one running on a leaf node, and the case for PAR into one on a branch. In particular notice that when binding the computations from each side of a synchronisation in PAR, the opposite branches of the state tree are used for each binding using

`stateRight` and `stateLeft`, respectively. The only question that remains to be answered is how should the state `s` be represented? For the time being I do not set any specific state, but it seems that the most sensible state is a Map from action labels to Dynamic values (Map a Dynamic). Then when a value is input on a channel it can be stored at the respective index, or when sent it can be taken from the respective index. Using this kind of state the following process can be built:

```
p :: a → a → Process a (Map a Dynamic) IO
p a b = OBS (Output a aB)$ TAU (return True) tB (return True)
                         $ OBS (Output b bB NIL)
  where
    aB x = do s ← get
              put (insert a x s)
    tB = do s ← get
            x ← lookup a s
            y ← getLine
            let z = ((fromDyn x (0 :: Int)) + (read y))
            put (insert b (toDyn z) s)
    bB = do s ← get
            x ← lookup b s
            return x
```

This process demonstrates both the internal state and a simple IO interaction. Function `p` represents the CCS process $a.\tau.\bar{b}.\mathbf{0}$, where the two parameters are the input and output channels. It inputs an integer value (wrapped up in a Dynamic), internally requests another integer value to be input from the keyboard (within the $\tau$ binding) and outputs the sum. This is done in terms of three bindings, `aB`, `tB` and `bB`, the bindings for the input, $\tau$ and output, respectively. The first takes the value passed to it by the environment and places it in the state map under the key named by the input channel. The second does the actual computation: it pulls the input value out of the state map, requests a value from the keyboard, adds these two values together and places them in the state map under the output channel. The third simply pulls the output value out of the map and returns this to be output to the environment.

One thing to note: for the sake of brevity, the middle `TAU` action is implemented slightly incorrectly. The associated binding `tB` waits on the keyboard input via `getLine` before continuing, to input a number. Action bindings should take negligible time to execute, and therefore `tB` should spawn a thread to get the keyboard input and immediately return. The associated postcondition, rather than returning `True` as it does in the example, should return `False` until this keyboard input is supplied. This will prevent the guarded process (which outputs $b$) from being activated until the input is available,

but allow other parallel processes to schedule their bindings, thus leading to proper concurrency. Incidentally, the precondition (**return** `True`) is correct, since the only condition of execution is that the input $a$ has been received, which is enforced by the process semantics.

A slightly more difficult problem exists though. The state space is not directly type-safe, as two parallel processes exchanging data may disagree on a value's type. This is very obvious from this example as it assumes the input Dynamic contains an integer value which it may not (however, in this instance if it doesn't it substitutes a $0$ as default). Thus, in the next section we explore how the problem of type safety could potentially be overcome.

### 8.2.4 Towards Strong Typing

Up until now, all the data being exchanged between processes has been effectively untyped. Although the parameter $\mathrm{v}$ can be set to any single type, this will be retained through the whole process structure and for every channel. This is inconvenient, and channels may obviously require to exchange different types of data. As I explained in the previous Section, using Dynamic values for this purpose is not directly safe. Therefore, in this section we will exploit some of the newest features of Haskell, in particular *GADTs* and *Type Families* (see Chapter 3), to enrich CCS processes with type information for the data being exchanged. This work is currently very experimental, but I think is very interesting for showing how Haskell can be potentially used for compile-time static verification.

Each CCS process will be associated with a typing context $\Gamma$. This typing context will associate a type, ranged over by $\theta$, with each channel name, representing the type of data it carries. To aid in describing this type-system, we will imagine a calculus syntax with embedded type annotations (although I will not give this an operational semantics, as the annotations are purely for pre-execution static analysis):

$$\mathcal{E}_t ::= \mathbf{0} \mid a:\theta.\mathcal{E}_t \mid \tau:\widetilde{a\times\theta}.\mathcal{E}_t \mid \mathcal{E}_t + \mathcal{E}_t \mid \mathcal{E}_t \mid \mathcal{E}_t \setminus a:\theta$$

Notice that $\tau$ witnesses a set of type associations. This is because I assume that every silent action reads from and writes to any of the values in scope. The expressions of the language are then typed as follows:

$$\frac{}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma \vdash E}{(a,\theta),\Gamma \vdash a:\theta.E} \qquad \frac{\Gamma \vdash E}{\widetilde{a\times\theta},\Gamma \vdash \tau:\widetilde{a\times\theta}.E}$$

$$\frac{\Gamma \vdash E, \Gamma \vdash F}{\Gamma \vdash E + F} \qquad \frac{\Gamma \vdash E, \Gamma \vdash F}{\Gamma \vdash E \mid F} \qquad \frac{(a,\theta),\Gamma \vdash E}{\Gamma \vdash E \setminus a:\theta}$$

Effectively, restriction behaves like an existential quantifier which witnesses the type

of the given channel. This type system can be embedded directly into Haskell's type system through GADTs, which allow each expression of the Process type to be given a different type (in terms of type parameter qualification). However due to limitations in Haskell's current implementation of type-families and other issues, we need to make some simplifications. Instead of having a type-context which associates channel names with types, we will have a type stack and use de-Bruijn indices (de Bruijn, 1972) to attach type data to the channels. The reason for this is two-fold:

- We cannot directly implement an array of types in Haskell due to type functions being severely limited in terms of recursion. This may be lifted at some point by so-called *closed type families* (Schrijvers et al., 2008) which promise to allow more liberal definition, but at the moment this problem is insurmountable;
- Even if we could have an array of types, we cannot represent a channel name/type pair because one is value-level whilst the other is type level. Thus we need to use de-Bruijn indices as an intermediary and then use the *Template Haskell* (Sheard and Peyton-Jones, 2002) meta-language to create macros to generate them from action names (Template Haskell, being at meta-level, can happily work with both types and values as they're both just syntax).

So, first of all we need to define these type-level indices. They will be represented as type-level natural numbers, with a Zero type and a parametric Successor type:

```
data Z   = Z
data S n = S n
```

Using these we have access to a complete set of natural numbers at type level, for instance 3 is S S S Z. Along with natural numbers I also need type lists and union types. The former will be used to represent type contexts, and the latter, elements of the context.

```
data Nil
data Cons x xs
```

The type-list is based on normal value level lists, though instead of two data constructors it has two type constructors, one for the empty list, and one for a type followed by a list. In addition a *type family* (see Chapter 3) is needed which will act as a type-level indexed lookup function for lists:

```
type family ts :!! : i
type instance (Cons x xs) :!! : Z = x
type instance (Cons x xs) :!! : (S n) = xs :!! : n
```

This defines an infix type-function :!!: which is parameterised over a type-list `ts` and a natural number `i`, the lookup index. The lookup implementation is a straightforward recursive function. If the index is Z then the type is the head of the type list, and thus `x` is returned. Otherwise the index is decremented and used in a recursive call using the tail of the list. For instance, the type Cons Int (Cons Float Nil) :!! : S Z collapses to Float.

```
data UnionType ts where
     It :: t → UnionType (Cons t ts)
     Alt :: UnionType ts → UnionType (Cons t ts)
```

A type union defines values whose type is one of a number of given types. In this instance a type union is parametrised over a list of types a value could be `ts`. A value in a type union is built using two constructors. The first, It, refers to a value which is typed by the type at the head of the union's type-list. The second, Alt, is a value whose type is in the tail of the list. For instance, we could define Alt (Alt (It 'a')) which has type UnionType (Cons a (Cons b (Cons Char ts))), where a and b may be any type, and ts is the remainder of the type-list. A satisfactory monomorphic type for the union might be UnionType (Cons Int (Cons Float (Cons Char Nil))). I now also define a type-class for manipulating this union-type:

```
class Union ts i where
     mkUnion :: i → ts :!! : i → UnionType ts
     appUnion :: Monad m ⟹ i → (ts :!! : i → a) → UnionType ts → m a
```

Union types are not easy to manipulate by default. I therefore provide the above class which enables the two most important manipulations, namely creating a union and applying a function to one. The reason for using a class rather than a function is because the input is obviously polymorphic, and therefore not definable in a single function. The two instances follow:

```
instance Union (Cons t ts) Z where
         mkUnion Z x = It x
         appUnion Z f (It x) = return (f x)

instance Union ts n ⟹ Union (Cons t ts) (S n) where
         mkUnion (S n) x = Alt (mkUnion n x)
         appUnion (S n) f (Alt x) = appUnion n f x
```

The first function, `mkUnion`, takes a list index and a value of the type at that index in the list and returns a union type containing the given value. For example the expression

`mkUnion (S Z) "hello"` creates a union where String is at index 1. The second function, `appUnion`, takes an index, a function from the type at that index to another polymorphic type, and a union type with the same type-list and attempts to apply the union type to the function. For instance `appUnion (S S Z) f u` would apply the function `f` to the value stored in union `u`, whose value is at the 3rd index. In the latter function the index is not strictly needed since it could be derived from the type, but Haskell requires both parameters to be explicitly quantified.

With all the underlying types defined I now proceed to define the new typed CCS process type:

```
data Process a ts m where
    -- The stalled process, 0
    NIL :: Process a ts m
    -- A data action, the data is scoped over the guarded process via a function
    DOBS :: DAction a ts m → Process a ts m → Process a ts m
    -- Data restriction, existential quantification of the encapsulated type t
    DRES :: Process a (Cons t ts) m → Process a ts m
    -- Choice / Summation, P + Q
    SUM :: Process a ts m → Process a ts m → Process a ts m
    -- Parallel Composition, P | Q
    PAR :: Process a ts m → Process a ts m → Process a ts m
    -- Silent action prefix
    TAU :: m Bool → m () → Process a ts m → Process a ts m

data DAction a ts m = Union ts i ⟹ DOutput i a (m (ts :!! i))
                    | UnionF ts i ⟹ DInput i a (m Bool) ((ts :!! i → m ()))
```

I have not included typed silent actions (i.e. `DTAU`), as for now I don't know the best way to represent them. The first thing to note is that Process now has parameter `ts` instead of `v`. This is the list of types which the given process exposes. The most interesting constructors in our new data-type are `DOBS` and `DRES`, the new version of prefix and restriction – the other constructors simply propagate the type-list through the process topology. The new typed action prefix uses the DAction type, which is the same as Action except the monadic operations are strongly typed using the type-list.

Both of the constructors *existentially quantify* the type `i` (see Chapter 3, Section 3.2.2). The parameter represents the index used to decide which type is being used from the type-list in the computation. The first, `DOutput` has a computation of type `(m(ts :!! i))`, a value of the `i`th type in the type-list, returned in monad `m`. The Union constraint is there to make sure that a union type can be built using the index and type list. In theory, if `i` were closed, for example if it had a natural number kind and could thus only be a natural number this would not be required. However, since by default `i` can be any type

and not just a natural number this constraint is needed to effectively make sure it is a natural number type.

The second constructor, `DInput`, is much the same as the former. The difference being the type represents a monad function rather than a value. Also instead of using the Union class constraint, another class UnionF is used. UnionF is similar, but it works on unions of functions instead of unions of scalar types. For reasons of implementation these need to be handled differently, and thus a different class (and type) is used.

Moving on from `DOBS`, `DRES` the restriction constructor is an existential quantification of the type of the restricted channel. Thus it takes a process with a given type-list and produces a process with the head type removed from the type-list. If we have `p :: Process a (Cons Int Nil) m`, for instance, and `p = a :: Int.0` then `DRES p :: Process a Nil m`. It is unfortunate that `DRES` can only restrict the channel numbered as $0$, but the reason is that removing an arbitrary element from the type-list would require a transformation more complex than type families allow. Thus processes must be crafted carefully to ensure that they are restricted correctly.

Because of these restrictions, although certainly Haskell is showing promise in allowing this kind of compile-time verification, it is not currently practical for my needs. As a result, whilst I include this section for interest, ultimately I will not be using it further.

### 8.2.5 Recursion

Returning to the untyped model, I now add the final operator of CCS: recursion. Recursion in CCS can be described in several ways, but in my model I will use the fix-point operator $\mu X.E$. The question of recursion is therefore really a question of process variable substitution, and the most efficient way of doing this. There are two possible ways of writing the recursion rule for CCS:

$$\text{Rec(1)}\ \frac{E \xrightarrow{\alpha} E'}{\mu X.E \xrightarrow{\alpha} E'\{\mu X.E/X\}} \qquad \text{Rec(2)}\ \frac{E\{\mu X.E/X\} \xrightarrow{\alpha} E'}{\mu X.E \xrightarrow{\alpha} E'}$$

The difference between these rules is the point at which substitution is performed. To see if a process $\mu X.E$ can perform a transition with rule (1), we first see if $E$ can do a transition, and then substitute the recursion variable *afterwards*. In rule (2), we *first* apply the substitution to $E$ and then see if the resulting process can do a transition. In guarded processes, such as $\mu X.a.X$, there is no difference between these rules. The difference occurs in unguarded processes, such as $\mu X.(a.X + X)$. If the first rule is applied, the process is capable of doing only a single transition $\xrightarrow{a}$ to $\mu X.(a.X + X)$, since the unguarded $X$ yields no transitions. In contrast rule (2), since it does the substitution first, would yield an infinite number of $\xrightarrow{a}$ transitions. Rule (2) is the "original" CCS rule, and the one on which all the standard proofs depend, in particular compositionality of $\mu X.E$ with respect to observation equivalence.

```
type PVar = ···
data Process a s m where
    − Fix point; μ X.E
    MU :: PVar → (Process a s m → Process a s m) → Process a s m
    UMU :: PVar → Process a s m → Process a s m
    − Recursion variable; X
    VAR :: PVar → Process a s m
    DVAR :: Process a s m
```

There are two possible ways of representing a fix-point in Haskell. The first representation, MU, uses a function to represent the equation. This approach is the most natural for Haskell since it uses the built-in substitution engine. Thus, for instance, the Process $\mu X.a.X$ would be represented as (in pseudo-code) `MU X (λx → a.x)`. The second representation, UMU, is slower during simulation, but more flexible for performing syntactic transformations. Rather than using Haskell functions, a process is represented as a regular process containing an instance of VAR wherever a variable is located. In this case substitution must be performed manually.

In my implementation I use both of these, the former for simulation and the latter for syntactic manipulation. Processes of the latter form can be "fixed" to turn them into the former. The reason for not simply using the former alone is due to the way functions work. If one wishes to perform a transformation Process a s m → Process a s m on a MU process, the transformation function must be composed onto the existing recursion function contained inside, as it is not possible to directly transform a function, only apply it to something. The transformation is then effectively hidden in the recursion function. If the transformation isn't idempotent then, during simulation, it will be applied over and over whenever the MU is expanded. Clearly this is inadequate, and therefore the latter form is used in these circumstances. The implementation of `step` is then as follows:

```
step (MU a f) = step (f (MU a f))
```

This follows rule (2), firstly substituting the MU expression into the process syntax and then calculating the transitions. Clearly, if the given process is unguarded (e.g. $\lambda x \to x$) this will loop infinitely. If required, rule (1) can be implemented thus:

```
step (MU a f) = [ Tran t (map (λ x → replaceDVar x (MU a f)) ps)
                | Tran t ps ← step c (f DVAR) ]
step DVAR     = []
```

This relies on function `replaceDVar :: Process a s m → Process a s m → Process a s m`, which replaces every occurrence of the distinguished variable DVAR in the first process

with the second process. This alternative implementation calculates the possible transitions of the process with every occurrence of the recursion variable substituted with DVAR (which has no transitions). Each resultant process is then turned back into a MU expression by fixing it using replaceDVar.

### 8.2.6 Abstract Time

Now that we have fully explored the CCS process calculus implementation, it is possible to proceed to implement CaSE$^{ip}$ . The key difference of course is the addition of the new timing operators, so I first present the new syntax (cf. Chapter 6 Section 6.5):

```
data Process a c s m where
    -- The stalled process; 0
    NIL :: Process a c s m
    -- The stalled process; Δ
    DELTA :: Process a c s m
    -- The stalled process; Δ_σ
    DELTAC :: c → Process a c s m
    -- Observable action prefix; a.P
    OBS :: Action a s m → Process a c s m → Process a c s m
    -- Silent action prefix; τ.P
    TAU :: StateT s m Bool → StateT s m () → StateT s m Bool → Process a c s m
                                                            → Process a c s m
    -- Clock prefix; σ.P
    TICK :: c → ClockB s m → Process a c s m → Process a c s m
    -- (External) Choice / Summation; P + Q
    SUM :: Process a c s m → Process a c s m → Process a c s m
    -- Parallel Composition; P | Q
    PAR :: Process a c s m → Process a c s m → Process a c s m
    -- Restriction; P \ a
    RES :: a → Process a c s m → Process a c s m
    -- Renaming; P ⟦a ↦ b⟧
    REN :: Process a c s m → (a, a) → Process a c s m
    -- Clock renaming; P⟦σ ↦ a⟧
    CREN :: Process a c s m → CRen a c → Process a c s m
    -- Clock Hiding; P / σ
    HID :: Process a c s m → c → ClockB (BTree s) m → Process a c s m
```

Much of this is familiar, the differences are the addition of time oriented operators. I have introduced a further sort as a type parameter to mimic the clock sort $\mathscr{T}$ called c. Before I move on to redefining step for the new syntax, we must first consider how

monadic bindings work with respect to clocks. I have introduced two types to convey these, which are used in clock renaming and prefix respectively:

```
data CRen a c    = CRenI c a | CRenO c a ([Dynamic] → Dynamic)
data ClockB s m  = ClkBInput ([Dynamic] → StateT s m ())
                 | ClkBOutput (StateT s m [Dynamic])
                 | ClkBNone
```

I treat clock ticks as multi-party computations formed by combining together the bindings of all the constituent clock prefixes. Thus clock prefixes carry a ClockB, which allows three types of binding to be associated with a clock prefix. A clock tick can either be:

- a **broadcast** if each prefix is inputting via `ClkBInput`;
- an **aggregation** if each prefix is outputting via `ClkBOutput`;
- **null**, via `ClkBNone` in which case the given prefix does not contribute to the overall effect of the clock tick.

This has two implications: firstly all the clock prefixes in a clock's domain must agree on whether the clock is input or output, and secondly the data has to come from somewhere, or go somewhere, respectively. The data sink/source can be either the clock hiding or renaming boundary. The type CRen has two constructors which are used to define what the behaviour should be in the case of an input or output.

A clock prefix input takes a list of dynamic values and produces a computation in the StateT monad. A clock prefix output is a computation producing a list of dynamic values. When a clock tick is formed these are combined by concatenating the respective lists. I provide an instance of Monoid which allows concatenation of clock prefix bindings:

```
instance (ActionHeader h, Monad m) ⟹ Monoid (ClockB h s m) where
  mappend (ClkBInput f) (ClkBInput g) = ClkBInput (λx → f x >>= g x)
  mappend (ClkBInput f) _ = ClkBInput f

  mappend (ClkBOutput m1 n1) (ClkBOutput m2 n2)
    = ClkBOutput (do x ← n1
                     y ← n2
                     return (x ++ y))
  mappend (ClkBOutput m1 n1) _ = ClkBOutput m1 n1

  mappend ClkBNone b = b

  mempty = ClkBNone
```

If two clock bindings are unalike then the first of the two is used as the combination and the second is ignored. If both the computations are input bindings then a new computation function is produced by sequentially composing the two computations, both applied to the same input. If both are output bindings then the results of the two computations are concatenated. The `step` function uses `mappend` to combine the bindings of the clock ticks on either side of a parallel composition or summation to produce a new clock tick.

When the tick finally reaches the hiding or renaming boundary it should, as for regular input/output action, be composed with a complementary action. If a hiding boundary, a simple clock binding is present which should be complementary. The hiding binding does not have an intrinsic state, but in theory has access to the entire state-space tree below it. However, since the hiding operator doesn't really have its own individual state it is normally expected to supply a constant value as input, or perform an IO action with the output. The more useful binding is present in the clock renaming boundary. Here the binding maps a clock tick binding onto an action binding. If the tick represents an input using `CRenI` then the value is simply sent to all the consumer processes (i.e. all those agents with clock prefixes). If an output using `CRenO` a function is supplied which converts the list of Dynamics into a single value which is simply output via the channel. This system allows a neat approach to forming one-to-many synchronisations, since clock inputs form a real broadcast operator in the style of CBS (Prasad, 1991) or CSP (Hoare, 1985). Outputs allow the opposite, where data is gathered from multiple locations and sent out over a single channel.

Having discussed the way bindings work, we now proceed to look at how the semantics should be defined. My approach mirrors that of the Operational Semantics in Chapter 6, Section 6.3. I first define the three important sets, namely the instability set $\Sigma_E$, the Initial Clock set $\mathcal{T}_E$ and the Initial Action set $\mathscr{A}_E$. To do this I first define a de-

rived type called TopSet which represents the concept of a Set with a maximal value ("full"). This is needed since the instability set can be equal to the countably infinite set $\mathscr{T}$ for a process like $\Delta$.

```
type TopSet a = Maybe (Set a)

tsEmpty :: TopSet a
tsEmpty = Just (Set.empty)

tsBot :: TopSet a
tsBot = tsEmpty

tsTop :: Ord a ⟹ TopSet a
tsTop = Nothing
```

I also define the standard set functions such as union (`tsUnion`), intersection (`tsInter`), membership (`tsElem`) and so on. This type is then used to define `stallC`, which in turn uses the other two sets `initA` and `initC`.

```
data Act a = I a | O a | T
initA :: (Ord a, Ord c) ⟹ Process a c s m → Set (Act a)
initC :: (Ord a, Ord c) ⟹ Process a c s m → Set c
stallC :: (Ord a, Ord c) ⟹ Process a c s m → TopSet c
```

For the sake of brevity (and as they are rather uninteresting) I don't provide full definitions here, they essentially reiterate the original definitions using Haskell syntax. With these three sets in hand it is now possible to give the new definition of `step`. First I redefine TransitionLabel to add the new clock transition type:

```
data TransitionLabel a c s m = Tick c (ClockB (BTree s) m)
                             | Observable (Action a (BTree s) m)
                             | Silent (SilentType a c)(StateT (BTree s) m Bool)
                                                      (StateT (BTree s) m ())
                                                      (StateT (BTree s) m Bool)
data SilentType a c          = ClockTick c | External | Synchronize a
```

The new TransitionLabel simply adds a clock tick transition with associated binding using ClockB. Notice that as before I am using a BTree, so data is gathered from the various parts of the state-space tree. I can now proceed to redefine `step`:

```
step :: (Ord a, Ord c, Monad m, Monoid s) ⟹ Process a c s m → [Transition a c s m]
step NIL = []
step DELTA = []
step (DELTAC c) = []
step (OBS l p') = [Tran (Observable (actLeaf l)) p']
step (TICK c m p') = [Tran (Tick c (clockBLeaf m)) p']
```

The basic constructors are very simple and similar to those experienced so far. As with actions, when a clock tick is formed the binding is raised to the leaf level of the state-space tree and placed into the transition. This is done using the analogous function `clockBLeaf`.

**Summation**

```
step (SUM p q) =
 let tp = step p; tq = step q
     iap = initA p; iaq = initA q; icp = initC p; icq = initC q in
     [ tpq | tpq@(Tran tl _) ← tp ++ tq, not (isTick tl) ] ++

     [ tp' | tp'@(Tran (Tick c _) _) ← tp
           , c `tsNotMember` uq, c `notMember` icq ] ++
     [ tq' | tq'@(Tran (Tick c _) _) ← tq
           , c `tsNotMember` up, c `notMember` icp ] ++
     [ Tran (Tick c1 (mappend m n)) (SUM p'' q'') | Tran (Tick c1 m) p'' ← tp
                                                  , Tran (Tick c2 n) q'' ← tq
                                                  , c1 == c2 ]
```

The semantics of SUM is somewhat more involved than before. The action transitions are the same – the first list gathers together transitions from both sides which are not clock ticks (indicated by the condition `not (isTick tl)`). There then are three ways of forming a clock tick, corresponding to the three SOS rules tSum1-3. The first is a tick originating from the left hand side, with the precondition that the clock is not in the instability set `up` or initial clock set `icp` of the right hand side. The second is simply the reflection of the first. The third is when a clock ticks on both sides simultaneously, causing both sides to advance. Notice in this case that the two clock bindings `m` and `n` are combined using the Monoid function `mappend`.

### Parallel Composition

```
step (PAR p q) =
let tp = step p; tq = step q
    t =
    [Tran (Observable $ actRight o) (PAR p qq)
    |Tran (Observable o) qq <- tq] ++

    [Tran (Observable $ actLeft o) (PAR pp q)
    | Tran (Observable o) pp <- tp] ++

    [Tran (Silent t (stateLeft pr) (stateLeft m) (stateLeft po)) (PAR p' q)
    | Tran (Silent t pr m po) p' <- r] ++

    [Tran (Silent t (stateRight pr) (stateRight m) (stateRight po)) (PAR p q')
    | Tran (Silent t pr m po) q' <- s] ++

    [Tran (Silent (Synchronize a) (return True) (stateRight o >>= stateLeft.i)
                  (return True)) (PAR p' q')
    |(Tran (Observable (Input a i)) p') <- tp
    ,(Tran (Observable (Output b o)) q') <- tq, a == b] ++

    [Tran (Silent (Synchronize a) (return True) (stateLeft o >>= stateRight.i)
                  (return True)) (PAR p' q')
    |(Tran (Observable (Output b o)) p') <- tp
    ,(Tran (Observable (Input a i)) q') <- tq, a == b]

    icp = initC p; icq = initC q; up = stallsC p; uq = stallsC q
    ias = not $ T `elem` initA (PAR p q)
    in (if ias
        then [ Tran (Tick cn1 (mappend (clockBLeft m) (clockBRight n))) (PAR p'' q'')
             | (Tran (Tick cn1 m) p'') <- tp
             , (Tran (Tick cn2 n) q'') <- tq
             , cn1 == cn2] ++
             [ Tran (Tick c (clockBLeft m)) (PAR p' q) | (Tran (Tick c m) p') <- tp
                                                       , c `tsNotMember` uq
                                                       , c `notMember` icq] ++
             [ Tran (Tick c (clockBRight m)) (PAR p q') | (Tran (Tick c m) q') <- tq
                                                        , c `tsNotMember` up
                                                        , c `notMember` icp]
        else []) ++ t
```

This semantics of PAR is rather long and involved, so I'll break it down. Firstly t is set to contain all the non-clock transitions, using the same rules as for standard CCS. This accounts for more than half of the function, but is included for completeness. The majority of the code dealing with clock transitions comes after the **in** keyword. The value ias is set to False if $P \mid Q$ can perform a $\tau$, checked using the initial action set of the composition initA (PAR p q). This value is then used to decide if any clock ticks should be added. If so a similar algorithm to that used for the SUM clause is used, the difference being that no reductions take place (i.e. the parallel operator remains in place). The complete set of transitions therefore comprises the non-clock transitions concatenated with the clock transitions, provided no $\tau$ transitions are present. The monoid function mappend is again used to combine the bindings, but this time m and n are transformed into the tree context, using functions clockBLeft and clockBRight, which enable to bindings to find their respective agent's state.

### Clock Hiding

```
step (HID p cs m) =
  [Tran (compClkB cn m n) (HID p' cs m) | Tran (Tick cn n) p' ← tp, cn == cs] ++
  [Tran (Silent t pr mo) (HID p' cs m) | Tran (Silent t pr mo) p' ← tp] ++
  [Tran (Observable a) (HID p' cs m) | Tran (Observable a) p' ← tp] ++
  (if (null $ [1|Tran (Tick cn _) _ ← tp, cn == cs])
    then [Tran (Tick cn mo) (HID p' cs m) | Tran (Tick cn mo) p' ← tp
                                          , not $ cn == cs]
    else [])
  where tp = step p
```

For the semantics of HID the first list is formed from any clock ticks on the hidden clock. The silent action is produced by composing the clock bindings of the tick and hiding operator using compClkB. The function compClkB produces a silent action TransitionLabel containing a binding made by composing the output binding with the input binding, in the same way that an action synchronisation works. To reiterate, this relies on the bindings in the clock tick and operator being complementary, a null binding is produced otherwise.

The next two lists simply allow silent and observable transitions through the operator without altering them. The final list is formed by allowing other clock ticks through without changing them, provided there is no tick on the hidden clock, in which case the produced $\tau$ would hold up all other clocks.

### Clock Renaming

```
step (CREN p rn@(CRenI c a)) =
 map (λx → case x of
              Tran (Tick t b) p →
                let ta = if (t == c)
                           then
                            case b of
                              ClkBInput f → Observable (Input a (λx → f [x]))
                              _ → Observable (Input a (λx → return ()))
                           else Tick t b
                in Tran ta (CREN p rn)
              Tran a p′ → Tran a $ (CREN p′ rn)) tp
 where tp = step p
```

Renaming a clock to an input action is the simpler of the two options. The function maps over the list of transitions `tp`, and upon finding a tick on the correct clock converts it into an input action. Specifically, when a tick on clock `c` is encountered with an input binding `f` an input action is produced whose binding applies its input value `x` to the clock input binding in a singleton list. This means that every clock prefix performing an action will input a single value when the input action synchronises with a suitable output. Bindings that do not match result in an input action with a null binding. All other types of actions are left alone.

```
step (CREN p rn@(CRenO c a f)) =
 map (λx → case x of
              Tran (Tick t b) p →
                let ta = if (t == c)
                           then case b of
                                  ClkBOutput m →
                                    Observable (Output a (m >>= return . f))
                                  _ → Observable (Output a (return (toDyn ())))
                           else Tick t b
                in Tran ta (CREN p rn)
              Tran a p′ → Tran a $ (CREN p′ rn)) tp
 where tp = step p
```

Renaming a clock to an output action is very similar, but in addition it uses the aggregator function `f` in the renaming operator to produce a single output from the list

of values produced by the clock tick. The resulting binding first executes the clock tick binding and then applies the aggregator to the list returned. Again, if the clock binding does not match, a null output is produced.

The remainder of the `step` cases relate to the standard non-clock related operators, such as restriction. I don't list these as the only difference is an additional transition list in each one which simply allows all clock transitions through without altering them. This completes the discussion of the CaSE$^{ip}$ implementation. I have now introduced a framework which will allow CaSE$^{ip}$ processes to be given a semantics and, when equipped with appropriate bindings, executed.

## 8.3 Verification

Having described how the process calculus CaSE$^{ip}$ can be implemented in Haskell, I now proceed to form a verification framework. This framework provides the components needed for finite-state based verification of processes. It is a general framework and hence can be applied to a wide variety of process models, not just CCS and CaSE$^{ip}$. This gives rise to my plugin based process experimentation environment, ConCalc, which I describe in Section 6.6.10.

### 8.3.1 Labelled Transition Systems

Verification of labelled transition systems is performed completely independently of the Process Calculus itself. Each LTS $(\mathcal{P}, \mathscr{A}, \rightarrow)$ consists of two data-types representing $\mathcal{P}$, the state label alphabet, and $\mathscr{A}$, the transition label alphabet. These two data-types are then associated with the transition relation via the following type-class:

```
class (Eq s, Trans (LTSTran s), Ord s, Monoid (LTSCtx s)) ⟹ LTS s where
  type LTSTran s
  type LTSCtx s

  prepareState :: s → s
  step         :: LTSCtx s → s → [(LTSTran s, s)]
  stateCtx     :: s → LTSCtx s
```

The type-class LTS brings together the transition type and state type, and defines the transition relation. LTS could be a three parameter type-class, parametrised over the state alphabet, transition alphabet and state context alphabet, but this would be cumbersome. As it is, I'm using GHC's *class associated type synonyms* (Chakravarty et al., 2005) to hide the other two parameters within the class (see Chapter 3, Section 3.6), since in most instances they are derivatives of the state type anyway. By "derivative" I mean

that a particular state type will often have an associated transition label type which is only relevant to that particular state type, and not others.

LTSTran is a type function from the state type to the type of transitions ($\not\alpha$), and LTSCtx is a type function to the LTS *context* type, which is used when an invisible but important context is required to determine the transitions (such as the clock context in CaSE or a process variable definition context). The class provides a total of 3 functions for manipulating a labelled transition system. The most important function is `step` which takes an LTS context and state, and returns a list of transition label / new state pairs – it embodies the transition relation $\rightarrow$. In addition `prepareState` is a general pre-step state function, which allows a transformation to be performed on the state, such as pruning (should there be structural congruence rules, for instance). Finally, `stateCtx` returns the default context of the given state.

Closely associated with the LTS class is the Trans class, which is used to describe transition labels:

```
class Eq t ⟹ Trans t where
    silentTrans    :: t
    isSilentTrans  :: t → Bool
    isInputTrans   :: t → Bool
    isOutputTrans  :: t → Bool
    isTimeTrans    :: t → Bool
```

This type-class allows classification of different types of transitions. Specifically silent actions, inputs, outputs and time transitions. It enables, for instance, observation abstraction by identifying which transitions are silent. These two type-classes together form the basis for the verification functions. For instance, I provide a basic function for extracting the traces of a process:

```
traces :: LTS s ⇒ s → [[LTSTran s]]
```

This takes an initial LTS state and produces a (potentially infinite) list of all the possible partial traces. In the following section I will produce an algorithm for generating transition graphs, and then use this for verification.

### 8.3.2 Graph Generation

An instance of LTS makes it possible to generate transition graphs for a given process. The idea is that process is simulated using the `step` function, and the entire behaviour of the process is recursively mapped out a state at a time. Generating a graph is effectively the hardest part of verification, and certainly a major bottle-neck. The major problem is how to decide if a given transition is actually a loop back to a state node in the graph

already considered, or if a new node is needed. This comes down to a problem of implementing an efficient syntactic equality checker, which I will explore later in this section.

To provide a starting point, I am using a version of the Haskell *Functional Graph Library* (FGL), created by Erwig (2001). FGL is a well featured library for manipulating functional representations of labelled directed graphs. In particular it provides a backend for *Graphviz* (Ellson et al., 2004) through the DOT language. Graphviz is a flexible utility for automatic drawing of various types of graphs. I have used it extensively in Thesis for the purpose of drawing directed graphs to represent transition systems (in particular see Section 8.4.3).

FGL provides a type-class Graph which is instantiated for each graph type `gr`. The graph type is binary kinded, parametrised over the node label type and edge label type, respectively. The standard definition for a graph in FGL is essentially a map from nodes to the "context" of that node, where a context is the incoming and outgoing edges. Each node is uniquely identified in the graph by the type Node, which is just a synonym for Int.

```
class Graph gr a b where
  empty    :: gr a b
  isEmpty  :: gr a b → Bool
  match    :: Node → gr a b → Decomp gr a b
  mkGraph  :: [LNode a] → [LEdge b] → gr a b
  labNodes :: gr a b → [LNode a]
  labEdges :: gr a b → [LEdge b]    ···
```

This partial definition shows some of the functions available in the basic Graph class. For reference, the type LNode is a labelled node, LEdge is a labelled edge and Decomp is a graph decomposition, containing a node, its context, and the remaining graph. The class definition I show above is slightly different to the standard FGL class, as it includes the label and transition type in the class head. The reason for this is that in order to generate a transition graph it is necessary to place restrictions on these, but the standard class won't allow this. Therefore I have exposed them. In addition to Graph there is a class called DynGraph, which represents graphs which can be altered.

The graph generation is performed within a State monad, LtsM:

```
type LtsGraph gr l s = gr s l
type LtsM gr l s k = State (LtsGraph gr l s, LtsGraph gr l s, Int, Int, Map s Node) k
```

The state contains two graphs used during generation. They have identical nodes, one for each state in the graph, but different transitions. The first contains the visible transitions, whilst the latter contains silent transitions – this helps in producing weak

transition systems. The state also contains two integers; the first represents the highest Node index in the graph, and the second represents the maximum number of nodes allowed (which can be used to prevent infinite graphs), or 0 if there is no limit. Finally the state contains Map, which associates state labels with Node indexes, and is used to decide whether a given state is in the graph.

```
extendLtsGraph :: LTS s ⇒ LTSCtx s → s → Node → LtsM gr (LTSTran s) s ()
extendLtsGraph c s sn =
  let ts = step c s in
  do (_, _, sz, up, _) ← get
     if ((up > 0) && ((length ts + sz) > up))
       then return ()
       else mapM_ (λ tr → let t = transitionLabel tr; s' = snd tr in
                          do n' ← addNodeEdge sn t s'
                             maybe (return ()) (extendLtsGraph c s') n') ts
```

This function orchestrates the process of creating a graph by stepping the given state to get the possible transitions, and then iteratively adding new nodes to the graph. It takes an LTS context, a state, and an integer representing the index of the current state node. It first evaluates `step` using the context and state. It then extracts the state, and queries `up`, the maximum state node number. If adding all the new transitions to the graph will exceed this number, then no computation is done. Otherwise, `mapM_` is used to iterate over the list of transitions. Each iteration extracts the transition label and the next state. The computation `addNodeEdge` is then executed which will (possibly) add a new node to the graph, representing the state, and direct an edge from the current state to the new state. If a new node is added, this function will return `Just` its index, otherwise `Nothing`. If a new node index was returned, `extendLtsGraph` is recursively called for the new state node.

```
addNodeEdge :: LTS s⇒ Node → LTSTran s → s
                    → LtsM gr (LTSTran s) s (Maybe Node)
```

The body of `addNodeEdge` is long and rather uninteresting, so I will simply summarise it. Its job is to query the existing graph and see if the given node is already present. If it is, then a new edge is formed from the current node (passed in) and the existing node. Otherwise, a new node is created and its index returned. If the given transition is silent then it is added to the silent graph in the state, otherwise it is added to the visible graph.

The advantage of storing two graphs becomes apparent if one wishes to create a *weak graph*. If a strong graph is required, then all that is required is the edge sets of the two

graphs are combined. If weak, then we can capitalise on the split. In order to determine the definition of $\stackrel{\hat{\tau}}{\Rightarrow}$, the weak transition relation, we can take the reflexive transitive closure of the weak graph. This will give a graph which describes the states which may be bridged silently. It is then simply a matter of applying transition composition for each transition in the strong graph with the transitions in the weak graph, i.e. $\stackrel{\hat{\tau}}{\Rightarrow} \cdot \stackrel{\alpha}{\rightarrow} \cdot \stackrel{\hat{\tau}}{\Rightarrow}$. The weak graph is of particular use since it enables the efficient checking of weak bisimulation and reduction of a transition system to the minimally visible graph.

To conclude this Section, I return to a problem posed earlier – i.e. the definition of `addNodeEdge` depends on a how we check if a given state node is already present in the graph. There are several issues with this. Although the state of LtsM includes a Map from state labels to nodes, it is difficult to use. Map is a tree structure, with the values at the leaves, and ordered by the key values. Therefore, in order to use this an ordering algorithm is needed for process syntax. Without it, we have to revert to checking every node in the graph for equality individually. When one considers that typical Cashew-A processes consist of over a hundred individual agents, graph generation is going to be time consuming. Therefore an instance of Ord must be present for a process to be practically verifiable. This unfortunately is a rather inefficient process since it is necessary to descend through the process syntax of at worst every process in the graph so far every time `addNodeEdge` is evaluated[2]. Therefore it is vital that graph generation is performed sparingly.

### 8.3.3 Timed Transition Graphs

Having introduced a method for transition graph generation for process calculi, I now briefly turn back to an issue which I considered in Chapter 6 Section 6.7. In order to stream line generation of a process semantics it should be possible to shortcut some of the processes to graphs. Then instead of generating one large transition graph, smaller ones can be generated and combined, thus leading to a speed up (particularly when the smaller graphs are *minimised* – see Section 8.3.6). However, whilst this is easy with CCS processes since they are isomorphic with labelled transition systems, this is not the case with CaSE$^{ip}$ and other abstract timed process calculi. Hence, as I said back in Chapter 6, we need a kind of *symbolic transition graph* to represent the extra clock context data. Specifically, each state needs to store the instability set $\Sigma_P$ which will allow composition with further processes. I therefore extend the CaSE$^{ip}$ syntax thus:

[2] In fact my current implementation simply uses the string representation for comparison, since this turns out to be a very efficient algorithm!

```
data Process a c s m where
    ...
    -- Expanded transition graph
    GRAPH :: (LtsGraph g (TransitionLabel a c h s m) (TopSet c, Process a c s m), Node)
          -> Process a c s m
```

The new GRAPH construct holds a timed transition graph which stores a TopSet at the states, representing $\Sigma_P$, together with the process itself. This is paired with a Node which references the current state. The implementation of `step` can then be expanded to simply extract the outgoing edges of the current state to give the processes possible transitions, whilst also updating the current state variable. This is of course much quicker in many cases than recursing through the process syntax to determine the transitions.

### 8.3.4 Partition Refinement

In this section I describe my *partition refinement* algorithm for Haskell. Partition refinement (Paige and Tarjan, 1987) is a method of dividing the states of a transition system into blocks of equivalent states by comparing the transitions each state may do and refining based on which states' transitions lead to the same blocks. It arises from the observation that two processes are bisimilar (i.e. of the same equivalence class) when they have the same *action maps*. An action map assigns to each action the set of equivalence classes to which performing that action may lead. This can be well illustrated by the categorical definition of bisimulation:

$$\begin{array}{ccc} P & \sim & Q \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\alpha} \\ P' & \sim & Q' \end{array}$$

That is, whenever $P$ can do an action $\alpha$ to a process (or equivalence class) $P'$, $Q$ must have a matching $\alpha$ transition which leads into a member of the same equivalence class, $Q'$ (and conversely). The partition refinement algorithm takes the set of all states of an LTS and iteratively splits it up into blocks based on whether this condition holds. Once no further partition refinement is possible, the set of blocks in the partition is the set of equivalence classes.

Haskell provides an elegant way of partitioning such a graph, and I have implemented partition refinement using several functions. The first function to be considered is `actMap`, which calculates a block's action map:

```
actMap :: (Ord t, Graph gr s t) ⇒ gr s t → Set Node → Set (Set Node)
                                → Map Node (Map t (Set (Set Node)))
actMap g b parts = Map.fromList $ map (λp . (p, Map.fromList
                                  $ map blocks (sucMap p))) (Set.toList b)
 where
 sucMap :: s → Map t [s]
 sucMap p = Map.toList $ Map.fromListWith (++) $ map (λ(x, y) . (y, [x])) $ lsuc g p
 blocks (a, ps) = (a, (Set.filter (λc . or $ map (λp′ . p′ `Set.member` c) ps) parts))
```

The function `actMap` is the backbone function in my partition refinement algorithm. It takes the transition graph, together with the block being examined and the partition as it stands and calculates the action map for each state in the given block. This data comes in the form of a map of maps, the inner map from transition labels to block sets (i.e. the possible blocks the action may lead into) and the outer map from states to the inner map. Thus, each state in the given block is associated with a map from the action labels it may invoke to the set of blocks each transition can lead to.

The function works by first building an associative list from actions to lists of successor states for each state in the block (using `sucMap`). It then converts this into an associative list from actions to blocks, depending on which block each successor state is in (using `blocks`). Finally the list is converted into a map, associated with the state being queried and the final action map is returned.

The next function to be considered is `part`:

```
part :: Ord t ⟹ Map Node (Map t (Set (Set Node))) → Set (Set Node)
part m = case Map.keys m of
           [] → Set.empty
           k : _ → let
                     (n, n′) = Map.partition (== (fromJust $ Map.lookup k m))
                                             (Map.delete k m)
                   in
                   (Set.fromList (k : Map.keys n)) `Set.insert` (part n′)
```

This function takes the state/action-map map and partitions the states into groups with the same action map. For instance, if state $P$ can do only an $\alpha$, resulting either in a member of blocks $B$ or $C$, then $Q$ would be in the same block if and only if it also could only do $\alpha$ into blocks $B$ or $C$. States with different outgoing actions and resultant block are distinguished.

The function works by taking the first state from the map (i.e. the first key), gathering states with the identical action map, placing these states in a block and then proceeding with the next state. It finishes when there are no more states remaining, when all have been placed into a block.

```
partitionr :: (Ord t, Graph gr s t) ⟹ gr s t → Set (Set Node)
partitionr g = partition' g
               [Set.fromList $ nodes g]
               (Set.singleton (Set.fromList $ nodes g))
partition' :: (Ord t, Graph gr s t) ⇒ gr s t → [Set Node] → Set (Set Node)
                                    → Set (Set Node)
partition' _ [] parts = parts
partition' g (b : bs) parts =
 let newPart = part $ actMap g b parts
     newParts = (Set.delete b parts) `Set.union` newPart
 in if ((Set.size newPart) <= 1)
  then partition' g bs parts
  else partition' g (Set.toList $ Set.filter ((>1) . Set.size) newParts) newParts
```

Finally, these two functions put it all together. The second, `partition'` does the actual partition refinement. It takes the transition graph (`g`), block queue (`bs`) (i.e. the blocks remaining to be partitioned) and the partition so far (`parts`), and produces the optimal partition. It first uses `actMap` and `part` to partition the first block on the queue. If this partitioning produces a partition of only 1 block, it means it was optimally refined and partitioning moves onto the next block. Otherwise, it produces a new set of partitions by removing the old block and adding the new blocks. It then creates a new block queue using the new partition and recurses using the new partition. If the queue becomes empty, it means the partition is optimal and thus it is returned.

The remaining function, `partitionr` is the function which is actually called by the user. It takes the transition graph and builds an initial partition and block using all the states in the graph and then proceeds to evaluate `partition'`. This completes the discussion of my partition refinement algorithm, except to note that the partition refinement algorithm in the section is a proof of concept and not optimised. In latter work I have optimised partition refinement to, for instance, work with transition graphs where both states and transitions are enumerated as integers. This allows Haskell's more efficient IntSet and IntMap data types to be used for storage of the partition, which are purpose built for use with integers and not arbitrary ordered types. This leads to a faster decision. There are several applications of the partition refinement algorithm, I highlight the two main ones.

### 8.3.5 Bisimulation Checking

Naturally enough, partition refinement can be used to create an efficient bisimulation checker – indeed this is one of its main uses. For the purposes of comparison I have actually implemented two algorithms, one of which uses partition refinement, and one which uses a recursive depth-first search algorithm. The two algorithms are useful for

comparing the two techniques, and provide better performance under particular conditions. Also note that both of these algorithms can be used for either weak or strong bisimulation checking. Recall from Chapter 6 (specifically Lemma 6.6.10) that checking for weak bisimulation on a strong graph is identical to checking bisimulation on the weak graph. Therefore, to perform a weak bisimulation check, the algorithm is simply applied to the weak graph instead of the strong graph.

Checking bisimulation through partition refinement is fairly trivial. It is simply a case of merging the two graphs into a single graph with disjoint node names, performing partition refinement and checking if the two start state nodes are in the same block. If so, then the two graphs are bisimilar. The following function, `bisimPR`, implements this:

```
bisimPR :: (Ord b, DynGraph gr a b, Monad m) ⇒ gr a b → gr a b → (Node, Node)
                                                → m (Set (Node, Node))
bisimPR g1 g2 (n1, n2) =
  let g1ns = nodes g1
      − Generate a map from nodes in graph 1 to new nodes in graph 2
      ns = Map.fromList $ zip g1ns $ newNodes (length g1ns) g2
      ns' = Map.fromList $ zip (newNodes (length g1ns) g2) g1ns
      g2nns = map (λ(n, l) → (fromJust $ Map.lookup n ns, l)) $ labNodes g1
      g2nes = map (λ(m, n, l) → (fromJust $ Map.lookup m ns
                                 , fromJust $ Map.lookup n ns, l)) $ labEdges g1
      − Build the combined graph
      gc = insEdges g2nes $ insNodes g2nns g2
      − Get a list of partitions
      ps = Set.toList $ partitionr gc
      bns n = zip (repeat n) ( map (λx → fromJust $ Map.lookup x ns')
                                 $ filter (>= (fromJust $ Map.lookup n1 ns))
                                 $ Set.toList $ fromMaybe Set.empty
                                 $ find (Set.member n) ps )
  in if (isJust $ find (λs →((fromJust $ Map.lookup n1 ns) `Set.member` s)
                                 && n2 `Set.member` s) ps)
     then return $ Set.fromList $ concat $ map bns g1ns
     else fail "bisimPR : No bisimulation exists"
```

It takes two graphs and a start node for each graph, and returns a set of node pairings wrapped in a monad (for the situations when the algorithm fails to find a bisimulation). The set of node pairings represents the bisimulation relation $\mathcal{R}$. The first part of the function combines the two graphs, which requires that all nodes in the first graph must be given new node identifiers so they don't clash with the second graph. Naturally a set of edges must also be produced between these new nodes. The new graph is then stored

at gc and fed through the partition refinement algorithm producing a partition at ps. I then create a function called bns which will take a node from the first graph and produce a list of bisimilar pairings for that node with the second graph.

Finally, the function checks to see if there is a block which both start states are members of (meaning they are bisimilar). If this is the case then the bisimulation pairings are produced by applying bns to every node in the first graph and placing the results in a set. If the start nodes are not bisimilar then a failure is returned.

The alternative algorithm using a recursive search algorithm is implemented in the function bisimRC shown below:

```
bisimRC :: (Eq b, Graph gr a b) ⇒ gr a b → gr a b → (Node, Node)
                                → StateT (Set (Node, Node)) Maybe ()
bisimRC g1 g2 (n1, n2) =
 do b ← get
    if ((n1, n2) `Set.member` b) then
      return ()
    else
      let t1 = lsuc g1 n1; t2 = lsuc g2 n2 in
          if (listEquiv (map snd t1) (map snd t2))
          then do put (Set.insert (n1, n2) b)
                  let ts1 = map toMap $ groupBy' (λx y → (snd x) == (snd y)) t1
                      ts2 = map toMap $ groupBy' (λx y → (snd x) == (snd y)) t2
                      b' = bisims ts1 ts2
                  mapM_ (λ(xs, ys) → existsBisim1 xs ys) b'
                  mapM_ (λ(xs, ys) → existsBisim2 xs ys) b'
          else fail "No bisimulation"
 where
  existsBisim1 t1 t2 = mapM_ (λy → msum $ map (λx → bisimRC g1 g2 (y, x)) t2) t1
  existsBisim2 t1 t2 = mapM_ (λy → msum $ map (λx → bisimRC g1 g2 (x, y)) t1) t2
```

It has a similar type to the former algorithm, but works in a state monad to produce the set of equivalent states. The state will be gradually built up during execution and is used to see which pairings have already been made. The algorithm begins by checking if the given node pairing is already in the state. If so then no further work is needed. Otherwise, it proceeds to check if the two states are bisimilar. It first extracts the successor edges of each node (the outgoing transitions) and places them in t1 and t2. Each of these is a list of node/transition label pairings. If the set of outgoing transition labels is not the same (checked using listEquiv), then obviously no bisimulation can exist and a failure is returned.

Otherwise the algorithm goes about attempting to see if each state from every transition in one graph, is equivalent to each state in the other graph with the same transition label. The pairing of the two states is first added to the monad's state. The action maps

of each state are then calculated (i.e. a map from transition labels to a list of states that each transition leads to) and the results placed in `ts1` and `ts2`. The bisimulations which need to be checked are then calculated by `bisims` which produces a list of all necessary pairings. Finally each pairing is fed recursively through `bisimRC` using `mapM_`, a form of monadic map which will stop if one of the pairing checks fails. If no failure occurs, the end result will be a complete set of pairings for the graph.

In testing, the recursive algorithm frequently proves to be faster than partition refinement (even when optimised), though with a few caveats. Firstly it does not produce a complete set of bisimulations for every single state in the graphs, only those needed to prove the first two states are bisimilar. This makes the recursive algorithm useless for performing bisimulation *minimisation*, which requires a complete set of equivalence classes. Secondly the recursive variant is less memory efficient than partition refinement, since it stores a complete tree-expansion of the graph in memory. In contrast partition refinement is iterative and so only holds data on the block being considered. Therefore both these algorithms certainly have their places.

### 8.3.6 Minimisation

Like bisimulation checking, minimisation is a very important technique for process calculus. It involves taking "large" labelled transition system and applying some form of algorithm to reduce the states to the minimal number, such that it remains within the same equivalence class. A minimised graph will therefore always be bisimilar with its original, but having less states it is more efficient for verification purposes. Minimisation is particularly useful to Cashew-A'scomponent model as it allows one to extract the smallest equivalence class of a component. When combined with a weak graph, it allows the interface of a component to be extracted whilst abstracting any irrelevant detail.

A minimisation can be acquired by the application of partition refinement:

```
-- Minimisation algorithm, uses partition refinement
minimize :: (Ord b, DynGraph gr a b) ⇒ gr a b → gr a b
minimize g =
 let es = labEdges g
     s' = supStates g
     em = map (λ(f, t, l) →
               let f' = fromMaybe f $ Map.lookup f s'
                   t' = fromMaybe t $ Map.lookup t s'
               in insEdge (f', t', l) . delLEdge (f, t, l)) es
 in if (null $ nodes g)
      then g
      else (delNodes (Map.keys s') . (foldl (.) id em)) g
```

```
-- Get a map defining which states should be replaced by other states for minimisation
supStates :: (Ord b, DynGraph gr a b) ⇒ gr a b → Map Node Node
supStates g = Map.fromList $ concat
            $ map ((λs → map (λt → (t, head s)) (tail s)) . Set.toList)
            $ Set.toList $ partitionr g
```

The main function is `supStates` which performs partition refinement and uses the partition to create a map from old states to minimal equivalent states. Specifically it maps over the partition and for each block creates a mapping from each state to the first in the set. This is then used by the function `minimize` to produce the minimal set of states and map the transitions to act on only these states.

### 8.3.7 Alternating Simulation

*Alternating Simulation* (Alur et al., 1998) is a semantic preorder which I briefly introduced in Chapter 6, Section 6.8. It provides a basis for checking compatibility by ensuring that a component provides more inputs and fewer outputs than the template at each state. It is therefore also an ideal relation for checking choreography conformance. In this Section I provide a basic Haskell implementation of the relation.

```
altsim :: (Eq b, Graph gr a b) ⇒ gr a b → gr a b → (Node, Node) → (b → Bool)
                               → (b → Bool) → StateT (Set (Node, Node)) Maybe ()
altsim p q (u, v) isIn isOut =
do b ← get
   if ((u, v) `Set.member` b)
     then return ()
     else let ut = lsuc p u; vt = lsuc q v
              extenip = nub $ filter isIn $ map snd ut
              extenop = nub $ filter isOut $ map snd ut
              exteniq = nub $ filter isIn $ map snd vt
              extenoq = nub $ filter isOut $ map snd vt
          in if ((all (`elem` exteniq) extenip) && (all (`elem` extenop) extenoq))
               then do put (Set.insert (u, v) b)
                       let xs = map (λn → altsim p q n isIn isOut)
                                    (listProduct us' vs')
                       if (null xs) then return () else msum xs
               else fail "No alternating simulation exists"
```

The function has a very similar type signature to the bisimulation algorithm. In addition it takes two predicates, one to check if a transition label is an input and one to check if one is an output. It produces a list of pairings if an alternating simulation can be

found. It follows the same structure as the recursive bisimulation algorithm. It uses four lists, the inputs and outputs the two graphs produce in the current states. It checks that every input provided by the second graph is also provided by the first, and that every output provided by the first is also provided by the second. If this is true then it recurses with each pair of outgoing states. Otherwise it fails to provide an alternating simulation. This algorithm is by no means optimal, but it provides a basic implementation.

### 8.3.8 A Process Experimentation Environment

*ConCalc* (Concurrency Calculator) is an interactive environment for process experimentation, based on the implementation work already covered in this Chapter. I primarily created it to be an aid to defining the semantics in Chapter 7. Previous experience has shown that in a clock-oriented calculus it is very easy to make mistakes, particularly in terms of which clocks are enabled, and which are disabled. Indeed our previous semantics (Norton et al., 2005) has several errors in it, and thus ConCalc has enabled a much more robustly designed semantic framework to be constructed. It has also allowed the process calculus semantics to be thoroughly tested.

ConCalc provides a command-line interface into which process expressions may be entered, tested for bisimilarity and drawn as graphs. A typical interaction with the ConCalc command-line may go like this:

```
  __   __   _  _   __           _     __
 / _\ /  \ | \| | / _\   /\    | |   / _\
/ /  / /\ \|  \ |/ /    /  \   | |  / /
\ \_ \ \/ /| \  |\ \_  / /\ \  | |_ \ \_
 \__\ \__/ |_|\_| \__\/_/  \_\\___| \__\

      The Concurrency Calculator
              Version 0.1.20081217

Loading CaSE_ip ...
Calculus  : Calculus of Synchronisation and Encapsulation + ip
Author(s) : Simon Foster
Variant   : Full
CaSE_ip> :graph a.(T.b'.0 + T.c'.d.0)
```

The commandline provides a number of commands for manipulating processes, the most imporant of which is graph generation.

This interaction produces the graph shown in Figure 8.2. The self-transitions in this graph marked with !! bear witness to the fact that this is a timed transition system. This is a special clock which is used to decide whether a state is patient or not. A !! transition is produced whenever only a subset of the clocks is stalled. This is related to the deficiency

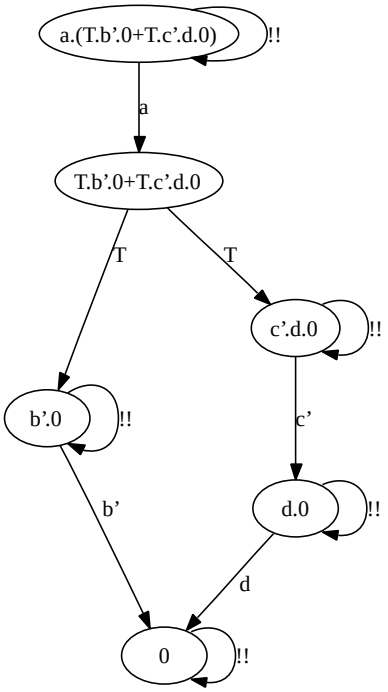



Figure 8.2: A typical ConCalc graph

of the standard LTS model in representing timed transition systems which I described in Chapter 6, Section 6.7 and also touched on in Chapter 7, Section 7.2.10. This !! clock is equivalent to the distinguished $\delta$ clock described in the latter reference.

ConCalc is *plugin* oriented, and supports any LTS based process calculus implemented in Haskell. In the above interaction ConCalc uses the `CaSE_ip` plugin, which implements the LTS semantics of CaSE$^{ip}$. A plugin must provide two things:

1. An instance of the type-class Proc which provides process variable functions and is a super-class of LTS;

2. A function `procCalc` of type ProcCalc which provides parsers and meta-data for the process calculus.

```
class (LTS p, Show p, Typeable p, Show (LTSTran p), Ord v, Show v, Ord (LTSTran p))
    ⟹ Proc p v | p → v where
  procHasVar   :: v → p → Bool
  procSubstVar :: v → p → p → p
  procFix      :: v → p → (Maybe p)
  procVar      :: v → p
```

Proc is parameterised over `p`, the type of process (Process in the case of CaSE$^{ip}$ or CCS) and `v` the type of process variable, which should be determinable from the process

type. The class, aside from providing a number of super-classes which allow LTS based facilities and visual output, allows manipulation of a process's variables. The functions listed:

- determine if a given process contains a given variable;
- perform a substitution for a given process and variable;
- fix a process over a particular variable (if possible); and
- return a process variable for the given variable

respectively. Variables are a central feature of ConCalc, in that a process may be assigned to a variable and these variables may then be used in other processes. These functions are used to employ late binding on the command-line.

ConCalc has been invaluable during the development of the work in this Thesis for experimenting with different process calculus variants. Since it is plugin oriented it has enabled me to compare and contrast the different process calculi and discover any flaws. For instance, using ConCalc I was able to discover that CaSE$^{mt}$ from Chapter 6 Section A.2 made clock choice too verbose, and was prone to causing infinite state-systems, due to its use of recursion to describe patience.

Another important facility which ConCalc provides is bisimulation checking using the algorithm described in Section 8.3.4. Below is an interaction with ConCalc which checks for weak bisimulation between $a.(\tau.b.\mathbf{0} + c.\mathbf{0}) + c.a.\mathbf{0}$ and $(a.\overline{z}.0 \mid z.b.\mathbf{0} + c.\mathbf{0}) \setminus z$.

```
CaSE_ip> a.(T.b.0 + c.0) + c.a.0 ~~ (a.z'.0 | z.b.0 + c.0)\z

(a.(T.b.0+c.0)+c.a.0,(a.z'.0|z.b.0+c.0)\z)
(T.b.0+c.0,(z'.0|z.b.0+c.0)\z)
(b.0,(0|b.0)\z)
(0,(z'.0|0)\z)
(0,(0|0)\z)
(a.0,(a.z'.0|0)\z)

0.002188s
```

Syntactically `a'` refers to an output on $a$ and `T` is a $\tau$. The infix operator $\sim\sim$ is weak bisimulation, $\approx$, whilst strong bisimulation can be checked for with operator $\sim$. tries to find a bisimulation via partition refinement, and if one exists displays the associated process pairings, followed by the amount of time it took to make the decision. Otherwise it displays `No weak bisimulation exists`.

## 8.4 Towards an implementation of Cashew-A

### 8.4.1 Overview

I conclude this Chapter by looking at what is the eventual aim of this implementation work, an execution semantics and verification environment for Cashew-A. As such this work is incomplete as only a fragment of the Cashew-A language has been implemented. However, what it demonstrates is the viability of my approach as a whole. Clearly if the semantic framework works then correct extensions of it will work also. To reiterate, I have a two-fold aim for the semantics presented in Chapter 7:

1. Verification of composite Web service descriptions using a variety of finite-state modelling checking techniques;
2. Execution of said descriptions by application of monadic bindings to the processes to realise data flow and message flow.

In Section 8.3 I introduced a variety of verification tools which I have developed, most of which centre on partition refinement. Indeed, minimisation is one of the most important techniques in allowing semantic generation in Haskell. As can be expected the graphs generated from the Cashew-A semantics are very large, maybe even prohibitively large. Although arguably this is due to an over reliance on non-deterministic CCS actions to drive orchestration scheduling (though a degree of non-determinism is always desirable), minimisation provides a way of optimising graph generation and verification. The idea is to minimise the generated graph at each level of abstraction in the Cashew-A syntax tree, thus avoiding an over-large number of states. These graphs will eventually be cached and used for regeneration of the workflow semantics, thus applying the compositionality property to cut down on unnecessary work.

Whilst a verification method is already available, the work presented in this Section is far from complete. As such many questions remain unanswered for how exactly the execution semantics should be formed. As we saw back in the Web services literature review in Chapter 2, WSMO employs *Abstract State Machines* (Börger, 1999; Fensel et al., 2007) which provide a common framework for giving a low-level description of an orchestration. As such, I have yet to formulate such a translation and clearly it is a non-trivial problem to combine data from the process calculus with Semantic Web service descriptions to give an executable ASM. Thus, for the time being I concentrate more on a general monadic approach, for which I hope an ASM-based frontend may be given. I will therefore provide a partial formulation of the Cashew-A operational semantics from Chapter 7 in Haskell, to demonstrate its feasibility and elegance.

### 8.4.2 Semantic Generation Framework

The overall idea of the semantic generation framework is illustrated in Figure 8.3. Each part of the workflow will be decomposed generated into a CaSE$^{ip}$ transition graph. The

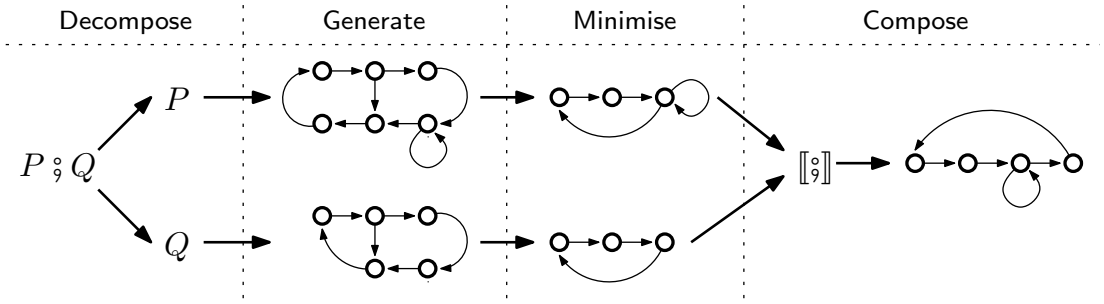



Figure 8.3: Overview of semantic generation

graph will then be minimised and composed via the appropriate semantic combinator to produce an overall transition graph. This process proceeds recursively until the semantics of the whole system is generated. The generation of the Cashew-A semantics centres around a collection of recursive functions, similar to those presented in Chapter 7. I first provide a collection of data types to represent the different parts of the Cashew-A syntax.

```
data Workflow    = Wf WfName Accept ControlFlow [Dataflow] Offer
data ControlFlow = Nary Nary | Cf CPat
data Accept      = AOr Accept Accept | AAnd Accept Accept
                 | AFalse | ATrue | AElem CInput
data CPat        = Seq CPat CPat
data Offer       = OOr Offer Offer | OAnd Offer Offer
                 | OFalse | OTrue | OElem COutput
data Dataflow    = SDPP (PfName, COutput) (PfName, CInput)
                 | ADPP (PfName, COutput) (PfName, CInput)
                 | DWP Input (PfName, CInput)
                 | DPW (PfName, COutput) COutput
data CPat        = Seq CPat CPat
                 | SPar CPat CPat
                 | APar CPat CPat
                 | Cho CPat CPat
                 | Iter CPat CPat
                 | Skip
                 | Halt
                 | Yield
                 | Perf Performance
```

The type Workflow is equivalent to the BNF definition of $\mathcal{W}$ in Chapter 5. Similarly, Accept, Offer, ControlFlow and Dataflow are equivalent to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$. I split the control flow up into the algebraic patterns and the n-ary patterns, though the latter are left

unimplemented for now. With types for representing the Cashew-A language, I also require several types to represent the various clocks and channels in Cashew-A semantics:

```
data CAction  = Activate Mod | Ready Mod | Execute Mod | Done Mod
              | Deactivate Mod | RelInput CInput | WfInput WfName CInput ···
data Mod      = I | J | N
data CClock   = WfClock WfName Int
              | PfClock PfName Int
              | WfCaClock WfName
              | WfYiClock WfName
              | ...
type CProcess m = Process CAction CClock (Map.Map CAction Dynamic) m
```

CAction represents the action sort $\mathscr{A}$ of a CaSE$^{ip}$ process simulating a Cashew-A workflow, for instance Activate represents channel **g** and Ready represents channel **r**. The associated type Mod represents the modifiers which can be applied to the protocol channels, such as the $i$ superscript represented as I for example. CClock is the clock sort $\mathscr{T}$ for a CaSE$^{ip}$ process. Workflow clock $\circledcirc_n^w$ is represented as WfClock w n, the performance clock is represented as PfClock p and the yield clock is represented as WfYiClock w. CProcess represents a CaSE$^{ip}$ process, with the agent state being a map from CAction to a Dynamic value. With these defined, I now show a cross-section of the associated semantic functions.

```
skipSem :: Monad m ⇒ CProcess m
skipSem = fixP varX (g?.((r!.e?.tk!.vx, inputN e) |> s?.vx))

haltSem :: Monad m ⇒ CProcess m
haltSem = fixP varX (g?.s?.vx)

choSem :: Monad m ⇒ CProcess m → CProcess m → CProcess m
choSem p q =
 resLR
 (rensL p 'PAR'
  rensR q 'PAR'
  (fixP varX (g?.gi!.gj!.(ri?.sj!.((r!.e?.ei!.tki?.tk!.vx, inputN e) |> (s?.si!.vx))
                         +rj?.si!.((r!.e?.ej!.tkj?.tk!.vx, inputN e) |> (s?.sj!.vx))
                         +s?.si!.sj!.vx))))
```

Above I demonstrate the semantics for skip $\epsilon$, halt $\delta$ and choice $\oplus$. Notice that the choice semantics takes two processes, representing the two options, and produces a third representing the choice between the two. To maintain readability of the Cashew-A semantics I have defined several infix operators to make the processes textually similar to the formal semantics in Chapter 7. The prefix operators are defined in the following class:

```
class CCSSyntax a p where
  (?.) :: a → p → p
  (!.) :: a → p → p
```

I have also defined several functions to achieve the same effects as in the formal counterpart. The functions `rensL` and `rensR` rename the channels of the internal processes, in a similar way to $\mathfrak{F}$ and $\mathfrak{G}$. Similarly, `resLR` restricts all these internal channels, like the set $\mathfrak{R}$. The key protocol channels are all defined as constants in the semantics Haskell module and given a similar name to the formal counterpart, for instance $\mathbf{g^i}, \mathbf{r^i}$ and $\mathbf{e^i}$ become `gi`, `ri` and `ei` (= `Execute I`), respectively. The disabling operator is represented as the binary function `|>`.

### 8.4.3 Examples

In this Section I provide a collection of workflows and their respective transition graphs which exemplify my implementation. The simplest workflow which can be generated has no preconditions, postcondition or dataflow, and has skip as the control flow. Clearly, this is a vacuous workflow, but it nevertheless represents the overall semantics of a workflow. The data-type and transition graph are given below (note: the start state for this and following graphs is always state 0):

```
wfSkip = Wf (read "wfSkip") ATrue (Cf $ Skip) [] OTrue
```

First a word about notation. Because Haskell does not directly support the same syntax as the operational semantics I have had to make some changes. The workflow clocks $\circledcirc_n^w$ are represented by the string `ss-w-n`. The yield clock $\odot^w$ is represented as `yy-w`. The channels mainly adopt the correct names, although the usual $i$ and $j$ subscripts are printed alongside the channel name. The graphs also have `T` to represent $\tau$ and annotate silent actions with the channel or clock name which lead to it (for readability).

The first transition is the receipt of $\mathbf{g}$ from the environment to enable the workflow. This is followed by the ticking of the first workflow clock, $\circledcirc_0^w$. Then the acceptor immediately issues an $\mathbf{r}$ signal to the governor, indicating the preconditions are met (there is only an empty precondition). The next workflow clock then ticks, followed by the governor enabling the control-flow process via $\mathbf{g^i}$. This immediately responds with a ready

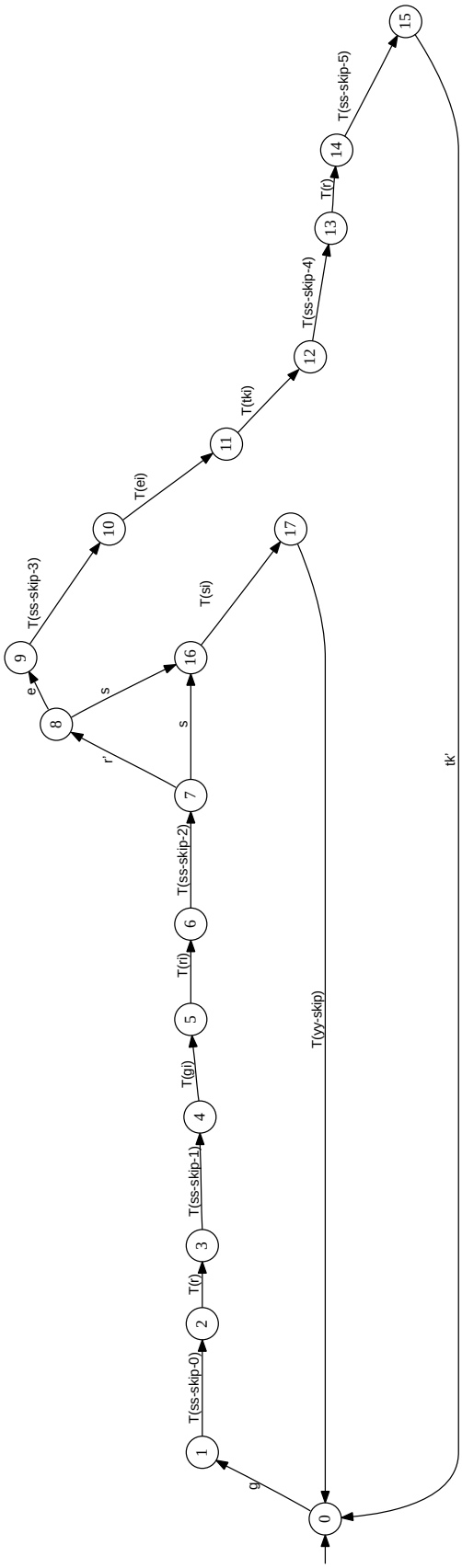



Figure 8.4: A simple “skip” workflow

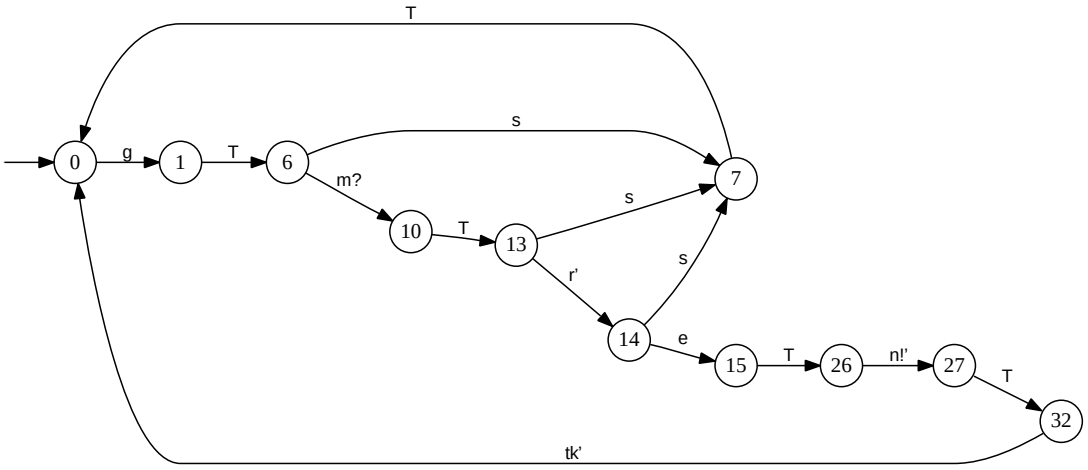



Figure 8.5: The echo workflow

signal on $\mathbf{r^i}$ and then the third workflow clock ticks. In the next two states it is possible for the workflow to be cancelled with the **s** channel, which will cause the top-level control flow to be stopped, and the workflow to return to its initial state. Otherwise, the workflow sends a readiness signal on **r**, receives permission to execute on **e** and then the next workflow clock ticks, indicating the workflow is executing. This is followed by the control flow being instructed to execute, which also immediately finishes (it is skip). The penultimate workflow clock ticks, the offeror indicates postcondition satisfaction via **r** (again it is empty) and the final workflow clock ticks to return the process to its initial state.

The next workflow is a little more interesting. It receives a message $m$, extracts a single part $a$, and then copies this into another message called $n$ which is sent. Again, this workflow has no preconditions or postconditions, but does have a single dataflow connection between the two performances.

```
wfEcho = Wf (read "wfEcho") ATrue (Cf $ pfRec "m" ["a"] `Seq` pfSend "n" ["b"])
                                  [adpp ("recM", "a") ("sendN", "b")]
                                  OTrue
```

For the sake of brevity, I have minimised this graph so that only external communications are shown, and as a result the $\tau$s have no parameters.

For a more a concrete a example, I refer back to the *Calculator* example, which was first shown in Chapter 4 and then again in Chapter 5. I reproduce the UML Activity Diagram in Figure 8.6 for the sake of comparison. I have implemented this in in Cashew-A using a total of four workflows:

- wfCalculator, the outer workflow which contains the main loop and handles receipt and sending of the initial and final messages, respectively;

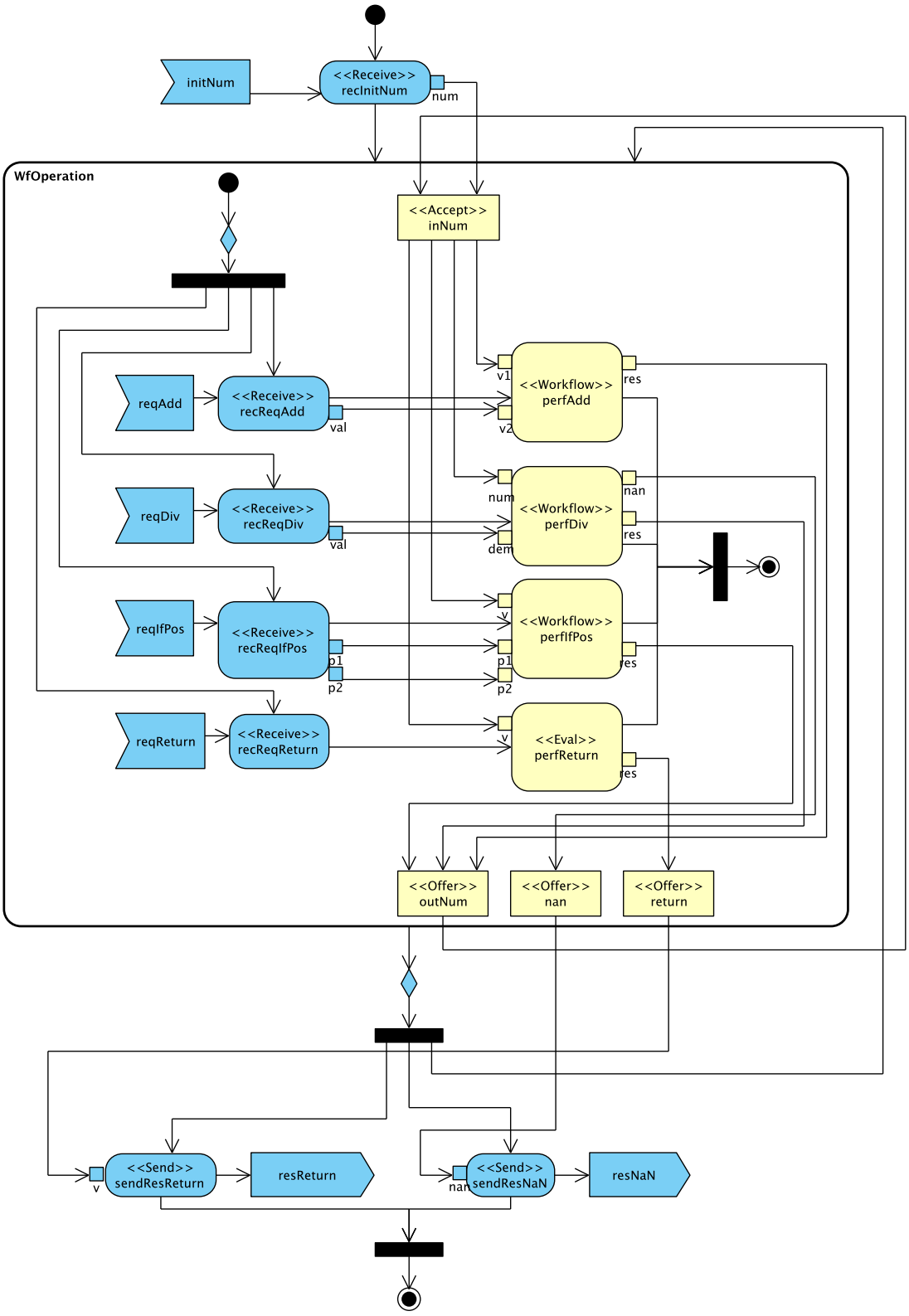



Figure 8.6: A simple Calculator orchestration as a UML 2 Activity Diagram

- wfOperation, the inner workflow which receives one of four messages and performs the necessary operation;
- wfDiv, which provides an implementation of integer division, and handles division by zero;
- wfIfPos, which implements the "if positive" performance.

I will provide definition and transition diagrams for all but the last (it is very similar to wfDiv, starting from the simplest. The most simple of the three is wfIfDiv, which inputs two numbers num and den, and attempts an integer division. If the given denominator is zero, the output nan is returned. Otherwise the result is returned in outpu t res. The Haskell source for this workflow is shown below:

```
wfDiv :: Workflow
wfDiv = Wf (read "wfDiv")
         (aelem "num" `AAnd` aelem "den")
         (Cf $ (Perf pfIsZero `Seq` Perf pfNaN) `Cho`
               (Perf pfNotZero `Seq` Perf pfUDiv))
         [dwp "num" ("pfIsZero", "num")
         ,dwp "num" ("pfNotZero", "num")
         ,dwp "num" ("pfUDiv", "v1")
         ,dwp "num" ("pfUDiv", "v2")
         ,dpw ("pfNaN", "nan") "nan"
         ,dpw ("pfUDiv", "res") "res"
         ]
         (oelem "nan" `OOr` oelem "res")
```

The control-flow begins by making a choice between whether the input den zero or non-zero. It makes this decision this via the two performances, pfIsZero and pfNotZero, which are evaluations with mututally exclusive preconditions. The input den is fed into both and whichever of the two declares readiness enables its respective choice branch. If the denominator is indeed zero then the performance pfNaN is executed, which simply provides a nullary output nan which is output from the workflow. If the number is non-zero then the actual integer division is performed within performance pfUDiv, which is also an expression evaluation. Depending on which branch of the choice executes either nan or res will be output.

The workflow semantics gives the labelled transition system displayed in Figure 8.7. The workflow is first activated and then must receive each of the two inputs (all the while being stopped is a possibility). At this point the workflow makes an internal choice representing the evaluation of one or the other zero test performances. On either branch the workflow declares readiness, is passed permision to execute and then either outputs a

nan or res. Notice state 51, which all the stop signals converge on. Also notice that unless the previous workflows, this workflow does not display all the states. In fact this particular workflow has over 200 states, but minimisation brings this down to the 17 states displayed. Nevertheless, I have retained the label names to make this fact conspicuous.

The next workflow I consider in the calculator example is the main operational workflow: wfOperation which actually performs an operation on the input. As can be seen from Figure 8.6 it is a single step workflow, receiving a message, performing an operation on the supplied data and then returning it through one of the three outputs. It inputs a single number in inNum which represents the accumulator (the workflow represents a push-button calculator), and outputs either an intermediate output in outNum, a final result in res or nan if one of the operations failed (e.g. division-by-zero). The Haskell representation of this workflow is shown below:

```
wfOpCf :: CPat
wfOpCf = (pfRec "reqAdd" ["val"] `Seq` Perf pfAdd) `Cho`
         (pfRec "reqDiv" ["val"] `Seq` Perf pfDiv) `Cho`
         (pfRec "reqIfPos" ["p1", "p2"] `Seq` Perf pfIfPos) `Cho`
         (pfRec "reqReturn" [] `Seq` (Perf pfReturn))
wfOperation :: Workflow
wfOperation =
  Wf (read "wfOperation")
  (aelem "inNum")
  (Cf $ wfOpCf)
  [dwp "inNum" ("pfAdd", "v1")
  ,dwp "inNum" ("pfDiv", "num")
  ,dwp "inNum" ("pfIfPos", "v")
  ,dwp "inNum" ("pfReturn", "v")
  ,sdpp ("recReqAdd", "val") ("pfAdd", "v2")
  ,sdpp ("recReqDiv", "val") ("pfDiv", "den")
  ,sdpp ("recReqIfPos", "p1") ("pfIfPos", "p1")
  ,sdpp ("recReqIfPos", "p2") ("pfIfPos", "p2")
  ,dpw ("pfAdd", "res") "outNum"
  ,dpw ("pfDiv", "res") "outNum"
  ,dpw ("pfDiv", "nan") "nan"
  ,dpw ("pfIfPos", "res") "outNum"
  ,dpw ("pfReturn", "res") "return"
  ]
  (oelem "outNum" `OOr` oelem "nan" `OOr` oelem "return")
```

This is essentially a reproduction of the workflow shown in Chapter 5, Section 5.7. Note that pfDiv and pfIfPos are their two respective workflows wrapped into perfor-

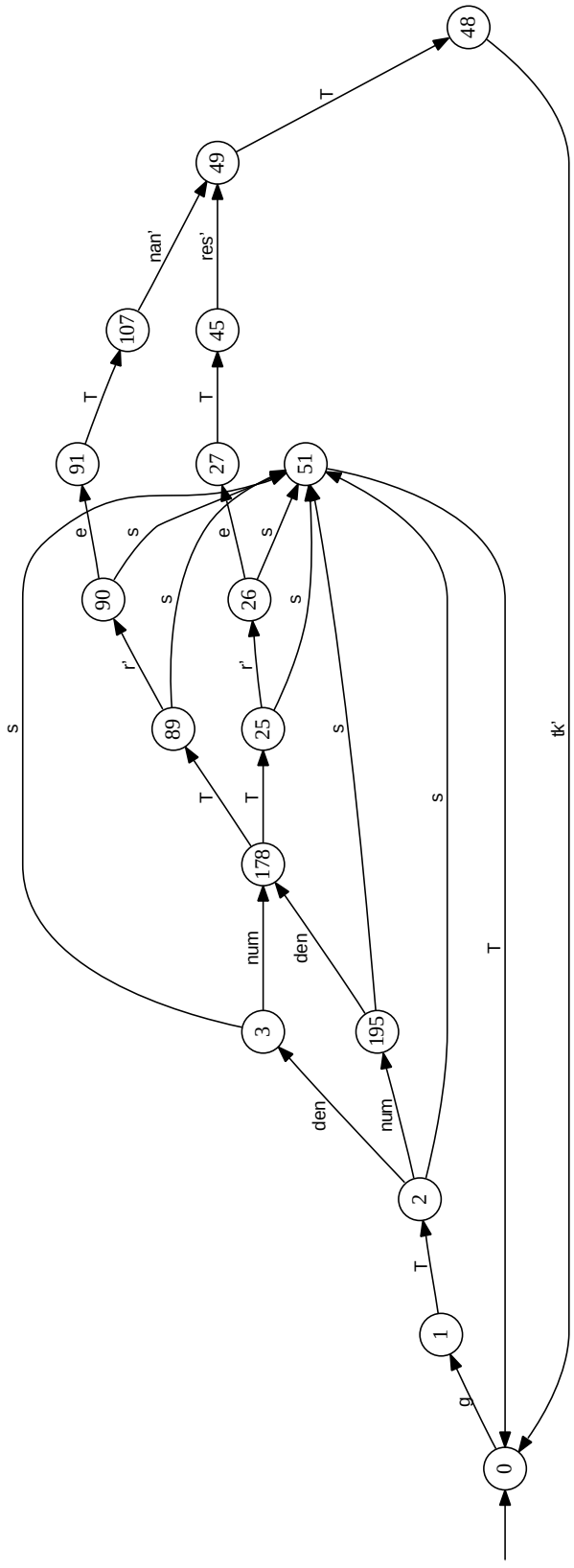



Figure 8.7: The integer division workflow wfDiv

mances. Performance pfAdd is simply an expression evaluation, and pfReturn echo the current value to the workflow output res.

This workflow gives the labelled transition system in Figure 8.8. It becomes activated in the usual way and then waits for one of the four input messages. Upon receiving one, it declares readiness and then acts appropriately. If the message received is reqReturn then workflow simply outputs the current value on return. If reqAdd or reqIfPos then the workflow performs the operation on the current value and then outputs the intermediate value on outNum. If the message is reqDiv then either a outNum or nan could result.

To complete the discussion of the calculator example, we come the main outer workflow, wfCalculator. This workflow takes wfOperation and runs it in a loop, exiting when a final result is output.

```
wfCalc :: Workflow
wfCalc =
  Wf (read "wfCalc")
  ATrue
  (Cf $ (pfRec "initNum" ["num"] `Seq`
        (Iter ((Perf $ Pf (read "pfWfOperation") $ PfWf wfOperation) `Seq` Yield)
         (Cho
          (pfSend "resReturn" ["v"])
          (pfSend "resNaN" ["nan"])
         ))))
  [sdpp ("recInitNum", "num") ("pfWfOperation", "inNum")
  , adpp ("pfWfOperation", "outNum") ("pfWfOperation", "inNum")
  , adpp ("pfWfOperation", "return") ("sendResReturn", "v")
  , adpp ("pfWfOperation", "nan") ("sendResNaN", "nan") ]
  OTrue
```

The workflow has no (internal) preconditions and no (internal) postconditions – this should be true of any workflow representing the top-level of a Web service. The initial value is received in message initNum which also starts of the calculator main loop. This input is passed on to the wfOperation workflow which executes on operation on it. An output is then received which determines what happens next. If a res or nan is output then the exit condition for the loop is satisfied, and hence an appropriate message will be sent and the Web service terminates (or rather returns to the initial state). Otherwise the loop is executed again, this time with the new input from the previous iteration. Notice that the dataflow connection from the output of the operation workflow to its input is *asynchronous*. This is important to ensure the value is not destroyed when the yield occurs. In contrast, the initial value received by recInitNum should be destroyed after it has been used once, and hence this connection is *synchronous*. This workflow gives the transition system found in Figure 8.9.

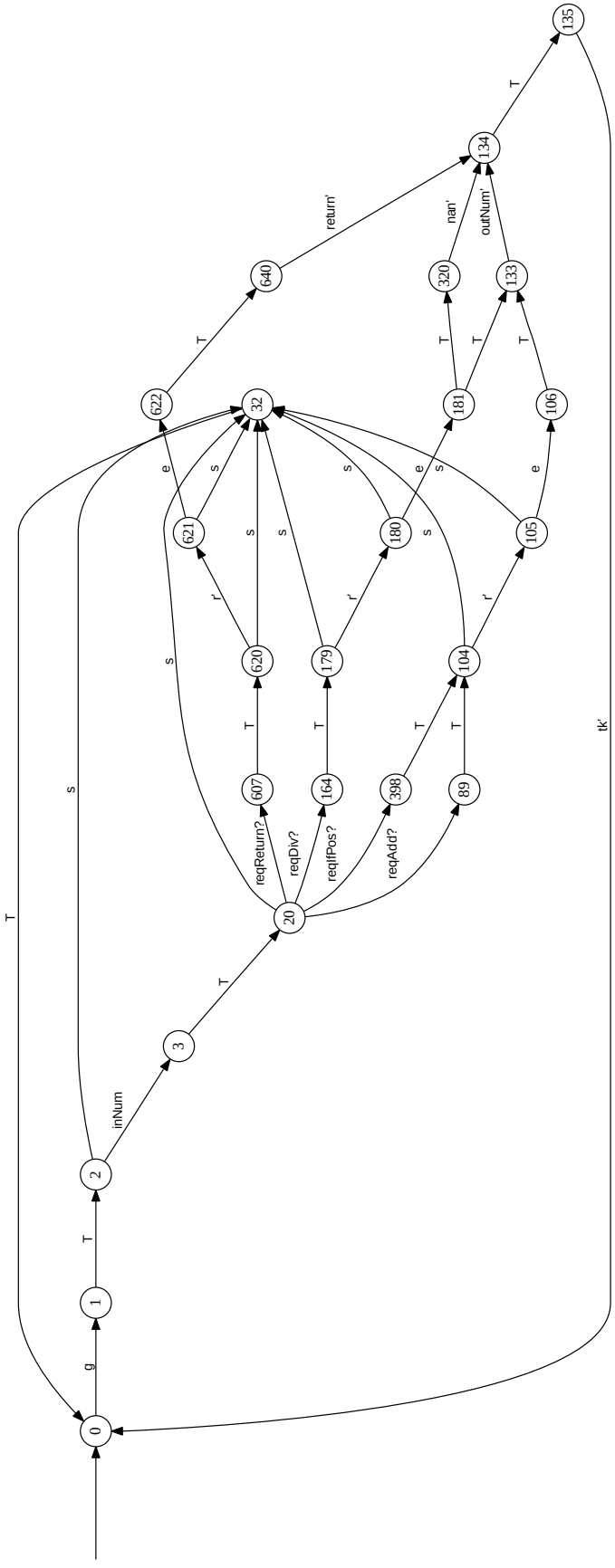



Figure 8.8: The calculator operation workflow wfOperation

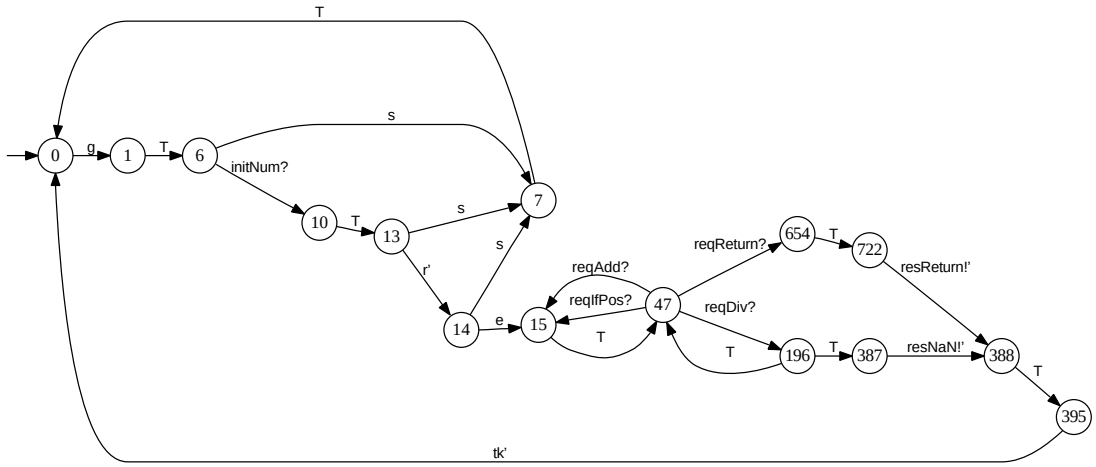



Figure 8.9: Complete Calculator Example with scheduling

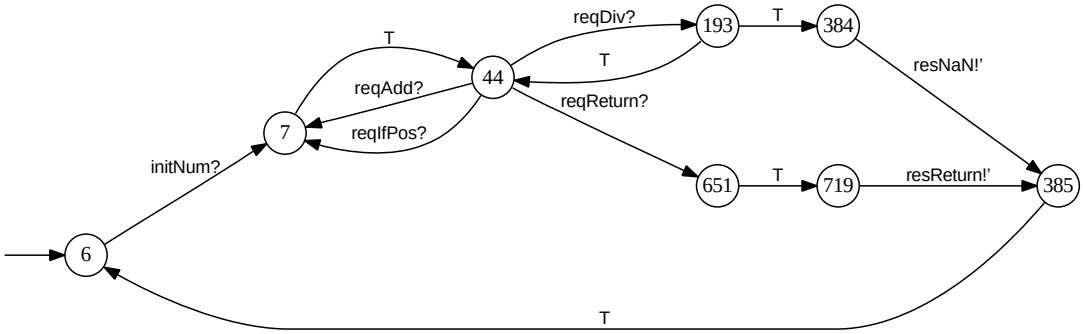



Figure 8.10: Calculator Interface

For completeness, and since this is the top-level workflow, I also provide the interface automaton in Figure 8.10. The latter transition system hides away all the scheduling channels since they are no longer needed by composing it with a *coordination* process, the equivalent of a basic scheduler. This interface could readily be used to advertise the behaviour of the calculator Web service. Therefore, this implementation certainly shows that Cashew-A is a viable model for service composition.

## 8.5 Conclusion

In this Chapter I have examined in detail the question of how CaSE$^{ip}$, and CCS-style process calculi in general, should be implemented. I have shown that Haskell provides an excellent framework for specifying a process calculus's syntax and operational semantics. I have also shown that such a semantics, when implemented and combined with the power of *Monads*, can be used to drive real-world interactions.

Furthermore, I looked at the whole issue of process verification. Firstly, I showed that *Type Families* and *Generalised Algebraic Datatypes* provide the necessary foundations for

typing CCS process interactions. Although support is still in its infancy, this is clearly an area of interest in the Functional Programming Community, even more so in *Dependently Typed* languages, such as *Epigram* (McBride and McKinna, 2004) and *Agda* (Norell, 2007). In the future I would expect that process monadic bindings could be fully typed, and guaranteed safe, prior to execution.

Secondly I provided a foundation for Finite State based verification of processes. I provided an algorithm for graph generation, and showed how this could be used to implement *minimisation* and *bisimulation* via the Partition Refinement algorithm. This work converges on *ConCalc*, my command line verification environment, which has been invaluable in experimenting with the Cashew-A semantics.

Finally, I showed the beginnings of a framework for representing the semantic mapping from Cashew-A into CaSE^ip in Haskell. Although incomplete it shows that such a mapping can be given using a syntax very close to the original formal semantics. In the future, I would like to see this implementation completed and used to write a Web service composition and verification server, which would support a graphical orchestration design tool, and provide verification and testing features in real-time. In order to do this though a better minimisation algorithm will be needed which can properly differentiate internal choice from parallel composition of silent actions.

What is also missing from the implementation of Cashew-A is the ability to execute a workflow. The framework exists as a I demonstrated in Sections 8.2.1, 8.2.3 and 8.2.6, but due to time constraints I have not as yet been able to tie this to the process syntax. Nevertheless, having tested out the bindings system, given further time this is certainly a viable execution method.

# Chapter 9

# Conclusions and Future Work

*In this final chapter I will provide a summary of all the work completed in this Thesis and draw together some conclusions. There is still much work that can be done in this area, and thus I outline my ideas for potential future work.*

## 9.1 Summary

THE MAIN ACHIEVEMENT of this Thesis is a new paradigm for rapid development of Composite Web services. Through the compositional semantic framework I have developed a partial implementation, and it is now possible to build a system for diagrammatically evolving a Web service, with verification facilities available in real-time. Furthermore, a future *Abstract State Machines* semantics would add the possibility of execution and deployment, whilst retaining a model which can be incrementally built without complete recompilation.

The work is unusual in some respects, because it focuses on such a wide variety of subjects. This is somewhat inevitable as the service-oriented architecture is itself a multi-disciplinary field. I have touched on subjects as diverse as *Business Process Modelling*, *Functional Programming* and *Process Algebra*. The work in the Thesis is primarily under three headings:

- Abstract Timed Process Calculus;
- Web service orchestration and choreography semantics;
- Functional implementation of the above.

Nevertheless the main aim of this Thesis has remained the same throughout, and that is to provide a *A Compositional Semantic Theory for Service Composition*. Specifically, my intention has been to provide a formal semantic framework for describing a wide variety of service-oriented features in a compositional manner, and I believe this has been largely

achieved. A cornerstone of this theory is my abstract timed process calculus, but before I could arrive at this goal, a number of intermediate steps were needed.

The first step was the development of a novel language called Cashew-A for describing orchestrations. This language draws on ideas from several preexisting Web service languages and process algebras, primarily, the original language Cashew-S (Norton et al., 2005), which was itself based on OWL-S (Martin et al., 2004). This language provided the basis for a dataflow based orchestration language, where precondition satisfaction is integral to the language rather than an abstract and hidden afterthought. I then took the dataflow model and expanded it to a full Boolean algebra, which allows more complicated pre-conditions and post-conditions in line with WSMO Goals.

The control flow aspect is a *CSP* style language, with the usual process algebra operators, but in addition several dataflow oriented constructs. The interleaving construct allows the execution order of a bag of processes to be determined by their preconditions. The *yield* construct forces a workflow to proceed maximally before allowing the process it guards to execute, thus forcing all possible dataflow propagation. In a similar vein, the language includes a wait operator, which forces the workflow to expend one time unit (relative to a metric determined by the workflow head) before executing the guarded process. Unlike yield, wait allows time to advance during the execution of a workflow's sub-components (as expected) and thus the wait operator is used to measure real periods of time.

The language was then extended with *compensation* which allows the effects of a workflow orchestration to be mitigated, whenever suitable compensation procedures for each step are provided. This extended language contains a compensation operator which allows a process to be paired with a compensation process. The language also contains an operator to throw an exception and commence the compensation process. Due to a lack of understanding as to how loops should interact with compensations, this fragment contains no loop operator, as it would seemingly make the system non-finite state.

I then turned my attention to giving Cashew-A an operational semantics. In Chapter 6 I explored some of the extensions which are needed to the base calculus CaSEin order to permit description of service composition with an associated choreography. The eventual result of this study was CaSE$^{ip}$ , an extension of CaSE, which includes a more liberal approach to patience which in turn allows, amongst other features, more timed choice possibilities. I provided an update of CaSE's *Temporal Observation Congruence* equivalence theory called and briefly explored *Alternating Simulation* as a method for describing choreography conformance.

I then used CaSE$^{ip}$ to given an operational semantics to Cashew-A. The semantic framework is designed in such a way that it is easily extensible, motivated by the fluid nature of Web service languages. It provides a sophisticated scheduling system where abstract time is central to deciding execution order. Dataflow also plays a key role, providing a graph structured layer on top of the traditional block-structured programming

constructs. The Operational Semantics was extended with a compensation mechanism, also driven by abstract time. As a result it is capable of representing various compensation strategies, though the compensation mechanism itself was limited to compensations which cannot themselves fail.

I then took this one stage further and described an implementation of CaSE$^{ip}$, step-by-step, which shows how processes can be used to orchestrate real world interactions with the aid of *Monads*. I further provided a basic verification framework, including support for graph generation and semantic bisimulation checking. In particular, my plugin-oriented process experimentation environment *ConCalc* provides verification features and is easily extensible. Finally I demonstrated a basic Cashew-A implementation in Haskell, complete with a working example, which could provide the foundation for a full service composition and verification server.

Having given a brief résumé of the work in this Thesis, I will now describe some possible areas of future exploration.

## 9.2 Areas of Further Exploration

### 9.2.1 Negated Preconditions and Transient Inputs

A Cashew-A workflow allows the specification of input preconditions using a simple Boolean algebra $(\mathcal{A}, \sqcup, \sqcap, \mathbf{0}, \mathbf{1})$. When a workflow is executed a component with a non-empty precondition begins in an unready state. Its inputs are then gradually populated by the execution of other components connected by dataflow, eventually allowing it to execute. An interesting question is what further behaviour could be modelled if in addition negative preconditions were allowed? With this system a component could become ready, but if one of the negative preconditions is satisfied, it would later become unready again.

In my analysis of the workflow patterns in Chapter 5, Section 5.8 I described the *Critical Region* pattern, in which two mutually exclusive regions in a workflow exist. This cannot be modelled in Cashew-A as it stands, because it is only possible for one component to *enable* another component via dataflow. With negative preconditions a component could also *disable* a component temporarily, and re-enable it when (for example) one critical region completes.

The reason this was not included in this work is because of uncertainty as to how priority should be modelled in this context. If a dataflow element becomes available which could disable a workflow, clearly it should be communicated immediately, or else the workflow could start. Clocks don't provide an obvious way of modelling this, however. I could use yields, but this would lead to very inefficient workflows. A more likely solution is the use of the RTC, and linking this to the readiness condition in a similar way to the wait $\circ P$ operator.

### 9.2.2 Value-added CaSE$^{ip}$

Value-added CaSE$^{ip}$ is an idea which was partially developed during this Thesis, but not sufficiently developed so as to include it. The idea is that when a WSMO Goal completes execution, aside from preconditions it also specifies *effects* on the world. These effects are specified logically, and are used by WSMO to do bespoke service composition. Currently my semantic model does not account for them in any way, although clearly when a number of Goals are composed an overall effect will be achieved.

My idea is to extend CaSE$^{ip}$'s action labels with a subscript denoting the effect or cost this particular action has. Then when a process graph representing an orchestration is minimised the overall effect could be computed and added the Goal description.

### 9.2.3 Protocol Mediation

The Cashew model presented thus far lacks an adequate form of mediating a service choreography onto a template client choreography. A coordination currently directly composes a Web service's choreography with choreography of the enclosed workflow, ensuring that a correspondence exists (via alternating simulation). This only works in the limited cases where the protocol of a Web service can be directly predicted beforehand. In many cases a more complex mapping will be required, such a mapping is known as a *protocol mediation*. Specifically, a workflow defines the order it needs to receive data from its partner Web service and this template protocol must be mapped onto an actual Web service protocol.

A protocol mediation is a four-mode transition system, specifically with input and output modifiers for each partner. Such a transition system is composed between the Web service choreography and the client choreography, and must correspond to both IO automata. It is unlikely that such a mediation can be derived directly given target choreographies, since such a process will undoubtedly be intractable. Thus rather than direct derivation, I propose a component library approach to protocol mediation. A protocol mediation can be represented as a Cashew-A workflow, with only workflow and messaging performances. There would be four messaging performances: **ReceiveClient**, **ReceiveServer**, **SendClient** and **SendServer**. It seems likely that there are protocol mediation *patterns*, such that each protocol mediator can be modularised. Preliminary work on *service interaction patterns* has already been done by Barros et al. (2005), who look at the different ways in which a Web service can communicate.

These protocol mediation patterns would describe how part of a protocol can be mediated to another protocol part. They would then be composed to form a complete mediation for the service protocol.

### 9.2.4 Enhanced Compensation Mechanism

The compensation mechanism presented in Chapters 5 and 7 is little more than a proof-of-concept. Whilst it shows that a compensation mechanism can be encoded in CaSE$^{ip}$, it does not provide many of the more advanced compensation features found in languages like WS-BPEL (Jordan and Evdemon, 2007) and StAC (Chessell et al., 2002). A particular missing feature is that a compensable transaction may only be contained in a single workflow. It ought to be possible that several workflows can be linked together in a single transaction. Although it is possible to propagate failure upwards to the parent workflow via outputs, it isn't possible to propagate a failure in a workflow to a child. To do this a change in the protocol is needed. This already somewhat exists in the form of the `x` channel which is used for allowing the speculative parallel operator to cancel unneeded threads. Currently, however, it only allows a workflow currently in progress to be cancelled, not one that is complete. Clearly though this is possible in CaSE$^{ip}$.

Another feature which the compensation mechanism does not provide is a *try-catch* mechanism, which would allow certain exceptions to be handled internally without the need for compensation. Currently if an exception is raised the entire workflow unconditionally aborts. Instead it should be possible to have an exception handler which has the option of either performing some corrective action, or passing the exception upward. In addition a similar mechanism is needed to allow compensations which can fail. In this case it should also be possible that a compensation action can throw an exception which is caught by the workflow parent.

In addition investigation is needed into how loops should be compensated, if indeed this is even possible in a finite state context. So to summarise the compensation mechanism, whilst certainly an adequate start, needs further investigation.

### 9.2.5 Typed CaSE$^{ip}$ implementation

In Chapter 8 Section 8.2.4 I studied how typed process bindings may be described using the most recent features added to the GHC Haskell implementation, such as *Generalised Algebraic Data-Types* (Peyton-Jones et al., 2006) and *Type Families* (Schrijvers et al., 2008). Although my investigation proved unfruitful in the short term, it nevertheless shows that a future Haskell will likely allow such an implementation. Indeed since this work began a new version of GHC has been released adding many new features, so clearly Haskell development is moving forward rapidly. It also seems likely that with so many novel features appearing a new revision of the Haskell standard will be necessary[1]. With respect to my work, it seems the most important future extension will be an overhaul of the *kind* system. Currently Haskell only has kinds for type application, e.g. $* \rightarrow *$, but kinds in the future will be customisable in a similar way to basic data-types, and thus able to restrict the scope of type families. This will in turn improve decidability of type-

[1]Work on such a new standard can be seen at `http://hackage.haskell.org/trac/haskell-prime/`

families and therefore flexibility, since their definitions will be closed over possible kinds. For instance it may be possible to capture a proper type-list construct, something which would be of great benefit. Such an extension is still a way off, though there is progress[2].

### 9.2.6 Complete Web service composition engine

Thus far the implementation of Cashew-A described in Chapter 8 is far from complete. Although I have provided a suitable basis using my Haskell CaSE$^{ip}$ implementation, as yet the semantics of Cashew-A are incomplete. I propose a representation of Cashew-A where each workflow caches a minimised representation of the workflow's semantics. A query language must also be developed which enables the frontend to manipulate the orchestration representation, such as adding a new performance or changing how they are composed. Whenever a change is made the appropriate cached LTSs should be updated and necessary verifications applied when necessary.

The minimisation algorithm will also need some work. For instance the existing algorithm does not work well in concert with internal choice. The main reason for this is interleaving semantics which means, that at a graph level, parallel actions and alternative actions look similar. Whilst the weak transition relation turns all silent actions into appropriate non-determinism at the Web service interface level it is necessary to distinguish internal choice from external choice. Furthermore an updated definition of *alternating simulation* (Alur et al., 1998) from Section 6.8 will be needed to allow checking conformance of CaSE$^{ip}$ choreographies. Indeed, such an extension will likely motivate a whole study in and of itself into the role of clocks (if any) in describing choreographies.

## 9.3 Outro

Although the overall theoretical contribution of this work is necessarily restricted, I argue that it is important in its context. In contrast to previous work on CaSE (Norton et al., 2003; Norton and Fairtlough, 2004; Norton, 2005*a*), I have made a relatively shallow exploration of the new calculus, and have instead gone on to explore a wide variety of subjects. I believe that the work in parallel areas is thus useful because it links theory with practice.

In my opinion, the greatest single achievement of this Thesis is the Operational Semantics given in Chapter 7. Here we have an extensible framework which is highly modular in nature and governed by standard protocols. What I have effectively shown in the semantics is that *Timed Process Calculus* is indeed an ideal theoretical paradigm for service composition. It is flexible and capable of providing both standard features of programming languages, along with a sophisticated dataflow model, message flow model, and compensation. I believe the standardised nature of this model could potentially pay

[2]See the tracker at `http://hackage.haskell.org/trac/ghc/wiki/KindSystem`

dividends should the ASM model be extended. It seems clear that it provides an appropriate method of inspecting a components state, which links nicely into the ontology structure. Much of my time in this work has been spent experimenting with the protocol, trying to discover the minimal number of stages needed. I believe I have perfected this, and along with the compensation protocol an excellent foundation is set down for future work.

Though this is the greatest achievement, it is by no means the only achievement. I am also very satisfied with the new libraries and programs I have implemented in Haskell during this time. Even if CaSE$^{ip}$ does not turn out to be useful to future researchers, the extensible nature of ConCalc and the LTS library mean that they are useful in any context. Indeed, I would say that the development of the Haskell tools has provided the greatest personal enjoyment. Together with the semantic framework, the pieces are now available to build a service composition server in Haskell. Indeed, it is my hope that in the future, the service composition engine I have herein envisioned will be realised, opening new possibilities for Web services, and advancing the *Semantic Web* vision.

# Appendix A

# Rejected Process Calculi

*In this Appendix I consider two process calculi which where developed and rejected as part of this Thesis work. Although not directly useful, they do illustrate two alternative pathways I attempted. The two calculi are called* ***Interruptible CCS*** *(ICCS), which focuses on localised interruption, and* ***CaSE$^{mt}$*** *, which focuses on generalisation of* ***CaSE****. The latter calculus is particularly informative in that it illustrates how* ***CaSE*** *became my final calculus,* ***CaSE$^{ip}$****.*

## A.1 Interruptible CCS

Interruptible CCS (ICCS) follows the example of Lüttgen (1998) by integrating higher priority actions into CCS which will act as interrupts. Lüttgen previously created a calculus called CCS$^{ch}$ i.e. CCS with Cleaveland and Hennessy style prioritisation (Cleaveland and Hennessy, 1990). In this calculus each action has an associated natural number index denoting its priority, e.g. $a:j$ and $\overline{b}:k$, where $j,k \in \mathbb{N}$. The higher the number of the index the lower the priority, with $0$ being the highest. When two actions synchronise, a prioritised silent action $\tau_k$ is produced which will prevent parallel lower priority actions from occurring. The pre-emptive power of such an action in this calculus is global. Therefore, if any process produces an action $a:0$, every other action of lower priority is pre-empted away.

However, since I desire a component setting, I distinguish this ICCS from Lüttgen's calculus in that it will localise pre-emption to a particular area of the process topology. The new calculus extends CCS with action preemption, in a style similar to that described above. Each action label has two varieties, a *prioritised* action written as $\underline{a}$, and an *unprioritised* action written as $a$. Whilst unprioritised actions behave exactly like regular actions in CCS, prioritised actions have a different style of synchronisation. When two complementary prioritised actions $\underline{a}$ and $\overline{\underline{a}}$ synchronise, a special prioritised synchronisation action $\langle a \rangle$ is created rather than a $\tau$. This prioritised action then pre-empts all unprioritised actions (including, by extension, clock ticks) within its boundary, which is defined using a hiding operator. A hidden prioritised action is observed as a regular $\tau$,

$$\begin{array}{lllllllllll}
\Lambda & = & \{a, b, c \cdots\} & \overline{\Lambda} & = & \{\overline{a} \mid a \in \Lambda\} & \mathscr{A} & = & \Lambda \cup \overline{\Lambda} \cup \{\tau\} \\
\underline{\Lambda} & = & \{\underline{a} \mid a \in \Lambda\} & \underline{\overline{\Lambda}} & = & \{\underline{\overline{a}} \mid a \in \Lambda\} & \underline{\mathscr{A}} & = & \underline{\Lambda} \cup \underline{\overline{\Lambda}} \cup \{\langle a \rangle | a \in \Lambda\} \\
\alpha & \in & \mathscr{A} & & & \underline{\alpha} \in \underline{\mathscr{A}} & & \lambda & \in & \mathscr{A} \cup \underline{\mathscr{A}}
\end{array}$$

$$\mathcal{E} \quad ::= \quad \mathbf{0} \mid \alpha.\mathcal{E} \mid \underline{\alpha}.\mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E}|\mathcal{E} \mid \mathcal{E}\{\!|a \mapsto b|\!\} \mid \mathcal{E} \setminus a \mid \mathcal{E}/a \mid \mu X.\mathcal{E} \mid X$$

Table A.1: ICCS Core Syntax

with its pre-emptive power removed.

For instance the process $\langle a \rangle.E + b.F$ is behaviourally equivalent to $\langle a \rangle.E$, since the $b$ action is always pre-empted. Furthermore, an interruptible component may be represented like so:

$$(\mu X.(a.b.c.X \rhd_c \underline{x}.0) \mid e.\underline{\overline{x}}.0) \setminus x/x$$

This component repeatedly inputs $a, b$ and $c$. However, if at some point an $e$ input is encountered, the component immediately executes $\langle x \rangle$ and halts. Here $e$ is a command to stop, and $x$ is the interruption channel which does it. Note, that this cannot be done in CCS since $x$ would be simply interleaved with the other actions. Notice that $x$ must be both restricted and hidden, the restriction removes the input and output actions and the hiding converts $\langle x \rangle$ to a $\tau$. Therefore any other process outside the hiding scope of $x$ will not be pre-empted. The operator $\rhd_c$ is called the *disabling operator*, and it allows the process on the right to interrupt the process on the left until $c$ occurs.

### A.1.1 Syntax

The symbol definitions and syntax of ICCS are shown in Table A.1. I use the usual $\Lambda$ and $\overline{\Lambda}$ to represent names and conames, with the additional dual sets $\underline{\Lambda}$ and $\underline{\overline{\Lambda}}$ to represent prioritised names and conames. In addition $\mathscr{A}$ represents the set of all unprioritised actions with $\tau$, and $\underline{\mathscr{A}}$ represents the set of prioritised actions, including a prioritised synchronisation action for each name (and no $\tau$). Other than these different sorts, the syntax of CCS is identical[1]. There are only two new operators, prioritised action prefix $\underline{\alpha}.E$ and interruption hiding $E/a$. This syntax does not include the disabling operator directly, as it can be derived by recursively summing the interrupting process with each action prefix.

### A.1.2 Operational Semantics

The operational semantics for ICCS can be found in Table A.2. I use a standard LTS-based operational semantics, that is $(\mathcal{E}, \mathscr{A} \cup \underline{\mathscr{A}}, \rightarrow)$, where $\rightarrow \subseteq \mathcal{E} \times (\mathscr{A} \cup \underline{\mathscr{A}}) \times \mathcal{E}$. Many

[1]Except that in this calculus and throughout the Appendix I am using $\{\!|\!\}$ brackets for renaming. This is because square brackets are used for vectors.

$$\textsf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \underline{\textsf{Act}}\ \frac{}{\underline{\alpha}.E \xrightarrow{\underline{\alpha}} E}$$

$$\textsf{Rel}\ \frac{E \xrightarrow{\lambda} E'}{E\{\!|f|\!\} \xrightarrow{f(\lambda)} E'\{\!|f|\!\}} \qquad \textsf{Res}\ \frac{E \xrightarrow{\lambda} E'}{E \setminus a \xrightarrow{\lambda} E' \setminus a}\ (1)$$

$$\textsf{Hid}\ \frac{E \xrightarrow{\lambda} E'}{E/a \xrightarrow{\lambda} E'/a}\ (2) \qquad \underline{\textsf{Hid}}\ \frac{E \xrightarrow{\langle a \rangle} E'}{E/a \xrightarrow{\tau} E'/a}\ (4)$$

$$\textsf{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}\ (3) \qquad \underline{\textsf{Sum1}}\ \frac{E \xrightarrow{\underline{\alpha}} E'}{E + F \xrightarrow{\underline{\alpha}} E'}$$

$$\textsf{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}\ (4) \qquad \underline{\textsf{Sum2}}\ \frac{F \xrightarrow{\underline{\alpha}} F'}{E + F \xrightarrow{\underline{\alpha}} F'}$$

$$\textsf{Com1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}\ (5) \qquad \underline{\textsf{Com1}}\ \frac{E \xrightarrow{\underline{\alpha}} E'}{E \mid F \xrightarrow{\underline{\alpha}} E' \mid F}$$

$$\textsf{Com2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}\ (5) \qquad \underline{\textsf{Com2}}\ \frac{F \xrightarrow{\underline{\alpha}} F'}{E \mid F \xrightarrow{\underline{\alpha}} E \mid F'}$$

$$\textsf{Com3}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}\ (5) \qquad \underline{\textsf{Com3}}\ \frac{E \xrightarrow{\underline{a}} E', F \xrightarrow{\overline{\underline{a}}} F'}{E \mid F \xrightarrow{\langle a \rangle} E' \mid F'}$$

1) $\lambda \notin \{a, \underline{a}, \overline{a}, \overline{\underline{a}}, \langle a \rangle\}$ 2) $\lambda \notin \{\underline{a}, \overline{\underline{a}}\}$ 3) $F \xrightarrow{\langle a \rangle}\!\!\!\!\!\rightarrow$ 4) $E \xrightarrow{\langle a \rangle}\!\!\!\!\!\rightarrow$ 5) $E|F \xrightarrow{\langle a \rangle}\!\!\!\!\!\rightarrow$

Table A.2: ICCS Operational Semantics

of the rules are similar to their counterparts in CCS, but with the addition of extra side conditions to deal with pre-emption. The rules with underlined names are the new rules which take account of the prioritised actions. These rules are virtually identical to the CCS rules as well, the exception being Com3 which produces an interruption action $\langle a \rangle$ instead of a $\tau$. Rules Sum1 and Sum2 require that the opposing side is not capable of performing a prioritised action. The same is true of Com1-Com3, but since the parallel composition may also synchronise to produce an interruption, the composition $E \mid F$ is checked for interruption. Hid hides interruption actions as $\tau$s.

ICCS has a number of interesting features. Like CaSE it has an implicit notion of priority hierarchy, but it works in the opposite direction. When an interrupt is hidden any unhidden interrupts have the ability to pre-empt away the $\tau$ produced. Thus the further up an action is hidden in the process topology, the more pre-emptive power it has. This allows us to achieve a similar effect to the prioritised $\tau$s of CCS$^{\text{ch}}$, where each $\tau$ has an index denoting its priority. Here, $\tau$s also have a priority, but based on their origin rather than intrinsic nature.

### A.1.3 Equivalence Theory

As in CCS, my intention is to build a weak bisimulation based equivalence which can be used for basic component substitutivity. As usual, I begin by considering a straightforward adaptation of weak bisimulation. I use the standard weak transition relation $\Rightarrow$ as given in Definition 2.3.2 (Chapter 2). To reiterate, the weak transition relation abstracts from silent actions by linking states which are accessible by doing a visible action surrounded by any number of silent actions. With it I derive a straightforward derivative of weak bisimulation for the new calculus:

**Definition A.1.1** *Naïve Interruptible Weak Bisimulation*
*A symmetric relation $\mathcal{R}$ is a Naïve Interruptible Weak Bisimulation provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{\lambda} E'$ then $\exists F'.F \stackrel{\hat{\lambda}}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$.*

*We say that $E \approx_n F$ whenever $\exists \mathcal{R}.\langle E, F\rangle \in \mathcal{R}$ and $\mathcal{R}$ is a Naïve Interruptible Weak Bisimulation.*

Whilst $\approx_n$ is an equivalence it is not a congruence, particularly with respect to parallel composition (which is vital for the base equivalence). To see why, consider the processes $E = \langle a\rangle.\langle b\rangle.\mathbf{0}$ and $F = \langle a\rangle.\tau.\langle b\rangle.\mathbf{0}$. Whilst it follows that $E \approx_n F$, if we parallel compose $G = c.0$ then they are distinguishable. This is because $\tau$ has less pre-emptive power than $\langle b\rangle$. Once $F$ has performed its first action $\langle a\rangle$, other parallel unprioritised actions like $G$'s $c$ can also occur. In $E$ this is not possible because there is no gap between the two prioritised actions.

There is a similar effect in the processes $E = a.\underline{b}.\mathbf{0}$ and $F = a.\tau.\underline{b}.\mathbf{0}$, with parallel process $G = \overline{a}.(c.\mathbf{0} + \overline{\underline{b}}.\mathbf{0})$. Whilst $F$ will allow $c$ to occur after the synchronisation on $a$ due to the presence of a $\tau$, $E$ will not. It is thus clear that in order to define a congruence relation, sequences of prioritised actions cannot be separated by silent actions. They behave fundamentally differently to regular actions, and so I modify the definition of weak bisimulation:

**Definition A.1.2** *Interruptible Weak Bisimulation*
*A symmetric relation $\mathcal{R}$ is an Interruptible Weak Bisimulation provided $\forall \langle E, F\rangle \in \mathcal{R}$:*

- *If $E \xrightarrow{\alpha} E'$ then $\exists F'.F \stackrel{\hat{a}}{\Rightarrow} F'$ and $\langle E', F'\rangle \in \mathcal{R}$;*
- *If $E \xrightarrow{\underline{\alpha}} E'$ then $\exists F'.F \xrightarrow{\alpha} F'$ and $\langle E', F'\rangle \in \mathcal{R}$.*

*We say that $E \approxeq F$ whenever $\exists \mathcal{R}.\langle E, F\rangle \in \mathcal{R}$ and $\mathcal{R}$ is an Interruptible Weak Bisimulation.*

It is easy to prove this is a congruence for the static operators – it is effectively weak bisimulation combined with strong bisimulation. I therefore only include the proofs for hiding and parallel composition.

**Theorem A.1.3** ***Compositionality of Interruptible Weak Bisimulation***
*Interruptible Weak Bisimulation is a congruence with respect to parallel composition and hiding.*

**Proof.** We construct a relation $\mathcal{R} = \{\langle C[E], C[F]\rangle \mid E \approxeq F\} \cup \approxeq$, where $C$ is a context (e.g. $[- \mid G]$) and show that it is a weak bisimulation. That is, every element satisfies both conditions of Definition A.1.2. We perform case analysis on each of the following constructs:

1. $C[E] = E \mid G$
2. $C[E] = E/a$

1. There are three possible transitions:
   (a) $E \mid G \xrightarrow{a} H$. Either $E \xrightarrow{a} E'$ with $H \equiv E' \mid G$ by Com1, or $G \xrightarrow{a} G'$ with $H \equiv E \mid G'$ by Com2.
      - If the former then by Com1 it follows that $G \overset{\langle a \rangle}{\nrightarrow}$. Furthermore, since $E \approxeq F$ it follows that $F \overset{a}{\Rightarrow} F'$ . Therefore by Com1 $F \mid G \overset{a}{\Rightarrow} F' \mid G$ and $\langle E' \mid G, F' \mid G\rangle \in \mathcal{R}$ as required.
      - If the latter then by Com2 it follows that $E \overset{\langle b \rangle}{\nrightarrow}$, and since $E \approxeq F$ it follows that $F \overset{\langle b \rangle}{\nrightarrow}$. Therefore $F \mid G \xrightarrow{a} F \mid G'$ and $\langle E \mid G', F \mid G'\rangle \in \mathcal{R}$ as required.
   (b) $E \mid G \xrightarrow{\tau} H$. This follows a very similar proof to the above, combined with the standard CCS proof (Milner, 1989*a*) in the case of a synchronisation.
   (c) $E \mid G \xrightarrow{\alpha} H$. Again, similar to the above items, but using rules Com1 and Com2, and therefore dispensing with the negative side-condition.

2. There are three possible transitions:
   (a) $E/a \xrightarrow{b} E'/a$
      - By Hid it follows that $E \xrightarrow{b} E'$;
      - Since $E \approxeq F$ then $F \overset{b}{\Rightarrow} F'$ with $E' \approxeq F'$;
      - Therefore by Hid it follows that $F/a \overset{b}{\Rightarrow} F'/a$ and $\langle E'/a, F'/a\rangle \in \mathcal{R}$ as required.
   (b) $E/a \xrightarrow{\tau} E'/a$. Either $E \xrightarrow{\tau} E'$ or $E \xrightarrow{\langle a \rangle} E'$.
      - If the former, then the proof follows the previous case exactly.
      - If the latter, then by Hid it follows that $E \xrightarrow{\langle a \rangle} E'$. Therefore $F \xrightarrow{\langle a \rangle} F'$ with $E' \approxeq F'$, and by Hid $F/a \xrightarrow{\tau} F'/a$ with $\langle E'/a, F'/a\rangle \in \mathcal{R}$ as required.
   (c) $E/a \xrightarrow{\alpha} E'/a$.
      - By Hid it follows that $E \xrightarrow{\alpha} E'$. Thus, $F \xrightarrow{\alpha} F'$ with $E' \approxeq F'$, and by Hid $F/a \xrightarrow{\alpha} F'/a$, with $\langle E'/a, F'/a\rangle \in \mathcal{R}$ as required.

The converse follows by symmetry. □

This definition, although theoretically correct and yielding a congruence, is *not* a weak bisimulation relation. Although normal actions are abstracted in the usual way, interruptions do not allow any amount of silence to be inserted, simply because silence allows other actions to occur which interruption prevents. This is very unsatisfactory, and very much conflicts with the abstract time view of silence, which does not distinguish an empty sequence from a silent action (as indeed CCS doesn't). More importantly though, standard algorithms for checking weak bisimulation (such as *partition refinement*) are useless, since in no way can this equivalence be derived from it.

Therefore an alternative must be found. The evidence suggests that for a relation to be a weak bisimulation derivative, it must have a concept of $\tau$ as the highest priority action which can be inserted between two actions without changing the contextual behaviour. This is the situation with CaSE and it is also the situation with Lüttgen's calculus, which had a highest priority $\tau : 0$ which can always been inserted between two actions.

Switching to a localised version of interruption removed this property, and made $\tau$ prey to interruption actions. So a possible way to solve this problem may be to have $\tau$ non-pre-emptable by $\langle a \rangle$ actions. However, this introduces further problems, as it would mean that $\mid$ is no longer associative. To see why consider the process $a.P \mid \overline{a}.Q \mid \langle x \rangle.R$. The two different bracketings of this expression lead to different results. If we bracket the left two processes, a synchronisation occurs and $\tau$ is output, which subsequently will be a possible transition along with $\langle x \rangle$. However, if we bracket the right two processes, $\overline{a}$ will be pre-empted, meaning no synchronisation can occur. Thus even this solution doesn't work.

It seems to be clear that having an interruption action isn't going to work in a localised setting. Since $\tau$s should be abstracted and components viewed opaquely, $\tau$ must retain its privileged position, meaning that any solution would require a very different calculus to CCS. Thus, although the exploration of ICCS has been interesting, I will take it no further and seek another avenue for representing interruption.

## A.2 CaSE Generalised

In this section, I build a process calculus which is a direct extension of CaSE, which seeks to generalise the timeout operators. Since this calculus does not fundamentally alter the assumptions of CaSE, such as maximal progress, determinism and the way patience operates, the equivalence theory will be the same.

I showed in Chapter 6 Section 6.2 that CaSE cannot represent a clock choice because neither fragile timeout nor stable timeout provide a suitable basis. Fragile timeout only allows clock choices where other clocks not involved in the choice are held up, e.g. $\lfloor \lfloor \Delta \rfloor \sigma(E) \rfloor \rho(F)$, in which only $\sigma$ and $\rho$ can tick. Stable timeout allows patience but, since

the operator is partly static, when a clock tick does occur the enclosing stable timeout operators will not reduce. For instance, in $\lceil\lceil\mathbf{0}\rceil\sigma(E)\rceil\rho(F)$ if $\sigma$ ticks the enclosing timeout on $\rho$ will only reduce once $E$ performs a non-clock action, i.e. $\lceil\lceil\mathbf{0}\rceil\sigma(E)\rceil\rho(F) \xrightarrow{\sigma} \lceil E\rceil\rho(F)$ .

There is clearly a problem with the timeout operators which needs to be dealt with. The first option for solving this problem is adding a new "patience" operator to CaSE which overcomes the difficulties of fragile timeout:

$$\text{Wait}\ \frac{}{*.E \xrightarrow{\sigma} E}$$

The *wait* operator is similar to the patient **0** operator in CaSE, but more flexible. Rather than simply idling on clock ticks, it waits for *any* clock tick permitted by the context to occur, and then enables the guarded process when it does. When placed on the left-hand side of a fragile timeout (the interrupting process) this operator will tick over every clock except the timeout's own. For instance in the process $\lfloor *.E\rfloor\sigma(F)$ if $\sigma$ ticks, $F$ will become enabled. If any other clock ticks, $E$ will become enabled.

The wait operator, when combined with fragile timeout and recursion, can replace some of the functionality of stable timeout. In particular patient clock prefix can be written as $\sigma.E \triangleq \mu X.\lfloor *.X\rfloor\sigma(E)$, which is equivalent to $\lceil 0\rceil\sigma(E)$. I do not use stable timeout for any purpose other than patient clock ticks, so this allows me to expunge it from the language, which is in itself useful as stable timeout is not axiomatised by (Norton, 2005*a*). More importantly however, the new operator allows a fully patient clock choice operator to be produced, for example:

$$\sigma.E \oplus \rho.F \quad = \quad \mu X.\lfloor\lfloor *.X\rfloor\sigma(E)\rfloor\rho(F)$$

If $\sigma$ ticks then $E$ is enabled, if $\rho$ ticks then $F$ is enabled. Any other clock tick will cause the process to loop to $X$, achieving the same effect as idling, and thus achieving a patient clock choice which naturally extends to any number of options.

However, this is only half the problem. In addition to allowing any external clock to patiently tick, we also need to prevent other clocks in the phase transition system from ticking at the wrong time. But it is not possible to cleanly prevent a specific clock from ticking in a process which only performs clock actions. We could try to do this using $\Delta_\sigma$, for instance in trying to represent Figure 6.4 we might write:

$$\mu X.\lfloor *.X + \Delta_{\{\psi,\rho\}}\rfloor\sigma(\lfloor\mu Y.\lfloor *.Y + \Delta_\sigma\rfloor\psi(X)\rfloor\rho(\mu Z.\lfloor *.Z + \Delta_{\{\psi,\rho\}}\rfloor\sigma(Y)))$$

$$\text{where } \Delta_{\tilde{\sigma}} \triangleq \sum_{\sigma\in\tilde{\sigma}} \Delta_\sigma$$

This process uses $\Delta_\sigma$ summed with $*.X$ to allow only clocks not used by the process to tick at a particular moment. However this operator won't suffice, if still using the

standard CaSE definition of $+$, where both sides must advance on the clock simultaneously as in parallel composition. The process $*.E + \Delta_\sigma$ therefore leaves the $\Delta_\sigma$ in place *after* the clock ticks. As behaviour proceeds round the transition system, a context of $\Delta$s will build up, giving rise to a potentially infinite state transition system. This once again is similar to the original issue with stable timeout, where contexts would be left behind which should reduce. The problem thus lies in the semantics of $+$. Furthermore, the wait operator seems a little arbitrary and perhaps more of a "hack" than a solution. Thus, rather than simply modifying CaSE I will now proceed to look for a more general solution.

Temporal CCS (Moller and Tofts, 1990) is a real-time process calculus, which I briefly described in Chapter 2, Section 2.5.1. It contains two versions of the choice operator, called *strong choice* $+\!\!+$ and *weak choice* $+$. Strong choice behaves the same as $+$ in CaSE with respect to time – it must progress at the same rate on both sides. However weak choice allows time to *decide* which process should be taken, provided only one of the two can tick. If both can tick they progress in parallel to maintain clock determinism, as before.

I thus extend CaSE with the weak and strong choice operators. Furthermore, I replace the timeout operator with a simpler insistent clock prefix operator $\underline{\sigma}.E$, which was derived in CaSE as $\underline{\sigma}.E \triangleq \lfloor\Delta\rfloor\sigma(E)$. These operators partially solve our problem since weak choice is effectively the same as timed choice $\oplus$. In addition though we need a way to inhibit clocks that should not tick. Thus as a dual to the clock prefix operator, I introduce a *negative* clock prefix operator $\neg\sigma.E$, which reduces when any clock *other* than $\sigma$ ticks. I call this new calculus CaSE$^{\text{mt}}$, i.e. CaSE with Moller and Tofts' choice operator. The full syntax can be found in Table A.3, where I define CaSE$^{\text{mt}}$ expressions $\mathcal{E}$.

$$\Lambda = \{a, b, c \cdots\} \quad \overline{\Lambda} = \{\overline{a} | a \in \Lambda\} \quad \mathscr{A} = \Lambda \cup \overline{\Lambda} \cup \{\tau\} \quad \mathcal{T} = \{\sigma, \rho \cdots\}$$

$$\alpha, \beta \in \mathscr{A} \quad \gamma, \delta \in \mathscr{A} \cup \mathcal{T} \quad E, F, G \cdots \in \mathcal{E}$$

$$\begin{aligned} \mathcal{E} ::= \; & \mathbf{0} \mid \mathbf{1} \mid \alpha.\mathcal{E} \mid \underline{\sigma}.\mathcal{E} \mid \neg\sigma.\mathcal{E} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} +\!\!+ \mathcal{E} \mid \mathcal{E}|\mathcal{E} \mid \mathcal{E} \setminus a \\ \mid \; & \mathcal{E}\{\!|a \mapsto a|\!\} \mid \mathcal{E}/\sigma \mid \mu X.\mathcal{E} \mid X \end{aligned}$$

Table A.3: Syntax of CaSE$^{\text{mt}}$

In addition to changes already outlined I have also made some additional, primarily cosmetic changes. Firstly whereas CaSE has $\mathbf{0}$ and $\Delta$ to represent the maximally patient and stalled processes respectively, in CaSE$^{\text{mt}}$ these have been replaced by $\mathbf{1}$ and $\mathbf{0}$. I have made this alteration because $\mathbf{0}$ is then the process which can never allow any transitions, and thus is the identity of $+$. In contrast $\mathbf{1}$ allows all clock ticks, and is thus the identity of $+\!\!+$, with $\mathbf{0}$ being its annihilator.

The language's structural operational semantics are given in Table A.4 using a Labelled Transition System $(\mathcal{E}, \mathscr{A} \cup \mathcal{T}, \rightarrow)$ with $\rightarrow \subseteq \mathcal{E} \times \mathscr{A} \cup \mathcal{T} \times \mathcal{E}$. I write $E \xrightarrow{\gamma} E'$

whenever the rules provide a way for $E$ to evolve into $E'$ by doing a $\gamma$. In addition, I write $E \xrightarrow{\gamma}$ as shorthand for $\exists E'.E \xrightarrow{\gamma} E'$.

| | | | | | |
|---|---|---|---|---|---|
| Act | $\frac{-}{\alpha.E \xrightarrow{\alpha} E}$ | tTick | $\frac{-}{\underline{\sigma}.E \xrightarrow{\sigma} E}$ | tStall | $\frac{-}{\neg\sigma.E \xrightarrow{\rho} E}\,(1)$ |
| Sum1 | $\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$ | tSum1 | $\frac{E \xrightarrow{\sigma} E'}{E + F \xrightarrow{\sigma} E'}\,(a,b)$ | tPatient | $\frac{-}{a.E \xrightarrow{\sigma} a.E}$ |
| Sum2 | $\frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$ | tSum2 | $\frac{F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} F'}\,(a,c)$ | tIdle | $\frac{-}{\mathbf{1} \xrightarrow{\sigma} \mathbf{1}}$ |
| Sum3 | $\frac{E \xrightarrow{\alpha} E'}{E + \!\!+ F \xrightarrow{\alpha} E'}$ | tSum3 | $\frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'}$ | | |
| Sum4 | $\frac{F \xrightarrow{\alpha} F'}{E + \!\!+ F \xrightarrow{\alpha} F'}$ | tSum4 | $\frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E + \!\!+ F \xrightarrow{\sigma} E' + F'}$ | tCom | $\frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E \mid F \xrightarrow{\sigma} E' \mid F'}\,(d)$ |
| Com1 | $\frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$ | Com2 | $\frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$ | Com3 | $\frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\alpha} E' \mid F'}$ |
| Res | $\frac{E \xrightarrow{\gamma} E'}{E \setminus a \xrightarrow{\gamma} E' \setminus a}\,(2)$ | Rel | $\frac{E \xrightarrow{\gamma} E'}{E\{\!|f|\!\} \xrightarrow{f(\gamma)} E'\{\!|f|\!\}}$ | Rec | $\frac{E\{\mu X.E/X\} \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'}$ |
| Hid | $\frac{E \xrightarrow{\alpha} E'}{E/\sigma \xrightarrow{\alpha} E'/\sigma}$ | tHid1 | $\frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma}$ | tHid2 | $\frac{E \xrightarrow{\rho} E'}{E/\sigma \xrightarrow{\rho} E'/\sigma}\,(c)$ |

1) $\rho \neq \sigma$ 2) $\gamma \notin \{a, \overline{a}\}$ $a)$ $E + F \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$ $b)$ $F \xrightarrow{\sigma}\!\!\!\!\!\!\rightarrow$ $c)$ $E \xrightarrow{\sigma}\!\!\!\!\!\!\rightarrow$ $d)$ $E \mid F \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$

Table A.4: CaSE$^{\text{mt}}$ Operational Semantics

The majority of the CCS rules are present and unchanged, although the sum rules, Sum1 and Sum2 for actions are replicated for weak choice and strong choice, Sum3 and Sum4 respectively. Rules prefixed with a lower-case t are timed rules, relating to clock ticks. Clock prefix is handled by rule tTick. Notice that the only possible transition that this operator can make is the parametrised clock tick – there are no other matching rules. This is important because it means that when placed in a choice a clock prefix will not permit patience. Therefore when placed in a choice, it will not cause a synchronisation with the other side. For instance $\underline{\sigma}.E + \underline{\rho}.F$ performs an exclusive clock choice – if clock prefix were patient this would act exactly the same as its CaSE counterpart. The weak choice operator $+$ has three rules to define its behaviour, which effectively mirror the rules of normal action choice. The first rule tSum1 states that if $E$ can perform a $\sigma$ tick to $E'$ and $F$ cannot perform a $\sigma$ tick, then $E + F$ can tick to $E'$ (provided no $\tau$s are present). The second rule tSum2 is the reflection of this rule for $F$. The third rule tSum3 is the same as the standard CaSE rule, which simply states that if both sides tick simultaneously then both advance. These three rules together provide a fully deterministic clock choice operator, which only allows choice resolution if exactly one side permits the clock to tick.

Strong choice is handled by the standard CaSE rule alone in tSum4. It cannot be removed altogether because it *requires* that both sides tick to allow an advance and is thus analogous to logical AND. Negative clock prefix is described by rule tStall, which states that any clock other than the parametrised clock $\sigma$ will allow the process to reduce, while $\sigma$ itself is held-up as in CaSE's $\Delta$ operator. The remainder of the rules are identical to their CaSE equivalents. The representation of our phase transition system in Figure 6.4 is now possible:

$$\mu X.(\sigma.\mu Y.(\psi.X + \rho.\mu Z.(\sigma.Y + (\neg\rho.Z + \neg\psi.\mathbf{0})) + \neg\sigma.Y) + (\neg\rho.X + \neg\psi.\mathbf{0}))$$

Nevertheless this does seem very complicated for such a simple process. The issue lies in making sure only the correct clocks are held up at the correct instant. Instead of using composed $\Delta$s which leave a context behind, I now use composed negative clock ticks. For instance $\neg\rho.X + \neg\psi.\mathbf{0}$ prevents $\rho$ and $\psi$ (i.e. NOT $\rho$ AND NOT $\psi$). When another clock ticks, the process simply recurses as it should, emulating a patient wait.

So why not have $\neg\rho.X + \neg\psi.X$? The reason is because even now when a clock ticks on both sides it *must* advance both sides – this is the only way clock determinism can be honoured. If both processes recursed back to $X$ we'd have the same problem as before except a lot worse, with two identical copies existing! Instead, only the first negative prefix in the summation guards the recursion variable $X$, the rest guard a $\mathbf{0}$. This still leaves a context behind, but it disappears on any transition and isn't such a problem. Nevertheless this process is unnecessarily complex for such a simple system.

What the exploration of this calculus shows is that adding extra operators for defining precisely which clocks are enabled and which are held up, whilst flexible, leads to very complex processes. The problem, therefore, is more one of how patience is handled fundamentally within the calculus's semantics. It shouldn't be necessary to explicitly disable a clock tick if it is not wanted, rather the calculus should assume that a clock tick isn't required if it is not explicitly present.

# Appendix B

# Proofs for Chapter 6

## B.1 Proof of Proposition 6.5.11

For any process $E$, if $\alpha \in \mathscr{A}_E$ then there exists an $E'$ such that $E \xrightarrow{\alpha} E'$.

**Proof.** By induction on the structure of process $E$.

- Cases $E \equiv \mathbf{0}$, $E \equiv \Delta$, $E \equiv \Delta_\rho$ and $E \equiv X$ trivially satisfy the statement since then $\alpha \notin \mathscr{A}_E$.
- $E \equiv \sigma.F$. Trivially true since $\alpha \notin \mathscr{A}_{\sigma.F}$.
- $E \equiv \beta.F$. If $\alpha \neq \beta$ then this is trivially true. Otherwise, $E \equiv \alpha.F$ and by rule Act $\alpha.F \xrightarrow{\alpha} F$ as required.
- $E \equiv F + G$. By hypothesis, $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$ and $\alpha \in \mathscr{A}_G$ implies $\exists G'.G \xrightarrow{\alpha} G'$. We know that $\alpha \in \mathscr{A}_{F+G}$ and therefore by definition either $\alpha \in \mathscr{A}_F$ or $\alpha \in \mathscr{A}_G$. Hence by rules Sum1 and Sum2, either $F + G \xrightarrow{\alpha} F'$ or $F + G \xrightarrow{\alpha} G'$ as required.
- $E \equiv \mu X.F$, where $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$. By definition $\mathscr{A}_{\mu X.F} = \mathscr{A}_F$, hence $\alpha \in \mathscr{A}_F$ and $\exists F'.F \xrightarrow{\alpha} F'$. Since Act is the only rule which can induce an $\alpha$ transition without preconditions we know that an expression of the form $\alpha.G$ for some $G$ must be a subexpression of $F$, and thus also of $\mu X.F$. Since the substitution cannot remove these subexpressions it follows that $F\{\mu X.F/X\} \xrightarrow{\alpha} F'\{\mu X.F/X\}$ and hence also $\mu X.F \xrightarrow{\alpha} F'\{\mu X.F/X\}$ as required.
- $E \equiv F \mid G$, where $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$ and $\alpha \in \mathscr{A}_G$ implies $\exists G'.G \xrightarrow{\alpha} G'$. We know that $\alpha \in \mathscr{A}_{F \mid G}$ and therefore by definition either $\alpha \in \mathscr{A}_F$, $\alpha \in \mathscr{A}_G$, or $a \in \mathscr{A}_F$ and $\overline{a} \in \mathscr{A}_G$ for some $a$ with $\alpha = \tau$. The first two cases follow using the same proof as for the $+$ operator but using rules Com1 and Com2. Otherwise $a \in \mathscr{A}_F$ and $\overline{a} \in \mathscr{A}_G$, so by hypothesis there exist $F', G'$ such that $F \xrightarrow{a} F'$ and $G \xrightarrow{\overline{a}} G'$. Then by rule Com3 it follows that $F \mid G \xrightarrow{\tau} F' \mid G'$ as required.

- $E \equiv F \setminus a$, where $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$. Since $\alpha \in \mathscr{A}_{F\setminus a}$ we know by definition that $\alpha \in \mathscr{A}_F$ and $\alpha \notin \{a, \overline{a}\}$. Therefore $F \xrightarrow{\alpha} F'$ for some $F'$, and by Res, $F \setminus a \xrightarrow{\alpha} F' \setminus a$ as required.

- $E \equiv F/\sigma$. If $\alpha \neq \tau$ then this is trivially true, as then $\mathscr{A}_{F/\sigma} = \mathscr{A}_F$, and therefore by the inductive hypothesis $F \xrightarrow{\alpha} F'$ and hence by rule Hid $F/\sigma \xrightarrow{\alpha} F'/\sigma$ as required. Otherwise either $\tau \in \mathscr{A}_F$ (then the former case holds), or else $\sigma \in \mathcal{T}_F$. In the latter case by Lemma 6.5.12 it follows that $\exists F'.F \xrightarrow{\sigma} F'$. Therefore, by rule tHid1 it follows that $F/\sigma \xrightarrow{\tau} F'/\sigma$ as required.

- $E \equiv F\{\!|a \mapsto b|\!\}$. Then $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$. We know that $\alpha \in \mathscr{A}_{F\{\!|a\mapsto b|\!\}}$. If $\alpha = b$ then either $a \in \mathscr{A}_F$ or $\alpha \in \mathscr{A}_F$. Therefore by the inductive hypothesis either $F \xrightarrow{\alpha} F'$ or $F \xrightarrow{a} F'$, and it follows by Rel that $F\{\!|a \mapsto b|\!\} \xrightarrow{\alpha} F'\{\!|a \mapsto b|\!\}$ or $F\{\!|a \mapsto b|\!\} \xrightarrow{b} F'\{\!|a \mapsto b|\!\}$ as required.

- $E \equiv F\{\!|\sigma \mapsto a|\!\}$, where $\alpha \in \mathscr{A}_F$ implies $\exists F'.F \xrightarrow{\alpha} F'$. We know that $\alpha \in \mathscr{A}_{F\{\!|\sigma\mapsto a|\!\}}$. If $\alpha \neq a$ then $\mathscr{A}_F = \mathscr{A}_{F\{\!|\sigma\mapsto a|\!\}}$ and $F \xrightarrow{\alpha} F'$ hence by Rel, $F\{\!|\sigma \mapsto a|\!\} \xrightarrow{\alpha} F'\{\!|\sigma \mapsto a|\!\}$ as required. Otherwise either $a \in \mathscr{A}_F$ (then the former case holds), or else $\sigma \in \mathcal{T}_F$. In the latter case by Lemma 6.5.12 it follows that $\exists F'.F \xrightarrow{\sigma} F'$. Therefore, by rule Rel it follows that $F\{\!|\sigma \mapsto a|\!\} \xrightarrow{a} F'\{\!|\sigma \mapsto a|\!\}$ as required.

Each case is proven and thus the inductive proof is complete. □

## B.2 Proof of Proposition 6.5.12

For any process $E$, if $\sigma \in \mathcal{T}_E$ then there exists an $E'$ such that $E \xrightarrow{\sigma} E'$.

**Proof.** By induction on the structure of process $E$.

- Cases $E \equiv \mathbf{0}$, $E \equiv \Delta$, $E \equiv \Delta_\rho$ and $E \equiv X$ are trivially true since then $\sigma \notin \mathcal{T}_E$.

- $E \equiv \alpha.F$. Trivially true since $\sigma \notin \mathcal{T}_{\alpha.F}$.

- $E \equiv \rho.F$. If $\sigma \neq \rho$ then this is trivial. Otherwise, $E \equiv \sigma.F$ and by rule Act $\sigma.F \xrightarrow{\sigma} F$ as required.

- $E \equiv F + G$. Then $\sigma \in \mathcal{T}_F$ implies $\exists F'.F \xrightarrow{\sigma} F'$ and $\sigma \in \mathcal{T}_G$ implies $\exists G'.G \xrightarrow{\sigma} G'$. We know that $\sigma \in \mathcal{T}_{F+G}$ and therefore by definition $\sigma \notin \Sigma_{F+G}$, and either $\sigma \in \mathcal{T}_F$, $\sigma \in \mathcal{T}_G$, or both. Therefore by the inductive hypothesis there exist $F', G'$ such that $F \xrightarrow{\sigma} F'$, $G \xrightarrow{\sigma} G'$, or both. If both are true then by tSum1 $F + G \xrightarrow{\sigma} F' + G'$. If $\sigma \in \mathcal{T}_F$ only, then by rule tSum2 and the fact that $\sigma \notin \mathcal{T}_G$ and $\sigma \notin \Sigma_G$ we have $F + G \xrightarrow{\sigma} F'$. Similarly, if $\sigma \in \mathcal{T}_G$ only, then by rule tSum3 $F + G \xrightarrow{\sigma} G'$. Therefore, in all cases $\exists E'.F + G \xrightarrow{\sigma} E'$ as required.

- $E \equiv \mu X.F$, where $\sigma \in \mathcal{T}_F$ implies $\exists F'.F \xrightarrow{\sigma} F'$. Since $\sigma \in \mathcal{T}_{\mu X.F} = \mathcal{T}_F$ and there is an $F'$ such that $F \xrightarrow{\sigma} F'$. By induction on the operational rules it follows that there is a process $G$ such that $\sigma.G$ is a subexpression of $F$, since this is the only process which can induce a $\sigma$ transition with no preconditions. Furthermore, it follows that $\Delta$, $\Delta_\sigma$ and $\tau.H$ for any $H$ are *not* subexpressions of $F$, as this would prevent a $\sigma$ transition. These subexpression statements also hold for $\mu X.F$. Now, we know that the substitution $F\{\mu X.F/X\}$ cannot introduce subexpressions not in $\mu X.F$ and $F$. Hence, it follows that $\sigma.G$ is still a subexpression of $F\{\mu X.F/X\}$ and $\Delta, \Delta_\sigma$ and $\tau.H$ for any $H$ aren't. Prevention of a $\sigma$ transition would require, by rules tSum1, tSum2 and tSum3, that $\sigma \in \Sigma_F$ (since rule Rec can only produce more transitions not fewer than its encapsulated expression), but this would only be possible if $\Delta$, $\Delta_\sigma$ or $\tau.H$ were subexpressions. Since we know they aren't and $\sigma.G$, it follows that an $F'$ is already instantiated such that $F\{\mu X.F/X\} \xrightarrow{\sigma} F'$ and hence $\mu X.F \xrightarrow{\sigma} F'$ as required.

- $E \equiv F \mid G$, with $\sigma \in \mathcal{T}_F$ implies $\exists F'.F \xrightarrow{\sigma} F'$ and $\sigma \in \mathcal{T}_G$ implies $\exists G'.G \xrightarrow{\sigma} G'$. By hypothesis, $\sigma \in \mathcal{T}_{F \mid G}$. First it is necessary to prove that $\tau \notin \mathscr{A}_{F \mid G}$, as this side-condition is required by all three clock rules for parallel composition. This can be shown by application of Lemmas 6.5.9 and 6.5.10 since we know that $\sigma \in \mathcal{T}_{F \mid G}$. Since $\mathcal{T}_{F \mid G} = \mathcal{T}_F \cup \mathcal{T}_G$, this gives rise to three possibilites:

  - $\sigma \in \mathcal{T}_F$ and $\sigma \notin \mathcal{T}_G$. Therefore $\exists F'.F \xrightarrow{\sigma} F'$. Since we know that $\sigma \notin \Sigma_{F \mid G}$, then $\sigma \notin \Sigma_G$. Since we already know that $\tau \notin \mathscr{A}_{F \mid G}$ it follows by rule tCom2 that $F \mid G \xrightarrow{\sigma} F'$ as required.
  - $\sigma \in \mathcal{T}_G$ and $\sigma \notin \mathcal{T}_F$. Follows the symmetric argument of the above with the final application of rule tCom3.
  - $\sigma \in \mathcal{T}_F$ and $\sigma \in \mathcal{T}_G$. Therefore $\exists F'.F \xrightarrow{\sigma} F'$ and $\exists G'.G \xrightarrow{\sigma} G'$. Since we already know that $\tau \notin \mathscr{A}_{F \mid G}$ it follows by rule tCom1 that $F \mid G \xrightarrow{\sigma} F' \mid G'$ as required.

- $E \equiv F \setminus a$. Restriction has no effect on clocks, so this case follows by simple induction.

- $E \equiv F/\rho$, where $\sigma \in \mathcal{T}_F$ implies $\exists F'.F \xrightarrow{\sigma} F'$. If $\sigma = \rho$ then $\mathcal{T}_E = \emptyset$ and the result follows. If $\rho \in \mathcal{T}_F$ then likewise $\mathcal{T}_E = \emptyset$. Otherwise, it follows that $\mathcal{T}_{F/\rho} = \mathcal{T}_F$, and therefore $\exists F'.F \xrightarrow{\sigma} F'$ as required.

- $E \equiv F\{\!|a \mapsto b|\!\}$. Action renaming has no effect on clocks, so this case follows by simple induction.

- $E \equiv F\{\!|\rho \mapsto a|\!\}$, where $\sigma \in \mathcal{T}_F$ implies $\exists F'.F \xrightarrow{\sigma} F'$. If $\sigma = \rho$ then $\sigma \notin \mathcal{T}_{F\{\!|\rho \mapsto a|\!\}}$ and thus the result follows trivially. Otherwise $\sigma \in \mathcal{T}_F$ and thus $\exists F'.F \xrightarrow{\sigma} F'$ as required.

Each case is proven and thus the inductive proof is complete. □

## B.3 Proof of Proposition 6.5.13

For any process $E$, if $\exists E'.E \xrightarrow{\gamma} E'$ then $\gamma \in \mathscr{A}_E \cup \mathcal{T}_E$.

**Proof.** By induction on the structure of process $E$, and then case analysis using every rule in Table 6.3. We can therefore omit processes which do not match any rule.

- $E \equiv \gamma.F$. By rule Act $E \xrightarrow{\gamma} F$. If $\gamma = \alpha$ then $\mathscr{A}_{\alpha.F} = \{\alpha\}$, thus satisfying the statement. If $\gamma = \sigma$ then $\mathcal{T}_{\sigma.F} = \{\sigma\}$, also satisfying the statement.
- $E \equiv F + G$, with (1) $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$ and (2) $\exists G'.G \xrightarrow{\gamma} G'$ implies $\gamma \in \mathscr{A}_G \cup \mathcal{T}_G$. The following rules can apply:
  - Sum1. Then $F \xrightarrow{\alpha} F'$ with $\gamma = \alpha$. Hence by assumption (1) $\alpha \in \mathscr{A}_F$ and therefore $\alpha \in \mathscr{A}_{F+G}$ as required.
  - Sum2. Then $G \xrightarrow{\alpha} G'$ with $\gamma = \alpha$. Hence by assumption (2) $\alpha \in \mathscr{A}_G$ and therefore $\alpha \in \mathscr{A}_{F+G}$ as required.
  - tSum1. Then $F \xrightarrow{\sigma} F'$ and $G \xrightarrow{\sigma} G'$. Therefore it follows by assumptions (1) and (2) that $\sigma \in \mathcal{T}_F$ and $\sigma \in \mathcal{T}_G$. Then by Proposition 6.5.10 it follows that $\sigma \notin \Sigma_F$ and $\sigma \notin \Sigma_G$. Therefore $\sigma \in \mathcal{T}_{F+G}$.
  - tSum2. Then $F \xrightarrow{\sigma} F'$ and $\sigma \notin \mathcal{T}_G \cup \Sigma_G$. By assumption (1) it follows that $\sigma \in \mathcal{T}_F$. Furthermore, by Proposition 6.5.10 it follows that $\sigma \notin \Sigma_F$ and therefore $\sigma \in \mathcal{T}_F \cup \mathcal{T}_G \setminus \Sigma_F \cup \Sigma_G = \mathcal{T}_{F+G}$.
  - tSum3. The same reasoning can be applied in the case of tSum2.
- $E \equiv \mu X.F$, with (1) $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. By rule Rec it follows that $F\{\mu X.F/X\} \xrightarrow{\gamma} E'$. We know that $F$ is sequential, and therefore the only way to induce a transition in $F\{\mu X.F/X\}$ is via rule Act. Hence it follows that $\gamma.G$ for some $G$ is a subexpression of $F$. Therefore $F$ without substitutions can only contain additional unguarded $X$ variables which do not prevent even a $\sigma$ transition from being produced. Therefore by assumption (1) it follows that $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. Since $\mathscr{A}_{\mu X.E} = \mathscr{A}_E$ and $\mathcal{T}_{\mu X.E} = \mathcal{T}_E$ we are done.
- $E \equiv F \mid G$, with (1) $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$ and (2) $\exists G'.G \xrightarrow{\gamma} G'$ implies $\gamma \in \mathscr{A}_G \cup \mathcal{T}_G$. The following rules can apply:
  - Com1. Then $F \xrightarrow{\alpha} F'$ with $\gamma = \alpha$. Hence by assumption (1) $\alpha \in \mathscr{A}_F$ and therefore $\alpha \in \mathscr{A}_{F \mid G}$ as required.
  - Com2. Then $G \xrightarrow{\alpha} G'$ with $\gamma = \alpha$. Hence by assumption (2) $\alpha \in \mathscr{A}_G$ and therefore $\alpha \in \mathscr{A}_{F \mid G}$ as required.

- Com3. Then $F \xrightarrow{a} F'$ and $G \xrightarrow{\overline{a}} G'$. Hence by assumption (1) $a \in \mathscr{A}_F$ and by assumption (2) $\overline{a} \in \mathscr{A}_G$. Therefore $\tau \in \{\tau \mid a \in \mathscr{A}_F \wedge \overline{a} \in \mathscr{A}_G\}$ and thus $a \in \mathscr{A}_{F \mid G}$ as required.
- tCom1. Then $F \xrightarrow{\sigma} F'$ and $G \xrightarrow{\sigma} G'$. Therefore it follows by assumptions (1) and (2) that $\sigma \in \mathcal{T}_F$ and $\sigma \in \mathcal{T}_G$. Then by Proposition 6.5.10 it follows that $\sigma \notin \Sigma_F$ and $\sigma \notin \Sigma_G$. Therefore $\sigma \in \mathcal{T}_{F \mid G}$.
- tCom2. Then $F \xrightarrow{\sigma} F'$, $\sigma \notin \Sigma_G \cup \mathcal{T}_G$ and $\tau \notin \mathscr{A}_{F \mid G}$. By assumption (1) $\sigma \in \mathcal{T}_F$ and by Proposition 6.5.10 it follows that $\sigma \notin \Sigma_F$. Therefore $\sigma \in (\mathcal{T}_F \cup \mathcal{T}_G) \setminus (\Sigma_F \cup \Sigma_G) = \mathcal{T}_{F \mid G}$ as required.
- tCom3. Uses the same reasoning as tCom2.

- $E \equiv F \setminus a$, with $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. Then by rule Res it follows that $F \xrightarrow{\gamma} F'$ and $\gamma \notin \{a, \overline{a}\}$. Hence $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. If $\gamma = \sigma$ then $\sigma \in \mathcal{T}_F = \mathcal{T}_{F \setminus a}$ as required. Otherwise, if $\gamma = \alpha$ then $\alpha \in \mathscr{A}_F \setminus \{a, \overline{a}\} = \mathscr{A}_{F \setminus a}$, since $\alpha \notin \{a, \overline{a}\}$.

- $E \equiv F\{\!|\sigma \mapsto a|\!\}$, with $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. If $F \xrightarrow{\gamma} F'$ then $\gamma \in \mathscr{A}_{F\{\!|\sigma \mapsto a|\!\}} \cup \mathcal{T}_{F\{\!|\sigma \mapsto a|\!\}}$ as required. Otherwise, if $\gamma = a$ then either $F \xrightarrow{\sigma} F'$ and hence $\sigma \in \mathcal{T}_F$. Therefore $\sigma \in \mathcal{T}_{F\{\!|\sigma \mapsto a|\!\}}$ as required.

- $E \equiv F\{\!|a \mapsto b|\!\}$, with $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. This follows using a very similar argument to the above.

- $E \equiv F/\sigma$, with (1) $\exists F'.F \xrightarrow{\gamma} F'$ implies $\gamma \in \mathscr{A}_F \cup \mathcal{T}_F$. The following rules can apply:
  - Hid. Then $\gamma = \alpha$ and $F \xrightarrow{\alpha} F'$. Hence by assumption (1) it follows that $\alpha \in \mathscr{A}_F$ and therefore $\alpha \in \mathscr{A}_{F/\sigma}$ as required.
  - tHid1. Then $\gamma = \tau$ and $F \xrightarrow{\sigma} F'$. Hence by assumption (1) it follows that $\sigma \in \mathcal{T}_F$ and therefore $\tau \in \mathscr{A}_F \cup \{\tau \mid \sigma \in \mathcal{T}_E\} = \mathscr{A}_{F/\sigma}$ as required.
  - tHid2. Then $\gamma = \rho$, $F \xrightarrow{\rho} F'$ and $\sigma \notin \mathcal{T}_F$. Hence by assumption (1) it follows that $\rho \in \mathcal{T}_F$ and since $\sigma \notin \mathcal{T}_F$, $\mathcal{T}_{F/\sigma} = \mathcal{T}_F$ as required.

Each case is proven and thus the inductive proof is complete. □

# Bibliography


Aceto, L., Fokkink, W., Ingólfsdóttir, A. and Luttik, B. (2005), 'CCS with Hennessy's merge has no finite-equational axiomatization', *Theoretical Computer Science* **330**(3), 377–405.

Alur, R., Henzinger, T., Kupferman, O. and Vardi, M. (1998), Alternating refinement relations, *in* 'Proc. 9th Intl. Conference on Concurrency Theory (CONCUR '98)', Vol. 1446 of *Lecture Notes in Computer Science*, Springer, pp. 163–178.

Andersen, H. and Mendler, M. (1994), An asynchronous process algebra with multiple clocks, *in* 'Proc. 5th European Symposium on Programming (ESOP '94)', Vol. 788 of *Lecture Notes in Computer Science*, Springer, pp. 58–73.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. and Weerawarana, S. (2003), Business Process Execution Language for Web-Services version 1.1, Technical report, IBM. Accessed 10/03/2009.
**URL:** *http://www.oasis-open.org/committees/download.php/2046/BPELV1-1May52003Final.pdf*

Ankolekar, A., Huch, F. and Sycara, K. P. (2002), Concurrent semantics for the web services specification language DAML-S, *in* F. Arbab and C. Talcott, eds, 'Proc. 5th Intl. Conference on Coordination Models and Languages', Vol. 2315 of *Lecture Notes in Computer Science*, Springer, pp. 14–21.

Baeten, J. C. M. and Verhoef, C. (1995), Concrete process algebra, *in* 'Handbook of logic in computer science (vol. 4): Semantic Modelling', Oxford University Press, Oxford, UK, pp. 149–268.

Barros, A., Dumas, M. and Hofstede, A. H. M. (2005), Service interaction patterns, *in* 'Proc. 3rd Intl. Conference on Business Process Management (BPM 2005)', Vol. 3649 of *Lecture Notes in Computer Science*, Springer, pp. 302–318.

Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J. and Tabet, S. (2005), Semantic Web Services Ontology (SWSO), Technical report, SWSI. Accessed 27/03/2009.
**URL:** *http://www.daml.org/services/swsf/1.0/swso/*

Bergstra, J. A., Ponse, A. and Smolka, S. A. (2001), Preface, *in* J. A. Bergstra, A. Ponse and S. A. Smolka, eds, 'Handbook of Process Algebra', North-Holland, pp. v–ix.

Bergstra, J. and Klop, J. (1984), 'Process algebra for synchronous communication', *Information and Control* **60**(1/3), 109–137.

Bocchi, L., Laneve, C. and Zavattaro, G. (2003), A calculus for long-running transactions, *in* 'Proc. of the 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)', Vol. 2884 of *Lecture Notes in Computer Science*, Springer, pp. 124–138.

Börger, E. (1999), High level system design and analysis using abstract state machines, *in* W. Stephan, P. Traverso and M. Ullman, eds, 'Current Trends in Applied Formal Methods (FM-Trends 98)', Vol. 1641 of *Lecture Notes in Computer Science*, Springer, pp. 1–43.

Box, D., Ehnebuske, D., Kakivaya, G., Laymann, A., Mendelsohn, N., Nielsen, H. F., Thatte, S. and Winer, D. (2000), Simple object access protocol (SOAP) 1.1, Technical report, World Wide Web Consortium. Accessed 10/03/2009.
**URL:** `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`

Bravetti, M. and Zavattaro, G. (2007), 'Service oriented computing from a process algebraic perspective', *Journal of Logic and Algebraic Programming* **70**(1), 3–14.

Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H. and Montanari, U. (2005), Comparing two approaches to compensable flow composition, *in* 'Proc. 16th Intl. Conference on Concurrency Theory (CONCUR 2005)', Vol. 3653 of *Lecture Notes in Computer Science*, Springer, pp. 383 – 397.

Bruni, R., Melgratti, H. and Montanari, U. (2005), Theoretical foundations for compensations in flow composition languages, *in* 'Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages (POPL'05)', ACM Press, New York, NY, USA, pp. 209–220.

Butler, M. and Ferreira, C. (2004), An operational semantics for StAC, a language for modelling long-running business transactions, *in* R. D. Nicola, G. Ferrari and G. Meredith, eds, 'Proc. of Coordination 2004', Vol. 2949 of *Lecture Notes in Computer Science*, Springer, pp. 87–104.

Butler, M., Ferreira, C. and Ng, M. (2005), 'Precise modelling of compensating business transactions and its application to BPEL', *Journal of Universal Computer Science* **11**(5), 712–743.

Butler, M. J., Hoare, C. A. R. and Ferreira, C. (2005), A trace semantics for long-running transactions, *in* A. Abdallah, C. B. Jones and J. Sanders, eds, '25 years of CSP', Vol. 3525 of *Lecture Notes in Computer Science*, Springer, pp. 133–150.

Butler, M. and Ripon, S. (2005), Executable semantics for compensating CSP, *in* M. Bravetti, L. Kloul and G. Zavattaro, eds, 'Proc. 2nd Intl. Workshop on Web Services and Formal Methods (WS-FM 2005)', Vol. 3670 of *Lecture Notes in Computer Science*, Springer, pp. 243–256.

Chakravarty, M. M. T., Keller, G., Peyton-Jones, S. and Marlow, S. (2005), Associated types with class, *in* 'Proc. 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2005)', Vol. 40 of *ACM SIGPLAN Notices*, ACM, New York, NY, USA, pp. 1–13.

Chessell, M., Ferreira, C., Griffin, C., Henderson, P., Vines, D. and Butler, M. (2002), 'Extending the concept of transaction compensation', *IBM Systems Journal* **41**(4), 743–758.

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. (2001), Web services description language (WSDL) 1.1, Technical report, World Wide Web Consortium. Accessed 10/03/2009.
**URL:** `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`

Cleaveland, R. and Hennessy, M. (1990), 'Priorities in process algebras', *Information and Computation* **87**(1), 58–77.

Cleaveland, R., Lüttgen, G. and Mendler, M. (1997), An algebraic theory of multiple clocks, *in* '8th Intl. Conference on Concurrency Theory (CONCUR '97)', Vol. 1243 of *Lecture Notes in Computer Science*, Springer, pp. 166–180.

de Alfaro, L. and Henzinger, T. A. (2001), 'Interface automata', *SIGSOFT Software Engineering Notes* **26**(5), 109–120.

de Bruijn, N. (1972), 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem', *Indagationes Mathematicae* **34**(5), 381–392.

Dijkstra, E. (1975), 'Guarded commands, nondeterminacy and formal derivation of programs', *Communications of the ACM* **18**(8), 453–457.

Ellson, J., Gansner, E., Koutsofios, E., North, S. and Woodhull, G. (2004), Graphviz and dynagraph – static and dynamic graph drawing tools, *in* M. Jünger and P. Mutzel, eds, '*Graph Drawing Software*', Springer, chapter 5, pp. 127–148.

Erwig, M. (2001), 'Inductive graphs and functional graph algorithms', *Journal of Functional Programming* **11**(05), 467–492.

Fensel, D., Polleres, A. and de Bruijn, J. (2007), 'D14v1.0. ontology-based choreography'. Accessed 10/03/2009.
**URL:** `http://www.wsmo.org/TR/d14/v1.0/`

Foster, S. (2005), HAIFA : An XML based interoperability solution for Haskell, *in* 'Preproc. 6th Symposium on Trends in Functional Programming (TFP 2005)', Tartu University Press, pp. 103–118.

Foster, S. (2007), Modelling compensation in timed process algebra, *in* 'Proc. 2nd European Young Researcher's Workshop on Service Oriented Computing (YR-SOC 2007)', pp. 31–37.

Foster, S., Hughes, A. and Norton, B. (2005), Composition and semantic enhancement of web-services : The CASheW-s project, *in* 'Proc. 1st Young Researcher's Workshop on Service Oriented Computing (YR-SOC 2005)', pp. 29–32.

Garcia-Molina, H. and Salem, K. (1987), Sagas, *in* 'Proc. of the 1987 ACM SIGMOD international conference on Management of data (SIGMOD '87)', ACM Press, New York, NY, USA, pp. 249–259.

Gray, J. (1981), The transaction concept: Virtues and limitations (invited paper), *in* 'Proc. 7th Intl. Conference on Very Large Data Bases (VLDB 1981)', IEEE Computer Society, pp. 144–154.

Haas, H. and Brown, A. (2004), Web-services glossary, Technical report, W3C Working Group. Accessed 10/03/2009.
**URL:** `http://www.w3.org/TR/ws-gloss`

Haerder, T. and Reuter, A. (1983), 'Principles of transaction-oriented database recovery', *ACM Computing Surveys (CSUR)* **15**(4), 287–317.

Hennessy, M. (1993), Timed process algberas: A tutorial, Technical Report 1993:02, COGS, University of Sussex.

Hennessy, M. and Regan, T. (1995), 'A process algebra for timed systems', *Information and Computation* **117**(2), 221–239.

Hoare, C. A. R. (1978), 'Communicating sequential processes', *Communications of the ACM* **21**(8), 666–677.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall.
**URL:** `http://www.usingcsp.com`

Jordan, D. and Evdemon, J. (2007), Web Services Business Process Execution Language version 2.0, Technical report, OASIS. Accessed 10/03/2009.
**URL:** *http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf*

Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y. and Barreto, C. (2005), Web services choreography description language version, Technical report, World Wide Web Consortium.
**URL:** `http://www.w3.org/TR/ws-cdl-10/`

Kick, M. (1999), Modelling synchrony and asynchrony with multiple clocks, Master's thesis, University of Passau.

Kuropka, D. and Nern, H. (2006), 'Semantics poses challenge for web services', *ICT Results* . Accessed 10/03/2009.
**URL:** *http://cordis.europa.eu/ictresults/index.cfm/section/news/Tpl/article/ID/82454*

Laneve, C. and Zavattaro, G. (2005), Foundations of web transactions, *in* 'Proc. Foundations of Software Science and Computational Structures (FOSSACS 05)', Vol. 3441 of *Lecture Notes in Computer Science*, Springer, pp. 282–298.

Lucchi, R. and Mazzara, M. (2007), 'A pi-calculus based semantics for WS-BPEL', *Journal of Logic and Algebraic Programming (JLAP)* **70**, 96–118.

Lüttgen, G. (1998), *Pre-emptive Modeling of Concurrent and Distributed Systems*, Shaker Verlag. ISBN 3-8265-3932-X.

Lüttgen, G. and Mendler, M. (2005), When 1 clock is not enough, *in* L. Aceto and A. Gordon, eds, 'Intl. Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond (PA 2005)', Vol. NS-05-3 of *BRICS Notes Series*, BRICS – Basic Research in Computer Science, Bertinoro, Italy, pp. 155–158.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayana, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N. and Sycara, K. (2004), OWL-S : Semantic markup for web services, Technical report, OWL-S coalition. Accessed 10/03/2009.
**URL:** *http://www.daml.org/services/owl-s/1.1/overview/*

Martin, J., Hailpern, B., Tarr, P. and Arsanjani, A. (2003), 'Web services: Promises and compromises', *Queue* **1**(1), 48–58.

McBride, C. and McKinna, J. (2004), 'The view from the left', *Journal of Functional Programming* **14**(01), 69–111.

McIlraith, S. A., Son, T. C. and Zeng, H. (2001), 'Semantic web services', *IEEE Intelligent Systems* **16**(2), 46–53.

Milner, A. J. R. G. (1980), *A Calculus of Communicating Systems*, Vol. 92 of *Lecture Notes in Computer Science*, Springer.

Milner, A. J. R. G. (1989*a*), *Communication and Concurrency*, Prentice Hall.

Milner, A. J. R. G. (1989*b*), 'A complete axiomatisation for observation congruence of finite-state behaviours', *Information and Computation* **81**(2), 227–247.

Milner, A. J. R. G. (1999), *Communicating and Mobile Systems: The Pi-Calculus*, Cambridge University Press.

Misra, J. and Cook, W. R. (2007), 'Computation orchestration: A basis for wide-area computing', *Journal of Software and Systems Modeling* **6**(1), 83–110.

Moller, F. and Tofts, C. M. N. (1990), A temporal calculus of communicating systems, *in* 'Proc. Intl. Conference on Concurrency Theory (CONCUR '90)', Vol. 458 of *Lecture Notes in Computer Sciences*, pp. 401–415.

Narayanan, S. and McIlraith, S. A. (2002), Simulation, verification and automated composition of web services, *in* 'Proc. 11th Intl. World Wide Web Conference (WWW2002)', pp. 77–88.

Nicollin, X., Sifakis, J. and Yovine, S. (1993), 'From ATP to timed graphs and hybrid systems', *Acta Informatica* **30**(2), 181–202.

Norell, U. (2007), Towards a practical programming language based on dependent type theory, PhD thesis, Chalmers University of Technology.

Norton, B. (2005*a*), Behavioural types for synchronous software composition, *in* 'Proc. of Workshop on Foundations of Interface Technologies (FIT 2005)'. Accessed 10/02/2009.
**URL:** *http://dip.semanticweb.org/documents/Barry-Norton-Behavioural-Types-for-Synchronous-Software-Composition.pdf*

Norton, B. (2005*b*), Experiences with OWL-S, directions for service composition: The Cashew position, *in* 'Proc. OWL : Experiences and Directions Workshop (OWLED 2005)'.

Norton, B. and Fairtlough, M. (2004), Reactive types for dataflow-oriented software architectures, *in* D. C. Martin, ed., 'Proceedings of 4th IEEE/IFIP Conference on Software Architecture (WICSA2004)', Vol. P2172, IEEE Computer Society Press, pp. 211–220.

Norton, B., Foster, S. and Hughes, A. (2005), A compositional operational semantics for OWL-S, *in* M. Bravetti, L. Kloul and G. Zavattaro, eds, 'Proc. 2nd Intl. Workshop on Web Services and Formal Methods (WS-FM 2005)', Vol. 3670 of *Lecture Notes in Computer Science*, Springer, pp. 303–317.

Norton, B., Lüttgen, G. and Mendler, M. (2003), A compositional semantic theory for synchronous component-based design, *in* '14th Intl. Conference on Concurrency Theory (CONCUR '03)', Vol. 2761 of *Lecture Notes in Computer Science*, Springer, pp. 461–476.

Norton, B., Pedrinaci, C., Henocque, L. and Kleiner, M. (2007), '3-level behavioural models for Semantic Web Services', *Journal of Multi-agent And Grid Systems (MAGS)* .

Paige, R. and Tarjan, R. (1987), 'Three partition refinement algorithms', *SIAM Journal on Computing* **16**(6), 973–989.

Peyton-Jones, S., ed. (2003), *Haskell 98 Language and Libraries: the Revised Report*, Cambridge University Press.

Peyton-Jones, S., Gordon, A. and Finne, S. (1996), Concurrent Haskell, *in* 'Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages', St. Petersburg Beach, Florida, pp. 295–308.

Peyton-Jones, S., Vytiniotis, D., Weirich, S. and Washburn, G. (2006), Simple unification-based type inference for GADTs, *in* 'Proc. 11th Intl. SIGPLAN Conference on Functional Programming (ICFP 2006)', Vol. 41 of *ACM SIGPLAN Notices*, ACM Press, New York, NY, USA, pp. 50–61.

Plotkin, G. D. (1981), A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, University of Aarhus.

Prasad, K. V. S. (1991), A calculus of broadcasting systems, *in* 'Proc. of the Intl. Joint Conference on Theory and Practice of Software Development (TAPSOFT '91)', Vol. 493 of *Lecture Notes in Computer Science*, Springer, pp. 338–358.

Prasad, K. V. S. (1995), 'A calculus of broadcasting systems', *Science of Computer Programming* **25**(2-3), 285–327.

Rabin, M. and Scott, D. (1959), 'Finite automata and their decision problems', *IBM Journal of Research and Development* **3**(2), 114–125.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C. and Fensel, D. (2005), 'Web service modeling ontology', *Applied Ontology* **1**, 77–106.

Sangiorgi, D. and Milner, R. (1992), The problem of "weak bisimulation up to", *in* 'Proc. 3rd Intl. Conference on Concurrency Theory (CONCUR '92)', Vol. 630 of *LNCS*, Springer, pp. 32–46.

Sangiorgi, D. and Walker, D. (2001), *The $\pi$-calculus: A Theory of Mobile Processes*, Cambridge University Press.

Schrijvers, T., Peyton-Jones, S., Chakravarty, M. and Sulzmann, M. (2008), Type Checking with Open Type Functions, *in* 'Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)', Vol. 43 of *ACM SIGPLAN Notices*, ACM, New York, NY, USA, pp. 51–62.

Sheard, T. and Peyton-Jones, S. (2002), Template metaprogramming for Haskell, *in* M. M. T. Chakravarty, ed., 'ACM SIGPLAN Haskell Workshop 02', ACM, pp. 1–16.

Stollberg, M. and Norton, B. (2007), A refined goal model for Semantic Web Services, *in* 'Proc. of the 2nd Intl. Conference on Internet and Web Applications and Services (ICIW 2007)'.

Tidwell, D. (2000), 'Web Services – The Web's Next Revolution'.
**URL:** *http://www.ibm.com/developerWorks*

van der Aalst, W. (2005), 'Pi Calculus Versus Petri Nets: Let Us Eat Humble Pie Rather Than Further Inflate the Pi Hype', *Business Process Trends* .

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. and Barros, A. (2003), 'Workflow Patterns', *Distributed and Parallel Databases* **14**(1), 5–51.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. and Barros, A. (2006), Workflow Control-Flow Patterns: A Revised View, Technical Report BPM-06-22, BPMcenter.org.

van Glabbeek, R. J. (2001), The linear time-branching time spectrum I: The semantics of concrete, sequential processes, *in* J. A. Bergstra, A. Ponse and S. A. Smolka, eds, 'Handbook of Process Algebra', North-Holland, chapter 8, pp. 3–99.

von Neumann, J. (1951), The general and logical theory of automata, *in* L. A. Jeffress, ed., 'Cerebral Mechanisms in Behavior: The Hixon Symposium', John Wiley and Sons, pp. 1–41.

Wong, P. Y. and Gibbons, J. (2008), A Relative Timed Semantics for BPMN, *in* 'Proc. 7th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2008)', Vol. 229 of *ENTCS*.
**URL:** *http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/foclasa2008.pdf*

Yi, W. (1991), CCS + time = an interleaving model for real time systems, *in* 'Proceedings of the 18th international colloquium on Automata, languages and programming', Vol. 510 of *Lecture Notes in Computer Science*, Springer, pp. 217–228.

# Index