On Confidentiality and Formal Methods

Michael James Banks

Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science

March 2012

Abstract

The contemporary challenge of engineering verifiably secure software has motivated various techniques for measuring and regulating the flow of confidential data from systems to their users. Unfortunately, these techniques suffer from a lack of integration with modern formal methods for software development, which inhibits their application in practice.

This thesis proposes a novel approach for integrating information flow security concerns with formal methods. Working in the Unifying Theories of Programming (UTP), this thesis presents a generic framework for modelling interactions between users and systems. This framework can be applied to encode information flow about a system's activities to its users. It thereby allows confidentiality properties to be formalised in the UTP as upper bounds on information flow to users.

The main contribution of this thesis is a unified platform for designing software that is not only functionally correct, but also secure by design. This platform specialises the information flow encoding to the *Circus* formal method, making it possible to specify confidentiality properties within *Circus* processes. In this setting, conflicts between functionality and confidentiality are represented as miracles, rendering insecure functionality infeasible.

The platform provides techniques for verifying that functionality and confidentiality properties are mutually consistent. These techniques can be applied to develop a process through a series of feasibility-preserving refinement steps, to achieve a system implementation that does not leak secret information to untrusted users. These techniques are evaluated with a brief case study.

Contents

Acknowledgements

I could not have wished for a more patient and supportive supervisor than Jeremy Jacob. I am deeply grateful to him for giving me the freedom to seek out my own course of research, and for guiding me to surmount the obstacles along the way. Our discussions have challenged me to craft my ideas with logical rigour and to present those ideas precisely and fluently, in speech and in writing.

I have been fortunate to find myself supported by such an intellectually stimulating research community in the Department of Computer Science. Special thanks are owed to all of my officemates through the years — especially Peter Booth, Matthew Naylor, Ankur Patel, Marco Perez, Chris Poskitt and Jason Reich — for their camaraderie, their enthusiasm and for encouraging me to keep going when all seemed lost.

The direction of my research has been influenced by engaging — and often enlightening — discussions with Ana Cavalcanti, Will Harwood, Colin Runciman, Jim Woodcock and Frank Zeyda. Thanks to James Wood (NHS CfH) and Jonathan Lewis (Phoenix Interactive) for taking the time to propose topics for case studies to me, which I regrettably was unable to carry out owing to time constraints.

I also owe thanks to the Engineering and Physical Sciences Research Council, for providing financial support for the duration of my doctoral studies; and to the Department of Computer Science, where I have spent a lively, productive and enriching seven years of study.

Declaration

Unless stated otherwise, all contributions of this thesis are my own original work. Parts of this thesis are based on published papers:

1. On Modelling User Observations in the UTP. In *3rd International Symposium on Unifying Theories of Programming (UTP 2010)*.

   Chapter 3 is an extended version of this paper.

2. Unifying Theories of Confidentiality. In *3rd International Symposium on Unifying Theories of Programming (UTP 2010)*.

   Chapter 4 incorporates some parts of this paper.

3. Specifying Confidentiality in *Circus*. In *FM 2011: Formal Methods*.

   Chapter 5 and Chapter 6 build on work presented in this paper. Some material is also reproduced in Chapter 2.

4. Confidentiality Annotations for *Circus*. In *Proceedings of the Fourth York Doctoral Symposium on Computer Science (YDS 2011)*.

   Some ideas in this paper are used in Chapter 4 and Chapter 5. The case study in Chapter 7 also extends work in this paper.

The List of References contains full details of these papers.

1 Introduction

1.1 Confidentiality

Any organisation which handles private or sensitive information assets is obliged to protect the *confidentiality* of those assets. The U.S. National Institute of Standards and Technology defines confidentiality as:

> *"Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information."* (NIST, 2006)

The choice of security mechanisms for protecting confidential data should be proportionate to the sensitivity of those data (Anderson, 2003). In particular, the cost of implementing security measures should be balanced against the risk of disclosing those data to unauthorised persons:

- For governmental or military establishments, the compromise of national secrets could have catastrophic consequences.
- Likewise, a business may possess valuable trade secrets, where unauthorised exposure may result in financial or legal repercussions.
- The revelation of sensitive data about individuals — such as their banking, employment or medical records — may compromise their privacy and facilitate identity theft or blackmail against them.

In practice, software systems which store confidential data are commonly guarded with firewalls, access control and encryption. Each of these measures are important, but they do not guarantee security: all too often, a determined adversary can bypass them by identifying and exploiting flaws in their design and implementation. For this reason, it is widely accepted that bolting security mechanisms onto a software system is insufficient to obtain reliable assurances that the system is really secure (McLean, 1987).

With the advent of ubiquitous Internet connectivity, the demand for software systems that are *secure by design* is greater than ever before. In the words of Santen et al. (2002):

> *"The consent is growing that secure systems cannot be built by adding security features ex post to an existing implementation, but that "security-aware" engineering of systems and software must take security concerns into account, starting from requirements engineering through architectural and detailed design to coding, testing and deployment."*

This sentiment is echoed by Jones (2009), reporting the outcomes of an international meeting of security experts drawn from academia, commerce and government. These experts envisage:

> *"The development and procurement of software and systems which are resilient and sustainable by design, where requirements such as security and privacy are, as a matter of course, defined at project initiation and implemented and assured throughout in risk-based, whole-life processes."*

In circumstances where the confidentiality of information is critical, it is therefore imperative that software systems are developed to minimise the possibility of undesirable leakages of confidential information. These developments demand the application of precise and robust software engineering techniques.

1.2 Formal Methods and Security

An influential school of thought in software engineering holds that programs should be treated as mathematical objects (Hoare, 1969; Scott and

Strachey, 1971; Dijkstra, 1976; Gries, 1981; Morgan, 1994). This philosophy has motivated the production of a broad spectrum of deductive techniques, collectively known as *formal methods*, for constructing and analysing models of computer systems with logical rigour. The field of formal methods is surveyed by Wing (1990), Saiedian (1996) and Woodcock et al. (2009), among others.

Software engineers can apply formal methods to provide a high level of assurance that software products are *correct by construction*. These methods emphasise the value of creating abstract models to specify the desired characteristics of systems. These models can be subjected to rigorous consistency checks to determine whether they faithfully represent the customer's requirements for the system. A satisfactory model can then be developed — through a series of correctness-preserving *refinement steps* — into an implementation of the system (Wirth, 1971).

In the study and application of formal methods, correctness is predominantly interpreted in terms of *functionality properties*, which dictate the relation between the inputs and outputs of systems. Functionality places a lower bound on the information that a system reveals to its users. In contrast, confidentiality is a *non-functional* property: in fact, it imposes an upper bound on the disclosure of information to untrusted users. Since these bounds oppose each other, standard methods for developing systems to achieve functional correctness may fail to achieve a secure system (Jacob, 1992).

The advantages of applying formal methods in the development of security-critical systems are well documented by Wing (1998) and Ryan (2001). Indeed, it is widely recognised that formalisation is essential for designing trustworthy cryptographic protocols (Burrows et al., 1989). However, much of the theory associated with formal notions of confidentiality has been developed independently from formal methods and other software engineering practices. As a consequence, well-established formal development platforms — such as VDM (Jones, 1990) and the B-Method (Abrial, 1996; Schneider, 2001) — do not incorporate facilities for encoding confidentiality requirements within system designs. Hence, practitioners of formal methods need specialised knowledge to integrate confidentiality properties within the development of systems successfully.

1.3 A Platform for Secure Software Development

This thesis describes a formal platform for designing software systems that satisfy a joint specification of confidentiality and functionality properties. By promoting confidentiality to the status of a first-class citizen, we make it accessible to formal methods practitioners who do not possess specialised security knowledge.

At the heart of our platform is a generic framework for modelling systems with multiple users, presented in Chapter 3. Based on the *Unifying Theories of Programming* (UTP) (Hoare and He, 1998), this framework is suitable for modelling a user's interactions across different semantic models of systems. In Chapter 4, we study how the framework can be applied to formalise users' knowledge about a system's behaviour, providing a medium for the specification of confidentiality properties.

We elevate the framework to a fully fledged formal development platform by embedding it within *Circus*. The *Circus* notation, introduced by Woodcock and Cavalcanti (2001), combines the process algebra known as CSP (Hoare, 1985a; Roscoe, 1997; Schneider, 1999) and the state-based style of Z (Spivey, 1992; Woodcock and Davies, 1996) to provide facilities for modelling state-rich concurrent and reactive systems.

In Chapter 5, we extend the UTP theory of *Circus* processes to support new constructs for expressing confidentiality attributes over the state and behaviour of *Circus* processes. This construct integrates seamlessly with the *Circus* syntax; it can be mixed with regular *Circus* constructs to specify a wide range of confidentiality properties.

It is possible to specify a process where its functionality and confidentiality properties conflict with each other. Such conflicts between properties are undesirable, because no implementation can possibly satisfy a contradictory specification. Chapter 6 presents a technique for detecting these conflicts, which draws inspiration from Dijkstra's weakest precondition

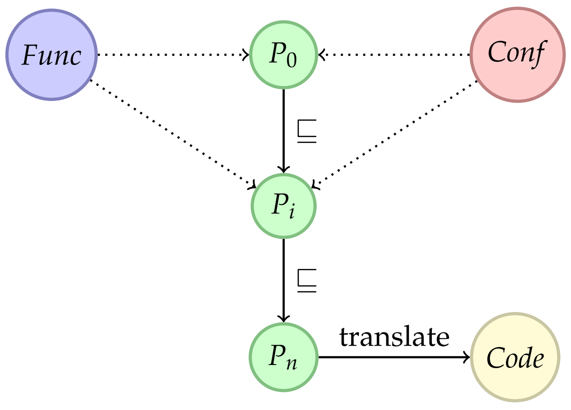

Figure 1.1: A depiction of a formal system development strategy incorporating both functionality and confidentiality requirements. Both functionality and confidentiality properties can be added to the development at each design step. These correctness properties are inherited under refinement, leading to a final design that may be implemented as code.

calculus. Moreover, this technique can be applied in combination with the *Circus* refinement calculus, to develop processes with the assurance that refinement steps do not induce conflicts between functionality and confidentiality.

To evaluate the effectiveness of these techniques, we present a case study development in Chapter 7.

1.4 Structure of this Thesis

Following this introduction, this thesis is structured as follows:

- Chapter 2 provides an overview of the UTP and *Circus*.
- Chapters 3 through 6 present the main contributions of the thesis.
- Chapter 7 delivers a case study that applies the techniques delivered in previous chapters.

- Chapter 8 summarises the contributions of the thesis and identifies topics for future work.
- The Appendix contains formal proofs of all theorems and lemmas given in the thesis.

This thesis is written with a formal methods audience in mind, owing to the nature of the material it covers. In particular, the reader should be familiar with the first-order predicate calculus and the algebra of CSP.

A discussion of the ethical issues surrounding computer security and information confidentiality is beyond the scope of this thesis. The reader interested in these issues may wish to consult Stoll (1988) and Schneier (2004) for various perspectives on this topic.

2 Background

2.1 Unifying Theories of Programming

This chapter provides a whistle-stop tour of Hoare and He's Unifying Theories of Programming (UTP). The UTP framework imparts a uniform semantics to a range of programming paradigms, spanning imperative, object-oriented and functional languages, across sequential and concurrent models of computation in timed and untimed domains.

The UTP strives for simplification: it achieves a *separation of concerns* by building semantic models of programming on top of more elementary programming models. The power of this modular approach is its ability to reveal connections between different programming paradigms, enabling results from one paradigm to be translated to other paradigms.

This chapter does not cover all aspects of the UTP, but focuses only on aspects that are relevant to this thesis. The reader who is familiar with the UTP may wish to proceed directly to Chapter 3. Likewise, the reader who desires a more comprehensive account of the UTP is directed to:

- the tutorial by Woodcock and Cavalcanti (2004), which focuses on the UTP theory of designs;

- the tutorial by Cavalcanti and Woodcock (2006), which constructs a UTP semantics for CSP processes; and

- the textbook by Hoare and He (1998), which is the definitive reference work for UTP.

2.2 Programs as Predicates

The semantic foundation of the UTP is the first-order predicate calculus, augmented with fixed point constructs from second-order logic.

Following the tradition of Hehner (1984a,b), *programs are predicates* in the UTP, and there is no distinction between programs and specifications at the semantic level. In the words of Hoare (1984b):

> *"A computer program is identified with the strongest predicate describing every relevant observation that can be made of the behaviour of a computer executing that program."*

In UTP parlance, a *theory* is a model of a particular programming paradigm. A UTP theory is composed of three ingredients:

- an *alphabet*, which is a set of variable names denoting the attributes of the paradigm that can be observed by an external entity;
- a *signature*, which is the set of programming language constructs intrinsic to the paradigm.
- a collection of *healthiness conditions*, which define the space of programs that fit within the paradigm.

In the UTP, healthiness conditions are typically defined as monotonic[1] idempotent predicate transformers. Given a healthiness condition $\mathbf{F}$, a predicate P is said to be $\mathbf{F}$-healthy if and only if $P = \mathbf{F}(P)$.

2.3 Relations

The most basic UTP theory is the alphabetised predicate calculus, which has no alphabet restrictions or healthiness conditions.

The theory of relations is slightly more specialised. It requires the alphabet of a relation to consist of only:

[1] $\mathbf{F}$ is monotonic if and only if $P_1 \sqsubseteq P_2$ implies $\mathbf{F}\,(P_1) \sqsubseteq \mathbf{F}\,(P_2)$, for all P_1 and P_2.

- undecorated variables ($a, b, c, \ldots$), modelling an observation of the program at the start of its execution; and
- primed variables ($a', b', c', \ldots$), modelling an observation of the program at a later stage of its execution.

Hereafter, we write x to denote the list of all undecorated variables in a relation's alphabet, and x' to denote all primed variables in the alphabet. The alphabet is *homogeneous* if the names of its primed variables match the names of its undecorated variables.

A relation models all observable behaviours of a program by mapping initial states (over x) to final states (over x').

Example 2.1. The relation $F2C \triangleq c' = (f - 32) \times 5/9$ models a program that converts temperature measurements from the Fahrenheit scale (f) to the Celsius scale (c'). By renaming c' and f to c and f' respectively:

$$C2F \quad \triangleq \quad F2C[c,f'/c',f] \quad = \quad f' = (c \times 9/5) + 32$$

we derive a program that performs the inverse conversion. $\Diamond$

2.3.1 Refinement

Program P_1 is refined by program P_2 — i.e. every observation of P_2 is an observation of P_1 — if and only if:

$$P_1 \sqsubseteq P_2 \quad \triangleq \quad [\, P_2 \Rightarrow P_1 \,]$$

where the square brackets $[\,\cdot\,]$ denote universal quantification over all variables in the alphabet (Dijkstra and Scholten, 1990).

Refinement succinctly captures the notion of program correctness: if $P_1 \sqsubseteq P_2$ holds, then P_2 correctly implements P_1. This notion of refinement is the same across all UTP theories.

The $\sqsubseteq$ relation is a partial order and induces a complete lattice over the space of relations. At the bottom of the lattice, the least refined

$$\begin{array}{rcll}
I\!I & \triangleq & x' = x & \text{(skip)} \\
a := E & \triangleq & a' = E \land u' = u & \text{(assignment)} \\
P_1 \, ; P_2 & \triangleq & \exists\, x_0 \bullet P_1[x_0/x'] \land P_2[x_0/x] & \text{(sequence)} \\
P_1 \sqcap P_2 & \triangleq & P_1 \lor P_2 & \text{(non-determinism)} \\
P_1 \lhd C \rhd P_2 & \triangleq & (C \land P_1) \lor (\neg\, C \land P_2) & \text{(conditional)} \\
\mu\, X \bullet F(X) & \triangleq & \bigsqcap \{X \mid F(X) \sqsubseteq X\} & \text{(recursion)}
\end{array}$$

Figure 2.1: The relational semantics of program constructs.

program **true** (sometimes called **abort**) has no constraints whatsoever on its behaviour. The lattice top is **false**, which admits no behaviour at all and cannot be implemented. Yet this "program" satisfies all specifications, which makes it a *miracle* (Nelson, 1989; Morgan, 1994). Miracles are conceptually very useful, because they give meaning to contradictory specifications.

Sometimes we write $P_1 \sqsubset P_2$ to specify P_2 is a *strict* refinement of P_1; i.e. P_1 and P_2 are not equivalent:

$$P_1 \sqsubset P_2 \quad \triangleq \quad P_1 \sqsubseteq P_2 \land P_2 \not\sqsubseteq P_1$$

2.3.2 Defining Program Constructs

The language constructs in the signature of the theory of relations are defined in Figure 2.1. Each construct is explained as follows:

- The skip statement, which does not alter the program state in any way, is modelled as the relational identity $I\!I$.
- The assignment of value E to a state variable a is defined by setting a' to E and keeping all other variables (denoted by u) constant.
- The sequential composition of two programs is just relational composition over their intermediate state.

- Non-deterministic choice between programs is their greatest lower bound.
- Conditional choice between programs is specified using a ternary operator, $\_ \lhd C \rhd \_$, where C is a relation.[2]
- A semantics for recursion is given by the weakest fixed point μF of a monotonic predicate transformer F. In particular, we have:

$$\mu F \quad = \quad \mu X \bullet F(X)$$

A corollary of the definition of μF is that an infinite recursion $\mu X \bullet X$ is equivalent to **abort**:

$$\mu X \bullet X \quad = \quad \bigsqcap \{X \mid X \sqsubseteq X\} \quad = \quad \mathbf{true}$$

2.4 Designs

Hoare and He (1998) identify an inconsistency between the theory of relations and our intuitive understanding of programs. Consider the relation $(\mu X \bullet X) \,; a := 1$. By calculation, we find that:

$$\begin{array}{lll}
(\mu X \bullet X) \,; a := 1 & & \\
= & \mathbf{true} \,; a := 1 & \text{[as above]} \\
= & \exists x_0 \bullet \mathbf{true}[x_0/x'] \wedge a := 1[x_0/x] & \text{[def ;]} \\
= & \exists x \bullet a := 1 & \text{[renaming]} \\
= & a' = 1 & \text{[def :=]}
\end{array}$$

This result implies that a program can break out of a non-terminating loop, which is inconsistent with the reality of programming. To resolve this inconsistency, a theory of programming needs to distinguish between terminating and non-terminating programs. This goal is met by the theory of *designs*.

[2]The notation $P_1 \lhd C \rhd P_2$ should be read "P_1 if C else P_2" (Hoare, 1985b).

A design models a program as a precondition and a postcondition. It also introduces two special Boolean observational variables into the alphabet:

- ok holds if the program has started properly; while
- ok' holds if the program has terminated.

The theory of designs is characterised by two idempotent monotonic healthiness conditions, **H1** and **H2**:

$$\begin{aligned} \mathbf{H1}(P) &\triangleq ok \Rightarrow P \\ \mathbf{H2}(P) &\triangleq P\,;((ok \Rightarrow ok') \wedge v' = v) \end{aligned}$$

where v denotes the list of state variables excluding ok.

H1 specifies that a design makes no predictions about its behaviour before it is started. **H2** specifies that a design cannot insist on non-termination by forcing ok' to be **false**.[3]

Provided P's alphabet includes ok and ok', **H1** and **H2** are commutative. Any relation that is **H1** and **H2**-healthy can be written in the form:

$$Pre \vdash Post \triangleq (ok \wedge Pre) \Rightarrow (ok' \wedge Post)$$

A design $Pre \vdash Post$ reflects a contract between the customer and the programmer: if the design is started in a state satisfying Pre, then it must terminate in a state such that $Post$ is satisfied. By this construction, refinement of a design corresponds to weakening its precondition and strengthening its postcondition (Hoare and He, 1998, Theorem 3.1.2).

Some program constructs in the theory of relations are not designs in their own right. These constructs need to be redefined as designs:

$$\begin{aligned} I\!I_D &\triangleq \mathbf{true} \vdash I\!I && \text{(design-skip)} \\ a := E &\triangleq \mathbf{true} \vdash a' = E \wedge u' = u && \text{(design-assignment)} \end{aligned}$$

[3]The definition of **H2** presented here is taken from Cavalcanti and Woodcock (2006). Its semantics is equivalent to Hoare and He's definition, but reformulated as an idempotent monotonic predicate transformer.

2.5 Reactive Processes

A reactive process communicates with its environment by engaging in a series of events. Therefore, a model of a reactive process needs to admit intermediate observations at points during its execution.

The theory of reactive processes is defined over an extended alphabet, featuring four pairs of distinguished observational variables. The Boolean variables *ok* and *wait* model the process status at the initial observation:

- $ok \wedge wait$: the process is waiting to start its execution;
- $ok \wedge \neg\, wait$: the process has started its execution properly;
- $\neg\, ok$: the process has not been started properly;

Likewise, the primed Boolean variables ok' and $wait'$ model the process status at subsequent observations:

- $ok' \wedge wait'$: the process has reached a stable intermediate point in its execution and is awaiting interaction with the environment;
- $ok' \wedge \neg\, wait'$: the process has terminated in a stable point.
- $\neg\, ok'$: the process has neither reached a stable intermediate point nor has terminated: it is said to have *diverged*.

The variables tr and tr' record the sequence of events observed by the environment. tr records all events prior to the process starting, while tr' records events that occur up to the point when the subsequent observation is made. Hence, the process trace is given by $tr' - tr$.

The ref and ref' variables record the events that may be refused by the process. They are unconstrained by the theory of reactive processes.

Reactive processes are characterised by three healthiness conditions:

$$\begin{array}{lcl}
\mathbf{R1}(P) & \triangleq & P \wedge tr \leq tr' \\
\mathbf{R2}(P) & \triangleq & P[\langle\rangle, tr' - tr / tr, tr'] \\
\mathbf{R3}(P) & \triangleq & (\mathit{II}_{rea} \lhd wait \rhd P)
\end{array}$$

where the reactive identity II_{rea} is defined as follows:

$$\mathit{II}_{rea} \quad \triangleq \quad \left(\begin{array}{ll} & \neg\, ok \wedge tr \leq tr' \\ \vee & ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge v' = v \end{array} \right)$$

A **R1**-healthy process can only extend the trace; in other words, it has no control over events prior to starting. A **R2**-healthy process is oblivious to events that occurred before it starts. A **R3**-healthy process has no effect on the observation until it commences execution.

Since **R1**, **R2** and **R3** are idempotent and commute with each other, they can be combined into a single healthiness condition **R**:

$$\mathbf{R}(P) \quad \triangleq \quad \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}(P)$$

2.6 Reactive Designs

The theories of designs and reactive processes each have valuable qualities. Designs are useful for modelling programs in terms of preconditions and postconditions, while reactivity enables the intermediate behaviour of programs to be captured. However, the healthiness conditions **H1** and **R1** do not commute with each other:

$$\begin{array}{rcl} \mathbf{H1} \circ \mathbf{R1}(P) & = & ok \Rightarrow (P \wedge tr \leq tr') \\ & \neq & (ok \Rightarrow P) \wedge tr \leq tr' \\ & = & \mathbf{R1} \circ \mathbf{H1}(P) \end{array}$$

Intuitively, a **H1**-healthy relation allows any observation whatsoever when *ok* is **false**, whereas a **R1**-healthy relation enforces $tr \leq tr'$ in all circumstances. This non-commutativity between **H1** and **R1** implies that no design is a reactive process, and vice versa.

The space of *reactive designs* is a sub-space of the reactive processes, which is derived by applying **R** to the space of designs.[4] Reactive designs

[4]Contrary to their name, reactive designs are *not* designs, since they are not **H1**-healthy.

have the shape:

$$P \quad = \quad \mathbf{R}\,(Pre \vdash Post)$$

where *Pre* specifies the conditions in which *P* does not diverge; and *Post* specifies all possible non-diverging outcomes of executing *P* in any state satisfying *Pre*. In this way, reactive designs facilitate the definition of process semantics in a design-like manner.

In particular, the bottom of the lattice of reactive designs is:

$$\mathbf{R}\,(\mathbf{false} \vdash \mathbf{true}) \quad = \quad \mathbf{R}(\mathbf{true})$$

which diverges whenever it is permitted to start execution. The top of the lattice of reactive designs is:

$$\mathbf{R}\,(\mathbf{true} \vdash \mathbf{false}) \quad = \quad \mathbf{R}(\neg\, ok)$$

which would achieve the impossible (postcondition **false**) if it could ever be executed. This process is the *reactive design miracle* (Woodcock, 2010).

The theory of reactive designs is extended by Hoare and He (1998) and Cavalcanti and Woodcock (2006) to give a UTP semantics to CSP processes. Two healthiness conditions characterise the sub-space of reactive designs corresponding to CSP processes:

$$\begin{aligned}
\mathbf{CSP1}\,(P) \quad &\triangleq \quad P \vee (\neg\, ok \wedge tr \leq tr') \\
\mathbf{CSP2}\,(P) \quad &\triangleq \quad P\,;\left(\begin{array}{ll} (ok \Rightarrow ok') & \wedge\ wait' = wait \wedge tr' = tr \\ & \wedge\ ref' = ref \wedge v' = v \end{array}\right)
\end{aligned}$$

CSP1 specifies that, if a process diverges, then only the extension of the trace is guaranteed. **CSP2** recasts **H2**: it specifies a process may not require non-termination.

Every CSP process is expressible in the form of a reactive design.

Theorem 2.2 (from Hoare and He (1998))**.** For each CSP process P:

$$P \quad = \quad \mathbf{R}\left(\neg\, P^f_f \vdash P^t_f\right)$$

where P^b_c abbreviates $P[b, c/ok', wait]$.

The theory of CSP processes serves as a baseline for the semantics of *Circus* actions. For brevity, we skip over this theory and jump straight to the closely related theory of *Circus* actions in Section 2.8.

2.7 *Circus*

Circus is a formal specification language which fuses the CSP process algebra with a notion of state, to achieve a cohesive platform for modelling state-rich concurrent and reactive systems.

The specification constructs of *Circus* are called *actions*. *Circus* actions can be grouped into three classes:

- CSP constructs, modelling interaction with the environment over a collection of named channels;
- Z schema expressions and specification statements (Morgan, 1994), representing state operations; and
- guarded commands (Dijkstra, 1976), orchestrating control flow according to the state.

A *Circus* process specifies an internal state (hidden from the environment), a state invariant and a collection of named actions. The behaviour of a *Circus* process is defined by a distinguished nameless main action, which follows the declarations of the other actions.

Example 2.3. Figure 2.2 presents a *Circus* process modelling a memory cell that stores an integer value from the *in* channel. The cell can be switched between two modes. In public mode, the value currently stored in the cell is broadcast on the *out* channel; whereas in private mode, an arbitrary value is broadcast on *out*. ◊

$MODE == \{PUB, PRV\}$
channel on, off
channel $in, out : \mathbb{N}$
process $Cell \triangleq$ **begin**
  state $Mem \triangleq [val : \mathbb{N}, m : MODE]$
  $Init \triangleq [Mem' \,|\, val' = 0 \land m' = PUB]$
  $Read \triangleq \left(\begin{array}{ll} & m = PUB \,\&\, out!val \rightarrow Skip \\ \Box & m = PRV \,\&\, \sqcap n : \mathbb{N} \bullet out!n \rightarrow Skip \end{array} \right)$
  $Write \triangleq in?n \rightarrow val := n?$
  $Switch \triangleq (on \rightarrow m := PRV) \Box (off \rightarrow m := PUB)$
  $\bullet\ Init\,;\ \mu X \bullet (Read \Box Write \Box Switch)\,; X$
end

Figure 2.2: An example *Circus* process: a memory cell.

2.8 *Circus* Actions

A UTP theory of *Circus* actions is defined by Oliveira et al. (2009), based on work by Woodcock and Cavalcanti (2002) and Oliveira (2005). This theory enriches the theory of CSP processes with facilities for accessing and manipulating state variables.

The primitive actions of *Circus* are inherited from CSP. The action *Skip* terminates immediately, while *Stop* deadlocks and *Chaos* diverges:[5]

$$\begin{array}{rcl} Skip & \triangleq & \mathbf{R}\left(\mathbf{true} \vdash tr' = tr \land \neg\, wait' \land v' = v\right) \\ Stop & \triangleq & \mathbf{R}\left(\mathbf{true} \vdash tr' = tr \land wait'\right) \\ Chaos & \triangleq & \mathbf{R}\left(\mathbf{false} \vdash \mathbf{true}\right) \end{array}$$

A specification statement (Morgan, 1994) is a construct for specifying operations on state, in terms of a precondition, a postcondition and a *frame* w of variables, whose values the operation may change. The *Circus*

[5]Notice the *Chaos* here is that of Hoare (1985a), rather than Roscoe (1997).

semantics of specification statements is:

$$w : [Pre, Post] \triangleq \mathbf{R}\left(Pre \vdash Post \land \neg\, wait' \land tr' = tr \land u' = u\right)$$

where u denotes all state variables outside the frame w. Normalised Z schemata can be translated to specification statements using the basic conversion rules given by Cavalcanti and Woodcock (1998).

Following Morgan (1994), assignments, assumptions and coercions[6] are defined in terms of specification statements:

$$\begin{aligned} a := E &\triangleq a : \left[\mathbf{true}, a' = E\right] \\ \{\, C \,\} &\triangleq \; : [C, \mathbf{true}] \\ [\, C \,] &\triangleq \; : [\mathbf{true}, C] \end{aligned}$$

It follows from the semantics of the specification statement that:

$$\begin{aligned} a := E &= \mathbf{R}\left(\mathbf{true} \vdash a' = E \land \neg\, wait' \land tr' = tr \land u' = u\right) \\ \{\, C \,\} &= \mathbf{R}\left(C \vdash \neg\, wait' \land tr' = tr \land v' = v\right) \\ [\, C \,] &= \mathbf{R}\left(\mathbf{true} \vdash C \land \neg\, wait' \land tr' = tr \land v' = v\right) \end{aligned}$$

Assumptions and coercions are annotation constructs, specifying conditions that are expected to be satisfied by the process state. The assumption $\{\, C \,\}$ behaves as *Skip* if the process state satisfies C, and diverges otherwise. The coercion $[\, C \,]$ *forces* the process state to satisfy C.

Example 2.4. The coercion $[\, C \,]$ behaves as *Skip* if C holds; for instance:

$$\begin{aligned} h := 1 ; [\, h = 1 \,] &= h := 1 ; [\, \mathbf{true} \,] &= h := 1 \\ h := 0 ; [\, h = 1 \,] &= h := 0 ; [\, \mathbf{false} \,] &= \mathbf{R}(\neg\, ok) \end{aligned}$$

but behaves as the reactive design miracle otherwise. $\Diamond$

[6]The terms "assumption" and "coercion" are adopted by Morgan (1994). In the design space, $\{\, \cdot \,\}$ and $[\, \cdot \,]$ are called "assertions" and "assumptions" respectively by Hoare and He (1998). To avoid confusion, we use Morgan's terms throughout this thesis.

As Example 2.3 shows, *Circus* actions can be composed using the CSP operators. As before, sequential composition ($;$) is relational composition, internal choice ($\sqcap$) is disjunction and recursion is defined in terms of the weakest fixed point operator. Other CSP operators are given a UTP semantics by Oliveira et al. (2009). We now outline the semantics of three *Circus* operators.

Prefixing A prefixing $c.e \rightarrow Skip$ never diverges and awaits synchronisation on channel c. Once it has made that synchronisation, it terminates:

$$c.e \rightarrow Skip \triangleq \mathbf{R}\left(\mathbf{true} \vdash \left(\begin{array}{c} tr' = tr \wedge (c,e) \notin ref' \\ \lhd\ wait'\ \rhd \\ tr' = tr \frown \langle (c,e) \rangle \end{array}\right) \wedge v' = v\right)$$

Here, an event is represented in the trace as a pair of a channel name and the value transmitted on that channel. The semantics of a CSP prefixing $c \rightarrow Skip$ (where channel c does not communicate values) is defined as $c.Sync \rightarrow Skip$, where $Sync$ denotes a generic synchronisation event.

The prefixing $c.e \rightarrow A$ can be rewritten as $c.e \rightarrow Skip\,;A$.

External choice An external choice $A_1 \Box A_2$ offers the environment the ability to select between the events offered by A_1 and A_2. Its semantics is:

$$A_1 \Box A_2 \quad \triangleq \quad \mathbf{R}\left(\neg\, {A_1}_f^f \wedge \neg\, {A_2}_f^f \vdash \left(\begin{array}{c} {A_1}_f^t \wedge {A_2}_f^t \\ \lhd\ tr' = tr \wedge wait'\ \rhd \\ {A_1}_f^t \vee {A_2}_f^t \end{array}\right)\right)$$

This choice does not diverge provided that neither A_1 nor A_2 diverge. Provided that A_1 and A_2 do not terminate instantaneously, then the environment can select from the initial events offered by either action. The prefixing operator specifies its refusals negatively, so the refusal set of ${A_1}_f^t \wedge {A_2}_f^t$ cannot contain any event that is accepted by either A_1 or A_2.

Once an event is performed, the choice between A_1 and A_2 is resolved.

Guards A guarded command $g \mathbin{\&} A$ behaves as A if g holds, and as *Stop* otherwise:

$$g \mathbin{\&} A \quad \triangleq \quad \mathbf{R}\left((g \Rightarrow \neg A^f_f) \vdash (A^t_f \lhd g \rhd tr' = tr \land wait')\right)$$

Definitions of the remaining *Circus* operators — including parallel composition and hiding — are given by Oliveira et al. (2009). Their details are not necessary to follow the rest of this thesis.

The UTP semantics of *Circus* actions facilitates the definition of a refinement calculus (Oliveira, 2005), featuring a wide variety of laws proved directly from the semantics. This refinement calculus enables software engineers to transform abstract process designs into concrete implementations, without exposure to the underlying *Circus* semantics.

2.9 The *Circus* Family

Building on the reference *Circus* semantics, several versions of *Circus* have been defined to address specialised domains in software engineering. These domains include (but are not limited to) object-oriented programming (Cavalcanti et al., 2005), real-time specification (Wei et al., 2010) and synchronously-clocked systems (Gancarski and Butterfield, 2009).

In the body of this thesis, we describe how notions of information confidentiality can be captured in the UTP and, from there, embedded into the reference *Circus* semantics. We speculate that our work could be replicated across the other versions of *Circus*, giving rise to languages in each domain with a confidentiality-sensitive semantics.

3 Multi-User Systems

3.1 Introduction

This chapter presents techniques for formally reasoning about shared computing systems that offer services to multiple end-users. This class of *multi-user* systems encompasses a wide variety of software products, ranging from operating systems and database servers to telecommunications networks and cloud computing facilities.

The word "user" has many connotations in computing but, in this chapter, we use the term to mean any entity that can interact with a computer system or be influenced by its behaviour. Thus, a user may denote a client that issues requests to the system and receives timely responses; an administrator responsible for monitoring the system; or an entity that observes the system's behaviour in a passive way.

3.1.1 Motivation

When designing a multi-user system, it is important to characterise the relationship between the system and its users, to determine whether the system will satisfy the expectations of its users.

A multi-user system may offer different services to different users. Hence, it is usual for the system to supply each user with its own dedicated interface. These interfaces impose structure upon the system's environment, enabling the system to distinguish between its users.

Many formal methods for software development treat the environment of a system as a single uniform entity. In these formalisms, the users of a system are conventionally modelled as components of the specification of the system itself. For instance, in state-based formalisms such as B and Z, a user's interface to a system is typically defined as a collection of operations that are considered to be accessible to the user. Likewise, when working with process algebras such as CSP, it is usual to model users as individual processes that synchronise with a process representing the operations offered by the system. Using a tool such as FDR (Roscoe, 1994), this approach can be used to verify that a process delivers the functionality expected by its users, by analysing the synchronisations between these processes for the absence of deadlock or divergence.

Modelling users as implicit components of a system works well for analysing the functionality delivered by the system to its environment. However, by failing to distinguish the users from the environment, this approach does not distinguish the interactions of users from the behaviours of the system. This shortcoming is inconvenient when dealing with correctness properties — such as confidentiality properties — where the relation between behaviours and interactions has intrinsic significance.

3.1.2 Contributions of this Chapter

In this chapter, we investigate an alternative approach for modelling users of systems. The core of this approach is outlined in Section 3.2: we specify a user's interface separately from a system, in terms of the elements of a system's behaviour that are visible to that user. In this way, we become able to analyse the interactions that users may perform with the system, and thence verify that a system design upholds the users' expectations about their interactions.

Section 3.3 presents a method for calculating the space of interactions that a user may perform with a given system. We explore how this approach can be applied to the theories of designs in Section 3.4 and reactive processes in Section 3.5.

We also investigate how knowledge of the interfaces of users can be used to justify certain kinds of functional design decisions in system developments. In Section 3.6, we outline a weakened notion of refinement applicable to multi-user systems that is framed by the observational abilities of isolated users or groups of collaborating users.

The work in this chapter constitutes the foundations of later chapters of this thesis, which build on the model of users to formalise the notion of information flow in the UTP and in *Circus*.

3.2 Modelling Users

Recall from Chapter 2 that, in the UTP theory of relations, a system is modelled in terms of all possible observations that an external entity could make of the system.

When reasoning about multi-user systems modelled as relations, we draw a distinction between two kinds of observations:

Behaviours are observations of a system made by the global environment.

Interactions are local observations of a system made by a single user.[1]

We assume that users cannot monitor the system directly, but can only observe the system through their respective interfaces. We capture this assumption by introducing a secondary observation space to model interactions separately from behaviours. To do this, we introduce two new lists of observational variables, x_V and x'_V, to encode interactions as alphabetised predicates:

- the x_V variables model the state of the user's interface before an interaction takes place; whereas
- the x'_V variables model the state of the interface after the interaction takes place.

[1]Although the term "interaction" suggests that users actively engage with the system and influence its behaviour, this need not be so.

3.2.1 Views

In general, a user's interface to a system relays only partial information about the system's behaviour to that user; after all, users need not be concerned with every detail of the system's behaviour. Hence, a user's interaction is a *projection* of the behaviour through that user's interface.

A *view* is a predicate expressing a relation between observation spaces. More precisely, we define a view as a relation from the space of behaviours to the space of interactions. Hence, a view formalises a user's interface to a system, because it determines which aspects of the system's behaviour show through the user's interface.

Example 3.1. Suppose that x and y are integer-valued observational variables. Consider the following views:

$$\begin{aligned} V1 &\triangleq a_1 = x \wedge b_1 = y \\ V2 &\triangleq c_2 = \max(x, y) \\ V3 &\triangleq d_3 = x \wedge (e_3 = 0 \lhd x < y \rhd e_3 = 1) \end{aligned}$$

$V1$ provides the exact values of x and y, since there is a one-to-one correspondence between these variables and a_1 and b_1. $V2$ provides the value of the larger of x and y, but no indication of whether $x < y$, $x = y$ or $x > y$. $V3$ provides the value of x, but offers only partial information about y's value in relation to x's value. ◊

When dealing with multiple views, each representing a different user, we require that each view's alphabet of interaction variables is disjoint from those of the other views. (All of the views listed in Example 3.1 are pairwise disjoint.)

Definition 3.2 (Disjoint views)**.** A pair of views V_1 and V_2 are *disjoint* if and only if their alphabets have no interaction variables in common:

$$\alpha V_1 \cap \alpha V_2 \quad \subseteq \quad x \cup x'$$

The abstractness of views allows us to describe the observational abilities of users in exact relation to the behaviour of systems. We note in passing that a view's structure should conform to the UTP theory in which it is applied; we return to this point in Sections 3.4 and 3.5.

3.2.2 Healthiness Conditions for Views

To ensure that all views represent a viable mapping between behaviours and interactions, it is necessary to impose some healthiness conditions on the structure of views.

Our first healthiness condition ensures that views do not restrict the behaviour of systems in any way.

Definition 3.3 (VH1). A view V is **VH1**-healthy if and only if V maps each behaviour to *at least one* interaction:

$$\mathbf{VH1}(V) \quad \triangleq \quad (\exists\, x_V, x'_V \bullet V) \Rightarrow V$$

VH1 may be more comprehensible when presented in a form that asserts the **VH1**-healthiness of a view, as given by Lemma 3.4.

Lemma 3.4 (VH1 reformulated).

$$\mathbf{VH1}\,(V) = V \quad \text{if and only if} \quad \left[\exists\, x_V, x'_V \bullet V\right]$$

A **VH1**-healthy view is a total relation from behaviours to interactions. If V does not map a behaviour ϕ to at least one interaction, then $\mathbf{VH1}(V)$ maps ϕ to *every* interaction. Hence, **VH1**-healthy views do not restrict the behaviour of systems in any way.

Remark 3.5. An alternative formulation of **VH1** would transform views by mapping behaviours associated with zero interactions to a nominated "null" interaction. However, our definition of **VH1** is simpler, because it avoids the need to specify what that null interaction should be.

Example 3.6. Predicates such as $a = b \wedge c_V > 0$ and **false** are not **VH1**-healthy, because they restrict the behaviour space. For instance:

$$\begin{array}{lll}
& \textbf{VH1}(a = b \wedge c_V > 0) & \\
= & (\exists\, x_V, x'_V \bullet a = b \wedge c_V > 0) \Rightarrow (a = b \wedge c_V > 0) & [\text{def } \textbf{VH1}] \\
= & a = b \Rightarrow (a = b \wedge c_V > 0) & [\text{pred calc}] \\
= & a = b \Rightarrow c_V > 0 & [\text{prop calc}]
\end{array}$$

which does not equal $a = b \wedge c_V > 0$. ◊

We could write a "precognitive" **VH1**-healthy predicate — for example, $a_V = a'$ — where the initial variables of an interaction depend upon the values of the final behavioural variables. Such a predicate does not correspond to any user interface in reality, because no interface can predict the behaviour of an unspecified system before that system starts its execution. We exclude these unrealistic predicates from the space of views by defining a second healthiness condition **VH2**.

Definition 3.7 (VH2 healthiness condition). A view V is **VH2**-healthy if and only if its alphabet contains only unprimed variables:

$$\textbf{VH2}(V) \quad \triangleq \quad \exists\, x', x'_V \bullet V$$

The definition of a healthiness condition to rule out precognitive predicates over the initial and final observational space would be complicated. With **VH2**, we cut this Gordian knot by assuming that a user's interface does not change during the system's execution.

By assuming that a user's interface is constant, we are free to encode views over the unprimed variables alone. The notation:

$$\Delta V \quad \triangleq \quad V \wedge V'$$

denotes the interface corresponding to the **VH2**-healthy view V, where V' denotes V with all unprimed variables renamed with a prime.

We now present some useful properties of **VH1** and **VH2**. Both **VH1** and **VH2** are idempotent.

Law 3.8 (VH1 idempotent). $\mathsf{VH1} \circ \mathsf{VH1}(V) = \mathsf{VH1}(V)$

Law 3.9 (VH2 idempotent). $\mathsf{VH2} \circ \mathsf{VH2}(V) = \mathsf{VH2}(V)$

Law 3.9 is a trivial consequence of the definition of **VH2**. Unfortunately, **VH1** and **VH2** do not commute with each other.

Law 3.10 (VH1-VH2 non-commutative). $\mathsf{VH2} \circ \mathsf{VH1}(V) \sqsubseteq \mathsf{VH1} \circ \mathsf{VH2}(V)$

We say that a view that is both **VH1**-healthy and **VH2**-healthy is **VH**-healthy. In light of Law 3.10, we define **VH** as the stronger combination of **VH1** and **VH2**.

Definition 3.11 (VH healthiness condition).

$$\mathsf{VH}(V) \quad \triangleq \quad \mathsf{VH1} \circ \mathsf{VH2}(V)$$

Corollary 3.12 (VH idempotent). By Law 3.8 and the definition of **VH**:

$$\mathsf{VH1} \circ \mathsf{VH}(V) \quad = \quad \mathsf{VH}(V)$$

Likewise, since the x' and x'_V variables are free in **VH**-healthy views:

$$\mathsf{VH2} \circ \mathsf{VH}(V) \quad = \quad \mathsf{VH}(V)$$

It follows from these results that **VH** is idempotent.

All of the views in Example 3.1 are **VH**-healthy, as are all the views that we consider in the following sections.

3.2.3 Conjoining Views

Given two disjoint views V_1 and V_2, the view $V_1 \wedge V_2$ models the combined interactions of a pair of users. This view represents those users sharing knowledge of their interactions with the system with each other: conjoining V_1 and V_2 ensures that pairs of V_1-interactions and V_2-interactions are consistent with the same behaviour.

Lemma 3.13 (Conjunction preserves **VH**)**.** If V_1 and V_2 are disjoint **VH**-healthy views, then $V_1 \wedge V_2$ is also **VH**-healthy.

3.2.4 Functional Views

While **VH1** insists that a view describes a total mapping from behaviours to interactions, this mapping need not be functional. A non-functional view would correspond to a user's interface that can yield *different* interactions for the *same* behaviour.

Example 3.14. A lossy communications channel can be modelled as a **VH**-healthy view. For instance, the view $a_V \in [a-k, a+k]$ could represent a signal that fluctuates by up to k units. This view could serve as a crude model of a physical wire carrying the signal. $\Diamond$

However, the interfaces to a system are typically designed to yield a deterministic projection of the system's behaviour, at a reasonable level of abstraction.[2] We define the space of these *functional* views in terms of a new healthiness condition **VH3**.

Definition 3.15 (**VH3** healthiness condition)**.** A view V is **VH3**-healthy if and only if V maps each behaviour to *exactly one* interaction or *indefinitely many* interactions:

$$\mathbf{VH3}(V) \quad \triangleq \quad (\exists_1\, x_V, x'_V \bullet V) \Rightarrow V$$

Every **VH3**-healthy view is also **VH1**-healthy.

Example 3.16. Provided $k > 0$, the **VH1**-healthy view $a_V \in [a-k, a+k]$ from Example 3.14 is not **VH3**-healthy:

$\mathbf{VH3}(a_V \in [a-k, a+k])$

[2] Quantum mechanics suggests that no physical interface can be truly deterministic at the lowest level, but this is generally not a concern for system designers in practice!

$$\begin{array}{cll}
= & (\exists_1 x_V, x'_V \bullet a_V \in [a-k, a+k]) \Rightarrow a_V \in [a-k, a+k] & [\text{def } \mathbf{VH3}] \\
= & \mathbf{false} \Rightarrow a_V \in [a-k, a+k] & [\text{not deterministic}] \\
= & \mathbf{true} & [\text{prop calc}]
\end{array}$$

The resulting view **true** models a user whose interface is disconnected from the system, so the user's interaction bears no resemblance whatsoever to the system's behaviour. ◊

From here onwards, the views we consider in examples are **VH3**-healthy, to aid intuition. However, unlike **VH1** and **VH2**, the special functional property of **VH3** is not essential to justify the theorems and lemmas presented in the following sections. For this reason, we do not restrict the framework to **VH3**-healthy views; rather, we appeal to **VH3** only when it is needed.

3.3 Calculating Interactions

In this section, we formalise procedures for translating a model of a system's behaviour into a complementary model representing a user's interactions with that system, and vice versa.

3.3.1 Localisation

Given a view V and (a behavioural model of) a system P, the predicate $\mathbf{L}(V, P)$ is the *image* of P projected through V. In other words, this predicate encodes all interactions with P that can be made through V.

Definition 3.17 (**L** predicate transformer)**.**

$$\mathbf{L}(V, P) \quad \triangleq \quad \exists x, x' \bullet \Delta V \wedge P$$

Example 3.18. Consider the condition Ex:

$$Ex \quad \triangleq \quad x \geq 0 \wedge y \geq 0 \wedge x + y = 10$$

The images of Ex corresponding to each view listed in Example 3.1 are:

$$\begin{aligned}
&\mathbf{L}(V1, Ex) = a_1 \geq 0 \wedge b_1 \geq 0 \wedge a_1 + b_1 = 10\\
&\mathbf{L}(V2, Ex) = c_2 \geq 5 \wedge c_2 \leq 10\\
&\mathbf{L}(V3, Ex) = (d_3 \geq 0 \wedge d_3 < 5 \wedge e_3 = 0) \vee (d_3 \geq 5 \wedge d_3 \leq 10 \wedge e_3 = 1)
\end{aligned}$$

Notice that **L** hides the behavioural variables of Ex, modelling a user's inability to observe those variables directly. $\Diamond$

We now describe three properties of **L** that we use later in this chapter.

Distributivity Each interaction with a system that behaves either as P_1 or P_2 is an interaction either with P_1 or with P_2.

Law 3.19 (**L** (V) is disjunctive)**.** $\mathbf{L}(V, P_1 \vee P_2) = \mathbf{L}(V, P_1) \vee \mathbf{L}(V, P_2)$

Monotonicity If $P_1 \sqsubseteq P_2$, then every V-interaction with P_2 (where V is **VH**-healthy) will also be a V-interaction with P_1.

Law 3.20 (**L** (V) is monotonic)**.** Provided P_1 and P_2 share an alphabet:

$$P_1 \sqsubseteq P_2 \quad \text{implies} \quad \mathbf{L}(V, P_1) \sqsubseteq \mathbf{L}(V, P_2)$$

Splitting views If a view V is divided into two disjoint views V_1 and V_2, such that $V = (V_1 \wedge V_2)$, then $\mathbf{L}(V_1, P) \wedge \mathbf{L}(V_2, P)$ is potentially weaker than $\mathbf{L}(V, P)$ itself. Intuitively, this is because the interactions given by $\mathbf{L}(V_1, P)$ and $\mathbf{L}(V_2, P)$ are not synchronised with each other. We generalise this result to an arbitrary number of views in Law 3.21.

Law 3.21 (Splitting views weakens projections)**.** Provided $V_1, \ldots, V_n$ are pairwise disjoint views:

$$\bigwedge_{i \in 1..n} \bullet \mathbf{L}(V_i, P) \quad \sqsubseteq \quad \mathbf{L}\left(\left(\bigwedge_{i \in 1..n} V_i\right), P\right)$$

3.3.2 Globalisation

Given an image U of interactions formed by projecting a system P through view V, some knowledge of the structure of P can be recovered by substituting V back into U. This procedure is formalised by the **G** predicate transformer.

Definition 3.22 (**G** predicate transformer)**.**

$$\mathbf{G}(V, U) \quad \triangleq \quad \forall x_V, x'_V \bullet \Delta V \Rightarrow U$$

The relation $\mathbf{G}(V, U)$ is the weakest specification of a system such that each behaviour of $\mathbf{G}(V, U)$ projects through V to an interaction of U.

Example 3.23. Continuing from Example 3.18, the conditions recovered by applying **G** to each **L**-projection of Ex are:

$$\begin{aligned}
\mathbf{G}(V1, \mathbf{L}(V1, Ex)) &= x \geq 0 \land y \geq 0 \land x + y = 10 \\
\mathbf{G}(V2, \mathbf{L}(V2, Ex)) &= \max(x, y) \geq 5 \land \max(x, y) \leq 10 \\
\mathbf{G}(V3, \mathbf{L}(V3, Ex)) &= (x \geq 0 \land x < 5) \lhd x < y \rhd (x \geq 5 \land x \leq 10)
\end{aligned}$$

Observe that $\mathbf{G}(V1, \mathbf{L}(V1, Ex)) = Ex$, because view $V1$ preserves the values of x and y in a_1 and b_1. However, both $\mathbf{G}(V2, \mathbf{L}(V2, Ex))$ and $\mathbf{G}(V3, \mathbf{L}(V3, Ex))$ are weaker than Ex, because information about the values of x and y is discarded by applying $\mathbf{L}(V2)$ or $\mathbf{L}(V3)$ to Ex. $\Diamond$

The predicate $\mathbf{G}(V, U)$ features all behaviours that V *only* maps to interactions in U. However, if V is **VH3**-healthy, then V maps each behaviour to exactly one interaction. This property of V enables the definition of $\mathbf{G}(V, U)$ to be reshaped, as Lemma 3.24 shows.

Lemma 3.24 (**G** and **VH3**)**.** Provided V is **VH3**-healthy:

$$\mathbf{G}(V, U) \quad = \quad \neg\, \mathbf{G}(V, \neg\, U) \quad = \quad \exists x_V, x'_V \bullet \Delta V \land U$$

$$\begin{tikzcd}
P \arrow[r, "\mathbf{L}\,(V)"] & \mathbf{L}\,(V,P) \\
\mathbf{G}\,(V,U) \arrow[u, dashed, "\sqsubseteq"] & U \arrow[l, "\mathbf{G}\,(V)"'] \arrow[u, dashed, "\sqsubseteq"]
\end{tikzcd}$$

Figure 3.1: The Galois connection between $\mathbf{L}$ and $\mathbf{G}$.

3.3.3 A Galois Connection

As Example 3.23 demonstrates, the $\mathbf{L}$ and $\mathbf{G}$ predicate transformers are not inverses of each other, since a view need not define a one-to-one correspondence between behaviours and interactions. They do, however, form a Galois connection under the refinement ordering.

Theorem 3.25 ($\mathbf{L}$ and $\mathbf{G}$ form a Galois connection)**.** The $\mathbf{L}$ and $\mathbf{G}$ predicate transformers form a Galois connection between the space of behaviours and the space of interactions linked by a given view. Thus:

$$U \sqsubseteq \mathbf{L}\,(V,P) \quad \text{if and only if} \quad \mathbf{G}\,(V,U) \sqsubseteq P$$

Theorem 3.25 enables us to apply existing results on Galois connections from the literature. For instance, Corollary 3.26 is an immediate consequence of Theorem 3.25.

Corollary 3.26. Substituting $\mathbf{L}\,(V,P)$ for U in Theorem 3.25 yields:

$$\mathbf{G}\,(V,\mathbf{L}\,(V,P)) \quad \sqsubseteq \quad P$$

Corollary 3.26 justifies the intuition that applying $\mathbf{L}\,(V)$ to a system P may discard information about P's behaviours that cannot be recovered by applying $\mathbf{G}\,(V)$ to the result.

In Chapter 4 of the UTP textbook, Hoare and He emphasise the value of Galois connections for linking UTP theories. Theorem 3.25 links UTP theories with counterparts of those theories defined over the space of user interactions. We explore these counterpart theories in Section 3.4 and Section 3.5.

Remark 3.27. A view V can be interpreted as a linking predicate between two data types, where behaviours are instances of the concrete data type, while interactions are instances of the abstract data type. In this setting, $\mathbf{L}(V)$ is an abstraction function and $\mathbf{G}(V)$ is a concretisation function.

This insight suggests how *abstract interpretation* (Cousot and Cousot, 1977, 1992) — a theory for approximating the behaviour of programs to simplify proofs of correctness properties, by abstracting away irrelevant detail — could be formulated in the UTP. By defining a view V to retain only information about a program P essential for a property, it would be more efficient to prove the property over $\mathbf{L}(V, P)$ rather than P.

3.3.4 Inference

Suppose a user knows the structure of a system P and its own view V. On making a interaction ψ with P, the user can calculate the space of all behaviours of P that *may* have generated ψ. This calculation is specified by Definition 3.28.

Definition 3.28 (Inference function).

$$\mathsf{infer}(P, V, \psi) \quad \triangleq \quad P \wedge \neg\, \mathbf{G}(V, \neg\, \psi)$$

Corollary 3.29. Lemma 3.24 implies that whenever V is **VH3**-healthy:

$$\mathsf{infer}(P, V, \psi) \quad = \quad P \wedge \exists\, x_V, x_V' \bullet \Delta V \wedge \psi$$

A function similar to that presented in Corollary 3.29 is called the *inference function* by Jacob (1988, 1989a). It expresses the maximum information that a user at V can deduce about the behaviour of P from its interaction ψ alone.

Example 3.30. Suppose Alice plays a simple guessing game $GUESS_0$. In this game, a number $n \in 1..10$ is chosen and concealed from Alice. Alice makes a guess g of the value of n, whereupon she is informed whether her guess is correct or greater or smaller than n, via the response variable r'. We specify the guessing game as follows:

$$GUESS_0 \quad \triangleq \quad n \in 1..10 \land g \in 1..10 \land \begin{pmatrix} & g > n \Rightarrow r' > 0 \\ \land & g = n \Rightarrow r' = 0 \\ \land & g < n \Rightarrow r' < 0 \end{pmatrix}$$

Alice's view allows her to observe the values of g and r, but not n:

$$ALICE \quad \triangleq \quad g_A = g \land r_A = r$$

If Alice guesses $g = 7$ and she is informed that $r' > 0$, we find:

$$\begin{array}{ll} & \mathsf{infer}\,(GUESS_0, ALICE, g_A = 7 \land r'_A > 0) \\ = & GUESS_0 \land \exists g_A, r_A, g'_A, r'_A, \bullet\ \Delta ALICE \land g_A = 7 \land r'_A > 0 \text{ [def } \mathsf{infer}] \\ = & n \in 8..10 \land g = 7 \land r' > 0 \qquad\qquad \text{[pred calc]} \end{array}$$

Hence, Alice can infer that $n \in 8..10$ from her interaction. $\Diamond$

From the perspective of the user at V, each behaviour of $\mathsf{infer}\,(P, V, \psi)$ is consistent with ψ. The stronger $\mathsf{infer}\,(P, V, \psi)$ is, the *fewer* behaviours of P are consistent with ψ, enabling the user at V to draw stronger inferences about the behaviour of P.

Remark 3.31. The inference function can also be used to calculate what information a user at V_1 can infer about the possible interactions through a disjoint view V_2 belonging to another user. For instance:

$$\mathbf{L}\,(V_2, \mathsf{infer}\,(P, V_1, \psi))$$

models all V_2-interactions with P that comply with a V_1-interaction ψ.

3.4 Multi-User Designs

So far, we have studied multi-user systems in the basic setting of alphabetised relations. We now investigate how the concepts we have developed can be specialised for the theory of designs.

3.4.1 Views for Designs

We interpret a UTP design D as starting in an initial state consisting of user inputs and terminating (or aborting) in a final state output to users.

It is reasonable to expect that the users of D can observe whether D has started or has terminated. Since the behavioural variables ok and ok' are hidden from users, it is necessary to extend the alphabet of a view V by introducing a new interaction variable ok_V to mirror ok. We formalise the expectation that $ok_V = ok$ (and $ok'_V = ok'$ by extension) with a new healthiness condition.

Definition 3.32 (**OK** healthiness condition)**.**

$$\mathbf{OK}(V) \quad \triangleq \quad V \wedge ok_V = ok$$

Definition 3.33 (**VHD**)**.** A view is **VHD**-healthy if and only if it is both **VH**-healthy and **OK**-healthy.

Henceforth, we require that views applied to designs are **VHD**-healthy.

3.4.2 Projecting Designs

The predicate that results from projecting a design through a **VHD**-healthy view is not itself a design, because it is defined over interactions, not behaviours. Yet this predicate is isomorphic to designs, so it possesses the special properties of designs in a renamed observational space. For this reason, we introduce a notation for projected designs in Definition 3.34.

Definition 3.34 (Local design)**.** Let V denote a view and $Pre, Post$ denote predicates with alphabet x_V, x'_V:

$$Pre \vdash_V Post \quad \triangleq \quad ok_V \land Pre \Rightarrow ok'_V \land Post$$

The $\mathbf{L}$ predicate transformer can be applied to a $\mathbf{VHD}$-healthy view V and design D, to obtain an interface design that expresses the behaviours of D as projected through V.

Lemma 3.35 ($\mathbf{L}$ and designs)**.** Provided V is $\mathbf{VHD}$-healthy, $\mathbf{L}\,(V, Pre \vdash Post)$ can always be expressed as a local design:

$$\mathbf{L}\,(V, Pre \vdash Post) \quad = \quad \neg\, \mathbf{L}\,(V, \neg\, Pre) \vdash_V \mathbf{L}\,(V, Post)$$

Moreover, if Pre is a condition (i.e. its alphabet contains only undecorated variables), then:

$$\mathbf{L}\,(V, Pre \vdash Post) \quad = \quad (\forall\, x \bullet V \Rightarrow Pre) \vdash_V \mathbf{L}\,(V, Post)$$

The precondition $PreL = (\forall\, x \bullet V \Rightarrow Pre)$ is the weakest predicate over interactions such that *all* behaviours of D that are consistent with each interaction of $PreL$ satisfy Pre. Therefore, if a user's interaction with D satisfies $ok_V \land PreL$, then that user can be certain that $ok \land Pre$ holds. In turn, the user can ascertain that $Post$ (and its projection $\mathbf{L}\,(V, Post)$) will hold upon D terminating.

Remark 3.36. If no projection of D through V guarantees that Pre holds, then the precondition of $\mathbf{L}\,(V, D)$ collapses to **false**. It follows that:

$$\begin{aligned} & \mathbf{false} \vdash_V \mathbf{L}\,(V, Post) & \\ = \quad & ok_V \land \mathbf{false} \Rightarrow ok'_V \land \mathbf{L}\,(V, Post) & \text{[Definition 3.34]} \\ = \quad & \mathbf{true} & \text{[prop calc]} \end{aligned}$$

and so a user at V can infer nothing about the final state of $\mathbf{L}\,(V, D)$.

3.4.3 Worked Example: A Byte Register

We now consider how the theory developed in the previous sections can be applied to a simple multi-user system. The aim of this example is to demonstrate how the specification and design of multi-user systems may be guided by knowledge of user interactions.

We focus on a register capable of storing a single (8-bit) byte. We model the register's value by an integer variable *reg* with domain 0..255. The register also features a binary-valued flag *err* that indicates numeric overflow when raised. The state space of the register is described by:

$$ST \quad \triangleq \quad reg \in 0..255 \wedge err \in 0..1$$

Consider an operation that doubles the value stored in *reg*, provided that the initial value of *reg* lies in the range 0..127 and the overflow flag is not raised. We model this operation as a UTP design as follows:

$$DBL \quad \triangleq \quad reg \in 0..127 \wedge err = 0 \vdash reg' = reg \times 2 \wedge err' = 0$$

Suppose that two users can observe the register: the first user can observe the values of the higher four bits of the value of *reg*, and the second user can observe the lower four bits of *reg*. The overflow flag *err* is visible to both users. The views of these users are defined as follows:

$$\begin{aligned} H &\triangleq \mathsf{OK}\left(reg_H = \left\lfloor \frac{reg}{16} \right\rfloor \wedge err_H = err\right) \\ L &\triangleq \mathsf{OK}\left(reg_L = reg \bmod 16 \wedge err_L = err\right) \end{aligned}$$

Effectively, the *H* view masks out the lower four bits of the register from the first user's perspective, while the *L* view hides the higher four bits from the second user. Both of these views are **VHD**-healthy.

Example 3.37. Consider the instantiation $\phi = (reg = 60 \wedge err = 0)$ of *ST*. This state projects through *H* and *L* as follows:

$$\mathsf{L}(H, \phi) \quad = \quad reg_H = 3 \wedge err_H = 0$$

$$\mathbf{L}\,(L, \phi) \quad = \quad reg_L = 12 \land err_L = 0$$

Notice the binary representation of *reg* (00111100) is given by concatenating reg_H (0011) with reg_L (1100). ◊

Using Lemma 3.35, we now calculate the projections of *DBL*'s behaviour through the views *H* and *L*, constrained by the register's state space *ST*.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{L}\,(H \land ST, DBL)} \\
= & \begin{pmatrix} \forall\, x \bullet (H \land ST) \Rightarrow reg \in 0..127 \land err = 0 \\ \quad \vdash_H \;\; \exists\, x, x' \bullet \Delta(H \land ST) \land reg' = reg \times 2 \land err' = 0 \end{pmatrix} \\
= & \begin{pmatrix} reg_H \in 0..7 \land err_H = 0 \\ \quad \vdash_H \;\; (reg'_H = reg_H \times 2 \lor reg'_H = (reg_H \times 2) + 1) \land err'_H = 0 \end{pmatrix}
\end{array}$$

This calculation indicates the user at *H* can be sure that *DBL*'s precondition is satisfied if and only if $reg_H \in 0..7$ (i.e. $reg \in 0..127$), since any other value of reg_H corresponds to a value of *reg* that violates the precondition of *DBL*.

In the postcondition of *DBL*, the value of the fifth most significant bit of *reg* determines whether $reg'_H = reg_H \times 2$ or $reg'_H = (reg_H \times 2) + 1$. Since that bit of *reg* cannot be observed through *H*, the user at *H* can only be sure of the value of reg'_H once the operation is complete.

It is instructive to consider what can be observed of *DBL* through *L*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{L}\,(L \land ST, DBL)} \\
= & \begin{pmatrix} \forall\, x \bullet (L \land ST) \Rightarrow reg \in 0..127 \land err = 0 \\ \quad \vdash_L \;\; \exists\, x, x' \bullet \Delta(L \land ST) \land reg' = reg \times 2 \land err' = 0 \end{pmatrix} & \\
= & \mathbf{false} \vdash_L (reg'_L = (reg_L \times 2) \bmod 16 \land err'_L = 0) & \text{[pred calc]} \\
= & \mathbf{true} & \text{[Remark 3.36]}
\end{array}$$

Since an observation at *L* provides no information regarding the most significant bit of *reg*, a user at *L* cannot distinguish between $reg \in 0..127$ or $reg \in 128..255$. Hence, that user cannot determine in any circumstances whether the precondition of *DBL* is satisfied. The outcome of this calculation reflects that, from that user's perspective, nothing can be guaranteed about *DBL*'s behaviour.

Depending on the context in which the register is used, this limitation may be unacceptable. Hence, we relax the precondition of *DBL* to cover all values of *reg* that can be stored by the register and, when $reg \geq 128$, to assign an arbitrary value from the range 0..255 to reg' and set err' to 1:

$$DBL2 \quad \triangleq \quad reg \in 0..255 \wedge err = 0 \vdash \begin{pmatrix} reg' = reg \times 2 \wedge err' = 0 \\ \lhd\ reg \in 0..127\ \rhd \\ reg' \in 0..255 \wedge err' = 1 \end{pmatrix}$$

Observe that $DBL \sqsubseteq DBL2$, because the postcondition of *DBL*2 reduces to the postcondition of *DBL* when the precondition of *DBL* is satisfied.

The projection of *DBL*2 through *L* is as follows:

$$\begin{aligned}
&\mathsf{L}\,(L \wedge ST, DBL2) \\
=\ & \begin{pmatrix} \forall x \bullet (L \wedge ST) \Rightarrow reg \in 0..255 \wedge err = 0 \\ \vdash_L\ \exists x, x' \bullet \Delta(L \wedge ST) \wedge \begin{pmatrix} reg' = reg \times 2 \wedge err' = 0 \\ \lhd\ reg \in 0..127\ \rhd \\ reg' \in 0..255 \wedge err' = 1 \end{pmatrix} \end{pmatrix} \\
=\ & \begin{pmatrix} reg_L \in 0..15 \wedge err_L = 0 \\ \vdash_L \begin{pmatrix} reg'_L = (reg_L \times 2) \,\mathsf{mod}\, 16 \wedge err'_L = 0 \\ \vee\ \ reg'_L \in 0..15 \wedge err'_L = 1 \end{pmatrix} \end{pmatrix}
\end{aligned}$$

After the *DBL*2 operation completes, the user at *L* can verify the doubling operation was successful by checking that its interaction satisfies the precondition of $\mathsf{L}\,(L, DBL2)$ and that $err'_L = 0$.

Remark 3.38. If the state of *DBL*2 is data-refined, then the corresponding data refinements must also be made to the *H* and *L* views. For instance, if we were to data-refine *reg* into eight binary variables $r7, \ldots, r0$ to represent the bits of the register, we would replace H with the view:

$$\mathbf{OK}\,(r7_H = r7 \wedge r6_H = r6 \wedge r5_H = r5 \wedge r4_H = r4 \wedge err_H = err)$$

which could be calculated by projecting a linking predicate through H.

3.5 Multi-User Reactive Designs

We now apply our framework to the UTP theory of reactive processes. This work allows us to formalise users' interactions with CSP or *Circus* processes, since these formalisms have a reactive design semantics.

3.5.1 Reactive Views

Our first task is to formalise how users can interact with reactive processes, by defining a family of *reactive views*. It is natural to interpret a user's interface in terms of the events that a reactive process communicates with its environment.

Definition 3.39 (Window)**.** A user's *window* is the set of events communicated by a reactive process which are visible to the user.

We now consider which observational elements of a reactive process are relayed to a user with window $\mathcal{W}$.

State A process does not expose its internal state to the external environment. Hence, a view should not convey any information about the state variables v to the user.

Stability We suppose the user can observe whether the process has reached a stable point in its execution by requiring its view to be **OK**-healthy, for the same reasons as those given in Section 3.4.

Progress We enable the user to determine whether the process is waiting to start, is waiting for any interaction with the environment or has terminated, by expanding the user's view with a local variable $wait_V$ corresponding to $wait$.

Trace Since events take place sequentially, we model a user as perceiving their order of occurrence. Hence, a user's observation of the global trace *tr* is given by restricting *tr* to only those events in $\mathcal{W}$:

$$tr_V \quad = \quad tr \upharpoonright \mathcal{W}$$

where $tr \upharpoonright \mathcal{W}$ denotes *tr*, but with all events outside $\mathcal{W}$ erased.

Refusals A user who interacts with the process directly may discover that, at some point, the events it wishes to perform with the process are refused. We model the projection of a process's refusal set as:

$$ref_V \cap \mathcal{W} \quad \subseteq \quad ref$$

This expression only reveals information about the events of *ref* in $\mathcal{W}$ to the user.

Remark 3.40. At first glance, it would seem more natural to define:

$$ref_V \quad \subseteq \quad ref \cap \mathcal{W}$$

but this expression implies that, from the user's perspective, the process can never refuse any event outside $\mathcal{W}$. When combined with the projection functions for the other observational elements, this expression leads to projected processes that violate the failures model of CSP: they would not engage in events outside $\mathcal{W}$, but would not refuse those events either.

We impose these expectations on views by defining another healthiness condition **VHR**.

Definition 3.41 (**VHR** healthiness condition)**.**

$$\mathbf{VHR}\,(\mathcal{W}, V) \quad \triangleq \quad \left(\begin{array}{ll} & \exists\, v \bullet \mathbf{VH}\,(V) \\ \wedge & ok_V = ok \wedge wait_V = wait \\ \wedge & tr_V = tr \upharpoonright \mathcal{W} \wedge ref_V \cap \mathcal{W} \subseteq ref \end{array}\right)$$

where v denotes the list of state variables, excluding $ok, wait, tr, ref$.

Following **VHD**, the inclusion of *ok* in **VHR** is vital to support any meaningful model of a user's interactions with a process. The *wait* variable, while not essential, is still important if we wish to study a user's interactions with a process in terms of its component actions. (Indeed, compositional reasoning plays a key role in Chapter 5.)

The inclusion of the trace and refusal variables is less important. If we take $\mathcal{W} = \emptyset$, then we derive a minimal interface representing a user who cannot engage in activity with the process, but who could still perceive divergence and termination via $\neg\, ok'_V$ and $\neg\, wait'_V$ respectively.

Remark 3.42. We may supply **VHR** with views modelling other facets of a user's interface to a process. For instance, an interface that indicates whether the process engages in an event in $\mathcal{Y}$ before an event in $\mathcal{Z}$ (where $\mathcal{Y}$ and $\mathcal{Z}$ are disjoint from $\mathcal{W}$) is modelled by the view:

$$\mathbf{VHR}\left(\mathcal{W}, xy_V \Leftrightarrow \exists\, \alpha, \beta \bullet \left(\begin{array}{rc} & tr = \alpha \frown \beta \wedge \alpha \upharpoonright \mathcal{Y} \neq \langle\rangle \\ \wedge & \alpha \upharpoonright \mathcal{Z} = \langle\rangle \wedge \beta \upharpoonright \mathcal{Z} \neq \langle\rangle \end{array}\right)\right)$$

These more elaborate views inhibit compositional reasoning about users' interactions with processes, which would be a hindrance in later chapters. For this reason, we do not consider these views further.

Recall from Section 2.5 that a **R2**-healthy process P is insensitive to the value of tr. If we limit our attention to **R2**-healthy processes, then the projection tr_V of tr given by **VHR** is redundant:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{R2}\,(tr_V = tr \upharpoonright \mathcal{W} \wedge tr'_V = tr' \upharpoonright \mathcal{W})} & \\
= & (tr_V = tr \upharpoonright \mathcal{W} \wedge tr'_V = tr' \upharpoonright \mathcal{W})[\langle\rangle, tr' - tr / tr, tr'] & [\text{def } \mathbf{R2}] \\
= & tr_V = \langle\rangle \upharpoonright \mathcal{W} \wedge tr'_V = (tr' - tr) \upharpoonright \mathcal{W} & [\text{substitution}] \\
= & tr_V = \langle\rangle \wedge tr'_V = (tr' - tr) \upharpoonright \mathcal{W} & [\text{property of } \upharpoonright] \\
= & tr_V = \langle\rangle \wedge tr'_V - tr_V = (tr' - tr) \upharpoonright \mathcal{W} & [\text{property of } -]
\end{array}$$

We adopt a slight variation of Δ**VHR** $(\mathcal{W}, \mathbf{true})$ as our model of a user's interface to a reactive process, given in Definition 3.43. This variation omits the $tr_V = \langle\rangle$ conjunct.

Definition 3.43 (Reactive interface)**.**

$$\mathcal{R}(\mathcal{W}) \triangleq \left(\begin{array}{ll} & ok_V = ok \wedge ok'_V = ok' \\ \wedge & wait_V = wait \wedge wait'_V = wait' \\ \wedge & tr'_V - tr_V = (tr' - tr) \upharpoonright \mathcal{W} \\ \wedge & ref_V \cap \mathcal{W} \subseteq ref \wedge ref'_V \cap \mathcal{W} \subseteq ref' \end{array}\right)$$

To some extent, the decision to work with interfaces of the $\mathcal{R}(\mathcal{W})$ form is arbitrary. Nevertheless, there is a close relationship between this form and existing work in the literature, which is revealed in Subsection 3.5.3.

Remark 3.44. While $\mathcal{R}(\mathcal{W})$ includes a projection of ref, that variable has no significance when dealing with **CSP3**-healthy processes, such as *Circus* actions (Oliveira et al., 2009).

3.5.2 Projecting a Reactive Process

For convenience, we define a specialised version of the **L** predicate transformer that encodes $\mathcal{R}(\mathcal{W})$ in place of a view.

Definition 3.45 (**LR** predicate transformer)**.** Provided P is **R2**-healthy:

$$\begin{array}{rcl} \mathbf{LR}(\mathcal{W}, P) & \triangleq & \mathbf{L}(\mathbf{VHR}(\mathcal{W}, \mathbf{true}), P) \\ & = & \exists x, x' \bullet \mathcal{R}(\mathcal{W}) \wedge P \end{array}$$

Applying **LR** to a **R**-healthy reactive process does not yield a **R**-healthy process, but a predicate that is isomorphic to a **R**-healthy process.

Definition 3.46 (Local **R**)**.** Let $\mathbf{R}_V(P) \triangleq \mathbf{R1}_V \circ \mathbf{R2}_V \circ \mathbf{R3}_V(P)$ and:

$$\begin{array}{rcl} \mathbf{R1}_V(P) & \triangleq & P \wedge tr_V \leq tr'_V \\ \mathbf{R2}_V(P) & \triangleq & P[\langle\rangle, tr'_V - tr_V / tr_V, tr'_V] \\ \mathbf{R3}_V(P) & \triangleq & \left((\exists v, v' \bullet \mathit{II}_{rea})[x_V, x'_V / x, x'] \lhd wait_V \rhd P\right) \end{array}$$

If P is $\mathbf{R}$-healthy, then $\mathbf{LR}\,(\mathcal{W},P)$ is $\mathbf{R}_V$-healthy. Since a reactive design is expressible as a design made $\mathbf{R}$-healthy, Lemma 3.47 presents a result similar to Lemma 3.35.

Lemma 3.47 ($\mathbf{LR}$ and reactive designs)**.** Provided P is a reactive design:

$$\mathbf{LR}\,(\mathcal{W},P) \quad = \quad \mathbf{R}_V\left(\left(\forall\, x,x' \bullet \mathcal{R}\,(\mathcal{W}) \Rightarrow \neg\, P^f_f\right) \vdash_V \mathbf{LR}\left(\mathcal{W},P^t_f\right)\right)$$

where P^b_c abbreviates $P[b,c/ok',wait]$ (as used in Theorem 2.2).

The predicate $\forall\, x,x' \bullet \mathcal{R}\,(\mathcal{W}) \Rightarrow \neg\, P^f_f$ describes all interactions for which a user at $\mathcal{W}$ can be certain that P does not diverge. The same effect manifests in the preconditions of multi-user designs, as described in Section 3.4.

Example 3.48. The reactive design semantics of $a \to b \to Skip$ is:

$$\mathbf{R}\left(\mathbf{true} \vdash \left(\begin{array}{c} (tr' = tr \wedge a \notin ref') \vee (tr' = tr \frown \langle a \rangle \wedge b \notin ref') \\ \lhd\, wait' \,\rhd \\ tr' = tr \frown \langle a,b \rangle \end{array}\right)\right)$$

The $\mathbf{LR}$-projection of $a \to b \to Skip$ through the window $\{a\}$ is:

$$\begin{array}{lll} & \mathbf{LR}\,(\{a\}\,,a \to b \to Skip) & \\ = & \mathbf{R}_V\left(\left(\begin{array}{l}(\forall\, x,x' \bullet \mathcal{R}\,(\{a\}) \Rightarrow \neg\,(a \to b \to Skip)^f_f) \\ \vdash_V\ \mathbf{LR}\,(\{a\}\,,a \to b \to Skip)^t_f\end{array}\right)\right) & \text{[Lemma 3.47]} \\ = & \mathbf{R}_V\left(\mathbf{true} \vdash_V \mathbf{LR}\left(\{a\}\,,(a \to b \to Skip)^t_f\right)\right) & \text{[no divergence]} \end{array}$$

Focusing on the postcondition of the interior local design, we obtain:

$$\begin{array}{lll} & \mathbf{LR}\left(\{a\}\,,(a \to b \to Skip)^t_f\right) & \\ = & \mathbf{LR}\left(\{a\}\,,\left(\begin{array}{c}\left(\begin{array}{ll} & tr' = tr \wedge a \notin ref' \\ \vee & tr' = tr \frown \langle a \rangle \wedge b \notin ref'\end{array}\right) \\ \lhd\, wait' \,\rhd \\ tr' = tr \frown \langle a,b \rangle\end{array}\right)\right) & \text{[semantics]} \end{array}$$

$$= \left(\begin{array}{c} \left(\begin{array}{cc} & \mathbf{LR}\left(\{a\}, tr' = tr \wedge a \notin ref'\right) \\ \vee & \mathbf{LR}\left(\{a\}, tr' = tr \frown \langle a\rangle \wedge b \notin ref'\right) \end{array}\right) \\ \lhd\, wait'_V \rhd \\ \mathbf{LR}\left(\{a\}, tr' = tr \frown \langle a, b\rangle\right) \end{array}\right) \qquad \text{[Law 3.19]}$$

$$= \left(\begin{array}{c} (tr'_V = tr_V \wedge a \notin ref'_V) \vee (tr'_V = tr_V \frown \langle a\rangle) \\ \lhd\, wait'_V \rhd \\ tr'_V = tr_V \frown \langle a\rangle \end{array}\right) \qquad \text{[def } \mathbf{LR}\text{]}$$

Hence, $\mathbf{LR}\left(\{a\}, a \to b \to Skip\right)$ is equal to:

$$\mathbf{R}_V\left(\mathbf{true} \vdash_V \left(\begin{array}{c} (tr'_V = tr_V \wedge a \notin ref'_V) \vee (tr'_V = tr_V \frown \langle a\rangle) \\ \lhd\, wait'_V \rhd \\ tr'_V = tr_V \frown \langle a\rangle \end{array}\right)\right)$$

which is isomorphic to the semantics of the process $a \to (Skip \sqcap Stop)$.

Likewise, the projection of $a \to b \to Skip$ through the window $\{b\}$ is:

$$\mathbf{R}_V\left(\mathbf{true} \vdash_V \left(tr'_V = tr_V \lhd wait'_V \rhd tr'_V = tr_V \frown \langle b\rangle\right)\right)$$

which is isomorphic to the process $Stop \sqcap b \to Skip$. ◊

3.5.3 Lazy Abstraction

In two papers (Roscoe et al., 1994; Roscoe, 1995), Roscoe introduced the notion of the *lazy abstraction* of a CSP process. The lazy abstraction of a process P, denoted by $\mathsf{Lazy}(\mathcal{W}, P)$, is a process describing the interactions that a user with window $\mathcal{W}$ can make of P.

Definition 3.49 reproduces Roscoe's definition of lazy abstraction.

Definition 3.49 (Lazy abstraction)**.** The lazy abstraction of a CSP process P to the user at window $\mathcal{W}$ is:

$$\mathsf{Lazy}(\mathcal{W}, P) \quad \triangleq \quad (P \,|[\, \mathcal{H} \,]|\, (\mu X \bullet Stop \sqcap (\sqcap e : \mathcal{H} \bullet e \to X))) \setminus \mathcal{H}$$

where $\mathcal{H}$ denotes $(\Sigma \setminus \mathcal{W})$, the set of all events outside $\mathcal{W}$.

The process[3] $\mu X \bullet Stop \sqcap (\sqcap e : \mathcal{H} \bullet e \rightarrow X)$ represents the activities of P that are controlled by the other users of P. Placing this process in parallel with P and hiding the $\mathcal{H}$ events models the behaviours of P as observed by the user at $\mathcal{W}$.

Example 3.50. The lazy abstractions of $a \rightarrow b \rightarrow Skip$ through the windows $\{a\}$ and $\{b\}$ respectively are:

$$\begin{aligned} \mathsf{Lazy}\left(\{a\}, a \rightarrow b \rightarrow Skip\right) &= a \rightarrow (Skip \sqcap Stop) \\ \mathsf{Lazy}\left(\{b\}, a \rightarrow b \rightarrow Skip\right) &= Stop \sqcap b \rightarrow Skip \end{aligned}$$

These processes mirror those derived in Example 3.48. $\Diamond$

Roscoe (1997) presents multiple forms of lazy abstraction defined over the various semantic models of CSP and proves their equivalence. We build on this work to establish a correspondence between the failures-divergences version of lazy abstraction and our **LR** predicate transformer.

Theorem 3.51 (Lazy abstraction as projection)**.** For any *divergence-free* CSP process P, $\mathsf{Lazy}(\mathcal{W}, P)$ is isomorphic to $\mathbf{LR}(\mathcal{W}, P_R)$, where P_R denotes the reactive design formulation of P.

Theorem 3.51 is useful, because it allows us to characterise **LR** algebraically. Hence, existing tools and techniques designed for calculating and analysing lazy abstractions can be ported to our framework. Theorem 3.51 also validates our reactive interface $\mathcal{R}$ (Definition 3.43). In the opposite direction, Theorem 3.51 suggests that lazy abstraction could be generalised by adopting a different interface specification instead of $\mathcal{R}$.

3.6 Refining Multi-User Systems

In previous sections, we have described how the UTP can be applied to calculate a user's interactions with systems. Following Jacob (1987), this

[3]This process is called $CHAOS(\mathcal{H})$ by Roscoe (1997). It should not be confused with the *Chaos* action of *Circus*: while $CHAOS(\mathcal{H})$ can accept or refuse any sequence of events drawn from $\mathcal{H}$, it never diverges, whereas *Chaos* can diverge.

section argues that it is permissible to weaken the canonical notion of refinement when developing multi-user systems. We also define multi-user notions of refinement in terms of the canonical refinement relation.

Intuitively, $P_1 \sqsubseteq P_2$ formalises the following *correctness criterion*:

CC1 P_2 may replace P_1 without its environment detecting the change.

This criterion assumes the existence of an entity which can monitor the entire environment of the system, and thereby observe its behaviour completely. However, in a multi-user system, the environment is structured by the interfaces of its users. If no user's interface spans the entirety of the environment, then the criterion is arguably too strong.

If we are developing a multi-user system, we may wish to weaken our correctness criterion to:

CC2 P_2 may replace P_1 without its users detecting the change.

Depending on the users' views, there are instances of P_1 and P_2 which uphold this criterion, but for which $P_1 \sqsubseteq P_2$ fails to hold.

Example 3.52. Returning to the byte register (Subsection 3.4.3), consider the following variant of the $DBL2$ operation that doubles the lower four bits of reg and the upper four bits of reg separately:

$$\begin{aligned}
INDBL2 &\triangleq reg \in 0..255 \land err = 0 \vdash \left(\begin{array}{c} HDBL \land LDBL \land err' = 0 \\ \lhd\ reg \in 0..127\ \rhd \\ reg' \in 0..255 \land err' = 1 \end{array} \right) \\
HDBL &= \lfloor reg'/16 \rfloor = (\lfloor reg/16 \rfloor \times 2) \\
LDBL &= reg' \bmod 16 = (reg \times 2) \bmod 16
\end{aligned}$$

$INDBL2$ allows the users at H and L to access separate registers without needing to keep those registers synchronised, which means that an implementation of $INDBL2$ may provide each user with their own local instance of the register.

Notice that $DBL2 \not\sqsubseteq INDBL2$, because when $reg \in 0..127$ and $err = 0$, $INDBL2$ always sets the fourth most significant bit of reg' to 0 regardless of the value of the fifth most significant bit of *reg*. However, the users at H and L cannot individually distinguish *INDBL*2 from *DBL*2, because *INDBL*2 provides the same interactions to each user as does *DBL*2. $\Diamond$

This section outlines a class of refinement relations — based on work by Jacob (1987, 1989b, 1992) — formulated over user interactions with a system, rather than the behaviours of the system. These refinement relations are more relaxed than behavioural notions of refinement: they satisfy **CC2** but not necessarily **CC1**.

3.6.1 Local Refinement

We call P_2 a *local refinement* of P_1 with respect to a view V if and only if each V-interaction with P_2 is also a V-interaction with P_1.

Definition 3.53 (Local refinement).

$$P_1 \sqsubseteq_V P_2 \quad \triangleq \quad \mathsf{L}\,(V, P_1) \sqsubseteq \mathsf{L}\,(V, P_2)$$

Local refinement allows new behaviours to be *added* to a system, provided these new behaviours do not induce new interactions through V. Hence, if $P_1 \sqsubseteq_V P_2$ holds, then a user at V cannot detect the replacement of P_1 with P_2.

Corollary 3.54. Definition 3.53 and Law 3.20 imply, for all V:

$$P_1 \sqsubseteq P_2 \quad \text{implies} \quad P_1 \sqsubseteq_V P_2$$

Unlike $\sqsubseteq$, the $\sqsubseteq_V$ ordering is not antisymmetric in general, because two different systems offering the same interactions through V are equivalent under $\sqsubseteq_V$. Nevertheless, $\sqsubseteq_V$ is always a pre-order.

Remark 3.55. If P_1 and P_2 are CSP processes and $V = \mathsf{VHR}\,(\mathcal{W}, \mathbf{true})$ then, by Theorem 3.51, $P_1 \sqsubseteq_V P_2$ is equivalent to the condition:

$$\mathsf{Lazy}\,(\mathcal{W}, P_1) \quad \sqsubseteq \quad \mathsf{Lazy}\,(\mathcal{W}, P_2)$$

This result implies $P_1 \sqsubseteq_V P_2$ can be tested with a CSP model checker.

Since local refinement is formulated in terms of a single user, it needs to be extended if we wish to reason about multiple users.

3.6.2 Co-operating and Independent Refinement

Jacob (1987) proposed two notions of refinement, known as *co-operating refinement* and *independent refinement*, intended for the development of multi-user systems.

Co-operating refinement assumes that users can exchange information with each other about their interactions with a system. This means the users could reconstruct more information about the system behaviour than they could infer from their individual interactions.

Independent refinement assumes users cannot communicate with each other; instead, the only information that each user can obtain about the behaviour of a system is from their own interaction.

Jacob defined co-operating and independent refinement in the form:

$$\begin{array}{rcl} P_1 \sqsubseteq^{co}_{\mathcal{A}} P_2 & \triangleq & \forall\, \phi_2 : \omega(P_2) \bullet \exists\, \phi_1 : \omega(P_1) \bullet \forall\, U : \mathcal{A} \bullet \phi_1 \cong_U \phi_2 \\ P_1 \sqsubseteq^{ind}_{\mathcal{A}} P_2 & \triangleq & \forall\, U : \mathcal{A} \bullet \forall\, \phi_2 : \omega(P_2) \bullet \exists\, \phi_1 : \omega(P_1) \bullet \phi_1 \cong_U \phi_2 \end{array}$$

where $\mathcal{A}$ denotes a set of users, $\omega(P)$ denotes the space of behaviours of P (with respect to some semantic model), and $\phi_1 \cong_U \phi_2$ holds if and only if the behaviours ϕ_1 and ϕ_2 yield the same interaction to the user U.

We express co-operating and independent refinement in the UTP by extending the definition of local refinement to a set of pairwise disjoint views VS (representing multiple users) in different ways.

Definition 3.56 (Co-operating and independent refinement)**.**

$$\begin{array}{rcl} P_1 \sqsubseteq^{co}_{VS} P_2 & \triangleq & P_1 \sqsubseteq_{\bigwedge VS} P_2 \\ P_1 \sqsubseteq^{ind}_{VS} P_2 & \triangleq & \bigwedge_{V \in VS} P_1 \sqsubseteq_V P_2 \end{array}$$

The UTP style obscures the essential equivalence between these definitions and Jacob's. For co-operating refinement, $P_1 \sqsubseteq_{\bigwedge VS} P_2$ demands that for each behaviour ϕ_2 of P_2, there exists a behaviour ϕ_1 of P_1 such that ϕ_1 and ϕ_2 yield the same $(\bigwedge VS)$-interaction, where the view $\bigwedge VS$ represents the combined interactions made by the users (Subsection 3.2.3). A similar comparison can be made for independent refinement.

As with $\sqsubseteq_V$, the $\sqsubseteq^{co}_{VS}$ and $\sqsubseteq^{ind}_{VS}$ orderings are pre-orders but not partial orders (Jacob, 1989b). When $VS = \{V\}$, both $\sqsubseteq^{co}_{VS}$ and $\sqsubseteq^{ind}_{VS}$ reduce to $\sqsubseteq_V$.

Co-operating refinement is no stronger than behavioural refinement:

$$P_1 \sqsubseteq P_2 \quad \text{implies} \quad P_1 \sqsubseteq^{co}_{VS} P_2$$

which follows from Law 3.20 and the definition of local refinement. Similarly, it follows from Law 3.21 that independent refinement is no stronger than co-operating refinement:

$$P_1 \sqsubseteq^{co}_{VS} P_2 \quad \text{implies} \quad P_1 \sqsubseteq^{ind}_{VS} P_2$$

Even though cooperating and independent refinement are generally weaker than behavioural refinement, they are strong enough to preserve the functionality inherent in a system's specification *from the perspectives of the users*, provided their assumptions about users are upheld.

Example 3.57. Returning to Example 3.52, it is the case that:

$$DBL2 \sqsubseteq^{ind}_{\{H,L\}} INDBL2 \quad \text{but not} \quad DBL2 \sqsubseteq^{co}_{\{H,L\}} INDBL2$$

Provided the register can only be accessed through H and L — and the users at H and L do not co-operate — the replacement of $DBL2$ with $INDBL2$ is justified by independent refinement. However, this replacement is not justified by co-operating refinement, because if the users at H and L combine their interactions, they can identify behaviours of $INDBL2$ that are not behaviours of $DBL2$. $\Diamond$

We do not study co-operating or independent refinement further in this thesis. Nevertheless, as Example 3.57 demonstrates, they provide system developers with an extra degree of flexibility in making design choices than behavioural refinement can offer. Indeed, they would be particularly appropriate in the design of distributed multi-user systems, because they offer the opportunity to distribute a system's workload across multiple processors without needing to keep those processors synchronised.

3.7 Conclusion

This chapter has presented a generic framework for studying the interactions between systems and their users. By separating the concern of modelling users from designing systems, we can specify directly which elements of a system's behaviour are viewable by users. With predicate transformers, these specifications of views can be applied to calculate a user's interactions with a system, as well as a user's inferences about the system's behaviour from those interactions.

This framework fits seamlessly within the predicate semantics of the UTP. Hence, it can be applied to derive a multi-user interpretation of any UTP theory. We have demonstrated its application to the theories of designs and reactive processes and, in the latter case, have identified a link with Roscoe's lazy abstraction.

We have also applied the framework to capture weaker notions of refinement that are suitable for multi-user system development. The connections between our framework, Roscoe's lazy abstraction and the notions of co-operating and independent refinement together imply there is scope for using model checkers for CSP for automatically verifying these weaker refinement relations hold between processes.

The notions of views and inferences are taken forward to the next chapter, where we study information flow security in the UTP context.

4 Confidentiality Properties

4.1 Introduction

Confidentiality demands that a system only releases secret information — such as cryptographic keys or classified documents — to users who are authorised to access that information. Since the advent of time-sharing computer systems in the late 1960s, much research has focused on protecting the confidentiality of data stored within multi-user systems. The concepts of *access control* and *inference control* have proved influential, in different ways.

4.1.1 Access Control

Access control models — most famously, the Bell–La Padula (BLP) model (Bell and La Padula, 1973, 1976) — describe an abstract mechanism for controlling the operations on the system's state that each user of the system is entitled to perform. It is intended that, if a system is implemented in accordance with this mechanism, then the mechanism will prevent a user from invoking an operation (or sequence of operations) to access data that it is not authorised to access.

Access control is readily comprehensible by software engineers and is widely deployed in software products. Yet despite its intuitive appeal, its adequacy for describing security policies has been criticised on both theoretical and practical grounds:

Covert channels A *covert channel* allows users to communicate data in a manner unanticipated by the system's designers (Lampson, 1973). These channels often emerge from shared resources: for instance, a high-level user (or a Trojan horse program invoked with high-level privileges) may modulate its consumption of a system resource (such as processor time or memory) to signal data to a low-level user. Covert channels lie outside the domain of access control, because they are an emergent property of the system's resources.

Narrow focus Access control models are formulated in terms of active subjects (users) operating over a collection of passive objects, such as files and database records. However, confidentiality requirements may apply to other facets of a system, such as whether a high-level user has engaged in particular activities.

Semantic ambiguity It is the job of software engineers to determine how an access control model should be mapped onto a system design. For instance, models such as BLP are specified in terms of "read" and "write" operations, but the semantics of these operations is not made precise. Such ambiguities can lead to insecurity: for instance, McLean (1987) outlined a hypothetical "System Z" which implements BLP, but downgrades the security level of all data objects for the duration of each operation it performs, enabling low-level users to access any object whatsoever.

4.1.2 Inference Control

A radically different philosophy for protecting the confidentiality of information is to regulate the *information flow* from systems to their users. We observed in Subsection 3.3.4 that a user can infer information about the system's behaviour by analysing its interaction with the system. By extension, that user can calculate information about the system's state, or the activities of other users (Denning, 1982).

A confidentiality property specifies which aspects of a system's behaviour are secret (in some way) and prescribes an upper bound on the flow of information about those aspects to low-level users. The meaning of confidentiality properties is unambiguous and is independent of the system under consideration. Unlike access control models, they do not prescribe any mechanism for safeguarding confidential data. Hence, confidentiality properties grant system designers the flexibility to choose how to implement security measures as they see fit (McLean, 1994b).

Various specialised techniques for verifying a system design against particular confidentiality properties have been proposed in the literature. However, the topic of integrating these techniques with existing high-integrity software engineering practices has remained under-explored.

4.1.3 Contributions of this Chapter

Building on the framework of user interactions developed in Chapter 3, this chapter outlines a style for specifying confidentiality properties over any aspect of the observation space defined by a UTP theory. This formulation of confidentiality properties provides a starting point for integrating notions of confidentiality into specification and programming languages with a UTP semantics.

Section 4.2 applies the predicate semantics of the UTP — coupled with the notion of views described in Chapter 3 — to formalise a user's inferences about the behaviour of systems. This work provides a semantic foundation for our definition of confidentiality properties, which is presented in Section 4.3.

Since confidentiality properties are inherently non-functional, they are typically kept separate from the functional specification of a system. In Section 4.4, we show how a specification of a *secure* system may be derived by merging a functionality specification with confidentiality properties. Then, in Section 4.5, we investigate why confidentiality properties are not preserved by refinement, but also show how this problem is resolved by merging functionality and confidentiality together.

In Section 4.6, we show how various information flow properties defined in the literature can be expressed in our confidentiality framework. We also compare our approach for specifying confidentiality with the body of existing work on information flow security in Section 4.7.

4.2 An Indistinguishability Semantics

Information flow is a product of the relationship between a system's behaviours and a user's inferences about those behaviours. In this section, we describe a method for extending UTP theories to encode information flow: in effect, we model a user's inferences about the behaviour of a system *in terms of the system's behaviours themselves*.

The cornerstone of our method is a mapping from the behaviours of a system to an isomorphic space of behaviours that we call the *fog space*. Intuitively, if a behaviour ϕ is mapped to a fog behaviour $\widetilde{\phi}$, then $\widetilde{\phi}$ represents an *alternative explanation* for ϕ in some respect. It is important to stress that our fog space is a fiction: we do not expect fog behaviours to manifest in any real system.

The fog space is conceptually very useful, because it enables us to formulate relational properties between behaviours that cannot be defined in the observable space alone. In particular, we employ the fog space to capture two different security concerns. In this section, we study how a user's inferences about the process's behaviour can be encoded in the fog. In Section 4.3, we investigate how the fog can serve as the medium underlying the specification of confidentiality properties.

4.2.1 Lifting Relations

We derive a fog alphabet $\widetilde{x}, \widetilde{x'}$ by renaming the observation variables x, x'. Given a relation P, the *lift* of P relates the observable behaviours of P to a renaming of those behaviours over the fog variables. Definition 4.1 presents a predicate transformer **U** ("up") for lifting relations.

Definition 4.1 (**U** predicate transformer)**.** Let $\widetilde{P}$ denote $P[\widetilde{x}, \widetilde{x'} / x, x']$:

$$\mathbf{U}(P) \quad \triangleq \quad P \wedge \widetilde{P}$$

From this point onwards, we say that a predicate is a *lifted relation* if its alphabet contains the x, x' variables and the $\widetilde{x}, \widetilde{x'}$ fog variables.

Example 4.2. To understand the structure of lifted relations, it is helpful to expand them into a form where observable behaviours are paired with fog behaviours. For instance:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{U}\,(h = 0 \lor h = 1)} & \\
= & (h = 0 \lor h = 1) \land (\widetilde{h} = 0 \lor \widetilde{h} = 1) & [\text{def } \mathbf{U}] \\
= & \left(\begin{array}{ll} & (h = 0 \land \widetilde{h} = 0) \lor (\widetilde{h} = 0 \land \widetilde{h} = 1) \\ \lor & (h = 1 \land \widetilde{h} = 0) \lor (\widetilde{h} = 1 \land \widetilde{h} = 1) \end{array}\right) & [\text{prop calc}]
\end{array}$$

Here, each observable behaviour is paired with each fog behaviour. $\Diamond$

Remark 4.3. The $\mathbf{U}$ function is related to the *separating simulation* functions defined by Hoare and He (1998). Separating simulations rename the alphabet of a relation, to facilitate reasoning about updates to shared variables. Here, $\mathbf{U}\,(P)$ models two completely separate instances of a system with no shared variables between them.

Theorem 4.4 highlights an important property of $\mathbf{U}$: it preserves the *structure* of a UTP theory.

Theorem 4.4 ($\mathbf{U}$ is an order-embedding)**.** The $\mathbf{U}$ predicate transformer is both monotonic and order-reflecting under the refinement ordering:

$$P_1 \sqsubseteq P_2 \quad \text{if and only if} \quad \mathbf{U}\,(P_1) \sqsubseteq \mathbf{U}\,(P_2)$$

A consequence of this order-embedding is that, by hiding the fog variables of a lifted relation $\mathbf{U}\,(P)$, we can recover the relation P. The $\mathbf{D}$ ("down") predicate transformer does just that.

Definition 4.5 ($\mathbf{D}$ predicate transformer)**.**

$$\mathbf{D}\,(Q) \quad \triangleq \quad \exists\, \widetilde{x}, \widetilde{x'} \bullet Q$$

Lemma 4.6 is a simple consequence of Definition 4.5.

Lemma 4.6 (**D** inverse of **U**). Provided P is a relation:

$$\mathbf{D}(\mathbf{U}(P)) \quad = \quad P$$

Since **U** is defined in terms of conjunction, it is conjunctive. Likewise, **D** is disjunctive, because existential quantification is disjunctive.

Law 4.7 (**U** is conjunctive). $\mathbf{U}(P_1 \wedge P_2) = \mathbf{U}(P_1) \wedge \mathbf{U}(P_2)$

Law 4.8 (**D** is disjunctive). $\mathbf{D}(Q_1 \vee Q_2) = \mathbf{D}(Q_1) \vee \mathbf{D}(Q_2)$

However, **U** is not disjunctive, and **D** is not conjunctive.

Law 4.9 (**U** and disjunction). $\mathbf{U}(P_1 \vee P_2) \sqsubseteq \mathbf{U}(P_1) \vee \mathbf{U}(P_2)$

Law 4.10 (**D** and conjunction). $\mathbf{D}(Q_1) \wedge \mathbf{D}(Q_2) \sqsubseteq \mathbf{D}(Q_1 \wedge Q_2)$

Since the conditional is defined as a disjunction (Subsection 2.3.2), Law 4.11 is a direct consequence of Law 4.9.

Law 4.11 (**U** conditional). $\mathbf{U}(P_1 \lhd C \rhd P_2) \sqsubseteq \mathbf{U}(C \wedge P_1) \vee \mathbf{U}(\neg\, C \wedge P_2)$

We established in Theorem 4.4 that **U** is monotonic. Likewise, **D** is monotonic, because existential quantification is monotonic.

Law 4.12 (**D** is monotonic). $Q_1 \sqsubseteq Q_2$ implies $\mathbf{D}(Q_1) \sqsubseteq \mathbf{D}(Q_2)$

Finally, negation does not distribute through **U** or **D**.

Law 4.13 (**U** and negation). $\neg\, \mathbf{U}(P) \sqsubseteq \mathbf{U}(\neg\, P)$

Law 4.14 (**D** and negation). $\mathbf{D}(\neg\, Q) \sqsubseteq \neg\, \mathbf{D}(Q)$

4.2.2 Modelling Inference

A long-standing principle for evaluating the security of a system is commonly known as Shannon's maxim (Shannon, 1949):

> *"the enemy knows the system being used"*

According to this principle, we should assume this "enemy" — in our case, an adversarial user of a system P — possesses complete knowledge of the design of P. The user's complete knowledge of P is reflected by the definition of $\mathbf{U}$: the space of fog behaviours in $\mathbf{U}(P)$ is no larger than the space of observable behaviours in $\mathbf{U}(P)$.

It follows from Shannon's maxim that a user can analyse its interaction ψ with P in tandem with its knowledge of P (and its interface) to infer all behaviours of P that are consistent with ψ. In Chapter 3, we modelled these deductions with the inference function (Definition 3.28). We now formalise a user's inferences about P's behaviour in the lifted space.

Recall from Subsection 3.2.1 that a view V maps behaviours to interactions. We say two behaviours ϕ_1 and ϕ_2 are *indistinguishable* through a view V if and only if ΔV maps both ϕ_1 and ϕ_2 to the same interactions.

Definition 4.15 (Indistinguishability)**.** ϕ_1, ϕ_2 are V-indistinguishable if and only if $\mathbf{L}(V, \phi_1) = \mathbf{L}(V, \phi_2)$

We capture a user's inability to distinguish between behaviours by defining an *indistinguishability relation* over the lifted space. An indistinguishability relation $\mathcal{I}$ relates the observable behaviour ϕ_1 to the fog behaviour $\widetilde{\phi_2}$ (and ϕ_2 to $\widetilde{\phi_1}$) if and only if ϕ_1 and ϕ_2 yield the same interaction to the user. In this way, $\mathcal{I}$ partitions the space of behaviours into equivalence classes.

Definition 4.16 presents a predicate transformer for constructing a (symmetric) indistinguishability relation from V, by reflecting the x_V variables back on themselves.

Definition 4.16 ($\mathbf{IR}$ predicate transformer)**.**

$$\mathbf{IR}(V) \quad \triangleq \quad \Delta(\exists x_V \bullet V \wedge V[\widetilde{x}/x])$$

By imposing $\mathbf{IR}(V)$ upon the lifted space of $\mathbf{U}(P)$, we model a user's inferences about the behaviour of P in terms of the Low-indistinguishable fog behaviours of $\widetilde{P}$. Definition 4.17 introduces a special form of $\mathbf{U}$ that incorporates $\mathbf{IR}$ in this way.

Definition 4.17 (**UI** predicate transformer)**.**

$$\mathbf{UI}\,(V, P) \quad \triangleq \quad \mathbf{U}\,(P) \wedge \mathbf{IR}\,(V)$$

$\mathbf{UI}\,(V, P)$ relates each observable behaviour ϕ of P to *only* those fog behaviours of P that are indistinguishable to ϕ through V. Therefore, it reflects the user's ability to infer all behaviours of P that are consistent with its interaction with P. It works in a similar way to the inference function, but abstracts away from the user's interactions with the process.

The **UI** predicate transformer inherits the characteristics of **U**; it too is an order-embedding from relations to lifted relations.

Example 4.18. Recall the guessing game described in Example 3.30. The indistinguishability relation corresponding to Alice's view is:

$$\mathbf{IR}\,(ALICE) \quad = \quad g = \widetilde{g} \wedge r = \widetilde{r} \wedge g' = \widetilde{g'} \wedge r' = \widetilde{r'}$$

The lifted form of $GUESS_0$ with respect to $g = \widetilde{g} \wedge r' = \widetilde{r'}$ is:

$$\left(\begin{array}{ll} & g \in 1..10 \\ \wedge & n \in 1..10 \wedge \widetilde{n} \in 1..10 \\ \wedge & g = \widetilde{g} \wedge r' = \widetilde{r'} \end{array} \right) \wedge \left(\begin{array}{ll} & g > n \Rightarrow r' > 0 \wedge \widetilde{g} > \widetilde{n} \\ \wedge & g = n \Rightarrow r' = 0 \wedge \widetilde{g} = \widetilde{n} \\ \wedge & g < n \Rightarrow r' < 0 \wedge \widetilde{g} < \widetilde{n} \end{array} \right)$$

This lifted relation indicates that Alice can distinguish between runs where $g > n$, $g = n$ and $g < n$. $\Diamond$

Remark 4.19. We could have defined **UI** to accommodate multiple fog alphabets, in order to represent behavioural indistinguishability with respect to multiple views. However, a single fog alphabet suffices to illustrate the main topics in this chapter, so we avoid cluttering the presentation with multiple fog alphabets.

In the sections that follow, we use views only to construct indistinguishability relations. This enables us to dispense with user interactions and work exclusively in the lifted space of behaviours.

4.3 Encoding Confidentiality

Some of a system's behaviours may entail the communication of secret data. The confidentiality of those data is violated if an adversarial low-level user Low can infer significant information about that data. Hence, regulating Low's inferences about the system's behaviour is crucial to safeguard the confidentiality of those data.

This section presents a scheme for encoding confidentiality properties that fits naturally with the semantics of lifted relations. We first outline the philosophy of this scheme, before explaining how these properties can be applied to evaluate the security of a system design.

4.3.1 Secrets and Cover Stories

Let ϕ denote a behaviour of a system P which is secret in some respect. If P behaves as ϕ, then it is imperative that Low cannot establish that fact, in order to maintain the confidentiality of ϕ.

Our scheme for specifying that ϕ is confidential is based on *cover stories*. Intuitively, a cover story behaviour for ϕ is a designated alternative behaviour of P that lacks the secret aspect of ϕ. The role of these cover story behaviours is to *confuse* Low; they prevent Low from determining when the system's actual behaviour is ϕ. Exactly which behaviours of a system are classed as secret — and which behaviours should be designated as cover stories — depends on the system's functionality and its security policy.

Example 4.20. Consider a military commander who issues the order "attack at dawn". If the attack is a secret, then we may interpret an order to "do not attack" as an appropriate cover story. However, if we suppose that an enemy (Low) knows an attack is inevitable (by observing troop movements) and the secret is the time of the attack, then "attack at noon" or "attack at dusk" may serve as cover stories instead. $\Diamond$

For a behaviour to be an effective cover story for ϕ, it must be Low-indistinguishable from ϕ. It follows that, if a system P features both ϕ and a Low-indistinguishable cover story for ϕ, then Low cannot determine (with certainty) from its interaction when P behaves as ϕ. Hence, if P does behave as ϕ, then the presence of Low-indistinguishable cover story behaviours in P *maintains the confidentiality of* ϕ, because Low cannot rule out the possibility that a cover story behaviour took place instead. We now formalise these concepts in the lifted space.

4.3.2 Obligations

Obligations[1] are our basic unit for expressing confidentiality properties. An obligation is a lifted relation, where the observable space represents secret behaviours and the fog space represents cover story behaviours.

Example 4.21. The obligation:

$$h = 0 \Rightarrow \widetilde{h} > 0$$

specifies that each behaviour where $\widetilde{h} > 0$ serves as a cover story for each behaviour where $h = 0$. ◊

An obligation may relate a behaviour to itself, meaning that behaviour is not confidential. For instance, Example 4.21 maps behaviours where $h \neq 0$ to themselves.

Example 4.22. Consider a program which accepts an integer h as its input and produces an output l' that is computed from h (in an unspecified manner). Given that Low views l' but not h, we may define some obligations to model certain confidentiality properties over the value of h:

$$\begin{aligned}
\theta_1 &\triangleq \widetilde{h} \neq h \\
\theta_2 &\triangleq \widetilde{h} \,\mathsf{mod}\, 2 \neq h \,\mathsf{mod}\, 2 \\
\theta_3 &\triangleq h \,\mathsf{mod}\, 2 = 1 \Rightarrow \widetilde{h} \,\mathsf{mod}\, 2 = 0 \\
\theta_4 &\triangleq h = 0 \Rightarrow \widetilde{h} \neq h
\end{aligned}$$

[1]The term "obligation" is borrowed from Seehusen and Stølen (2007, 2009), but we ascribe a different meaning to the term.

θ_1 specifies that every value of h is secret, while every non-equal value of $\widetilde{h}$ is a cover story. In short, θ_1 specifies that Low must be unable to establish the exact value of h. θ_2 specifies that Low cannot determine the parity of h. θ_3 specifies that any even value of h is a cover story for any odd value of h. Finally, θ_4 specifies that Low must be unable to determine that $h = 0$. $\lozenge$

Remark 4.23. In this chapter, we focus on specifying confidentiality properties over the whole behaviour of the system. However, it is sometimes desirable to limit Low's inferences about a high-level (High) user's interactions with a system. Given a relation θ_H — defined over High's interaction variables x_H, x'_H and fog variables $\widetilde{x_H}, \widetilde{x'_H}$ — the obligation:

$$\forall\, x_H, x'_H, \widetilde{x_H}, \widetilde{x'_H} \bullet \Delta\,(H \wedge H[\widetilde{x}, \widetilde{x_H}/x, x_H]) \Rightarrow \theta_H$$

projects θ_H through H to yield an obligation over x, x' and $\widetilde{x}, \widetilde{x'}$.

4.3.3 Conformance

Let P denote a relation and θ denote an obligation. Informally, P *conforms* to θ with respect to a view L if, for each behaviour ϕ exhibited by P, there exists *at least one* behaviour $\widetilde{\phi}$ of $\widetilde{P}$ such that:

- $\widetilde{\phi}$ is indistinguishable through L from ϕ; and
- θ prescribes $\widetilde{\phi}$ as a cover story for ϕ.

If P fails to conform to θ, then Low could rule out all cover stories for a particular behaviour of P. In this case, we interpret P as being able to leak secret information to Low.

Definition 4.24 codifies this conformance condition.

Definition 4.24 (Conformance).

$$P \propto_L \theta \quad \triangleq \quad [\,P \quad \Rightarrow \quad \mathbf{D}\,(\mathbf{UI}\,(L, P) \wedge \theta)\,]$$

Example 4.25. The following table evaluates a selection of candidate programs against the obligations listed by Example 4.22 according to the conformance condition, assuming Low's view LV is defined as $l_L = l$.

Program	θ_1	θ_2	θ_3	θ_4
$l' = h$	$\times$	$\times$	$\times$	$\times$
$l' = 0$	✓	✓	✓	✓
$l' = h \bmod 2$	✓	$\times$	$\times$	✓
$l' = \lfloor h/2 \rfloor$	✓	✓	✓	✓
$l' = h \vee l' = 0$	$\times$	$\times$	✓	✓

(Each ✓ indicates $P \propto_L \theta$ and each $\times$ indicates otherwise.)

The program $l' = h$ allows Low to establish the exact value of h, so it violates each of the obligations. Conversely, the program $l' = 0$ reveals no information about h to Low, so it conforms to all the obligations.

The program $l' = h \bmod 2$ enables Low to distinguish between $h \in \{0, 2\}$ and $h \in \{1, 3\}$, so Low can determine the parity of h, in violation of θ_2 and θ_3. The program $l' = \lfloor h/2 \rfloor$ enables Low to distinguish between $h \in \{0, 1\}$ and $h \in \{2, 3\}$, but this does not violate any of the obligations.

The program $l' = h \vee l' = 0$ gives Low perfect information about h when $l' \in \{1, 2, 3\}$. However, when $l' = 0$, Low cannot determine any information about the value of h. $\Diamond$

Obligations need to be defined with respect to the UTP theory in which a system is modelled. However, they may be defined without making presumptions about the system's functionality.

4.3.4 The Lattice of Obligations

The ordering over obligations is simply the refinement ordering. If $\theta_1 \sqsubseteq \theta_2$, then for each behaviour ϕ, θ_2 prescribes no more cover stories for ϕ as does θ_1. Therefore, if a relation P conforms to θ_2, then P must also conform to θ_1.

Lemma 4.26 ($\sqsubseteq$ is closed under $\propto_L$)**.**

$$\theta_1 \sqsubseteq \theta_2 \wedge P \propto_L \theta_2 \quad \text{implies} \quad P \propto_L \theta_1$$

Example 4.27. Recalling Example 4.22, we have $\theta_3 \sqsubseteq \theta_2$ and $\theta_4 \sqsubseteq \theta_1$. It follows that each program in Example 4.25 conforming to θ_2 also conforms to θ_3, and each program conforming to θ_1 conforms to θ_4. $\diamondsuit$

The refinement ordering induces a complete lattice over the space of obligations, with a unique top and bottom. Every system conforms to the weakest obligation **true**, but only the miracle **false** conforms to the strongest obligation **false**.

We denote a set of obligations by Θ. The least upper bound and greatest lower bound of Θ are standard.

Definition 4.28 (obligation lub)**.** $\bigsqcup \Theta \triangleq \bigwedge \Theta$

Definition 4.29 (obligation glb)**.** $\bigsqcap \Theta \triangleq \bigvee \Theta$

Even if a system P conforms to θ_1 and θ_2 individually, P need not conform to $\theta_1 \wedge \theta_2$, as Example 4.30 shows.

Example 4.30. Suppose θ_1 and θ_2 prescribe mutually exclusive cover stories: for instance, $\theta_1 = \widetilde{h} = 1$ and $\theta_2 = \widetilde{h} = 2$. Then $\theta_1 \wedge \theta_2 =$ **false**. $\diamondsuit$

It follows from Lemma 4.26 that, if a system conforms to $\theta_1 \wedge \theta_2$, then it conforms to θ_1 and θ_2 separately.

Law 4.31 (lub conformance)**.** $P \propto_L (\theta_1 \wedge \theta_2) \quad \text{implies} \quad P \propto_L \theta_1 \wedge P \propto_L \theta_2$

There is a dual law for the greatest lower bound of obligations, which also follows from Lemma 4.26.

Law 4.32 (glb conformance)**.** $P \propto_L \theta_1 \vee P \propto_L \theta_2 \quad \text{implies} \quad P \propto_L (\theta_1 \vee \theta_2)$

4.3.5 Multi-Obligations

The notion of conformance is rather weak: it demands that Low is unable to rule out at least one cover story behaviour — out of possibly many cover stories that an obligation prescribes — for each behaviour of a system. However, confidentiality properties often demand that Low cannot rule out *any* cover story drawn from a set of behaviours, which is a more severe imposition.

We now define an extension of obligations to encode properties such as these. Following the style of Mantel (2003), we specify a *multi-obligation* (σ, τ) as a pair:

- The *obligation template* τ is an obligation with alphabet including *auxiliary variables*.
- The restriction σ is a predicate transformer; given a system design P, $\sigma(P)$ is a predicate over the auxiliary variables.

The purpose of σ is to specify a range of values to fill the auxiliary variables of the obligation template. Hence, a multi-obligation (σ, τ) encodes a set of obligations, where each valuation of the auxiliary variables satisfying $\sigma(P)$ is substituted into τ to instantiate a separate obligation.

Example 4.33. The multi-obligation $(\lambda\, P \bullet i \in S, \widetilde{h} = i)$ specifies that Low must be unable to rule out each of the individual values in S for h. It encodes the set of obligations $\{\widetilde{h} = i \mid i \in S\}$. $\Diamond$

Definition 4.34 extends the conformance condition to multi-obligations. A relation P conforms to (σ, τ) if and only if P conforms to each σ-instantiation of τ. We assume the alphabet of P is disjoint from the auxiliary variables named by σ and τ; if they are not, then we may rename those auxiliary variables to resolve such clashes.

Definition 4.34 (Multi-obligation conformance)**.**

$$P \propto_L (\sigma, \tau) \quad \triangleq \quad [\, P \wedge \sigma(P) \quad \Rightarrow \quad \mathbf{D}\,(\mathbf{UI}\,(L, P) \wedge \tau)\,]$$

Typically, we specify σ to be a constant function. Nevertheless, some kinds of confidentiality properties are sensitive to the details of a system's behaviours. These properties can be encoded by defining σ as a predicate transformer over the space of systems.

Definition 4.35 presents one possible formulation of the least upper bound of an indexed set $\{(\sigma_1\,,\,\tau_1)\,,\ldots,(\sigma_n\,,\,\tau_n)\}$ of multi-obligations.

Definition 4.35 (Multi-obligation least upper bound)**.**

$$\bigsqcup i \bullet (\sigma_i\,,\,\tau_i) \triangleq \left(\lambda\, P \bullet \bigwedge i \bullet j = i \Rightarrow \sigma_i(P)\,,\, \bigwedge i \bullet j = i \Rightarrow \tau_i\right)$$

where j is a fresh variable that is free in each σ_i and θ_i.

Since the indexing variable is part of the alphabet of the obligation template and restriction of the multi-obligation $\bigsqcup i \bullet (\sigma_i\,,\,\tau_i)$, it is quantified by the universal closure of the multi-obligation conformance condition (Definition 4.34). Hence, we have:

$$P \propto_L \bigsqcup i \bullet (\sigma_i\,,\,\tau_i) \quad \text{if and only if} \quad \forall\, i \bullet P \propto_L (\sigma_i\,,\,\tau_i)$$

4.4 Deriving Secure Specifications

So far, we have treated obligations (i.e. specifications of confidentiality) separately from specifications of the functionality of systems. In this section, we show how confidentiality properties can be combined with a functional specification, to realise a *secure* specification of a system.

4.4.1 Reconciling Obligations

Let FC denote the combination of a lifted functionality specification of the form $\mathbf{UI}\,(L,P)$ and an obligation θ:

$$FC \quad = \quad \mathbf{UI}\,(L,P) \wedge \theta$$

FC maps each behaviour ϕ of P to only those fog behaviours of $\mathbf{UI}\,(L,P)$ that θ lists as cover stories for ϕ. If P conforms to θ, then each behaviour of P is present in FC. On the other hand, if P does not conform to θ, then

P features at least one behaviour ϕ that $\mathsf{UI}(L, P)$ associates with no fog behaviour that θ prescribes as a cover story for ϕ. Hence, a contradiction is induced in FC which renders the behaviour ϕ infeasible.

Example 4.36. Recall the program $l' = h \vee l' = 0$ from Example 4.25 and the obligation $\theta_1 \triangleq \widetilde{h} \neq h$ from Example 4.22. By lifting the program (with respect to Low's view LV) and conjoining θ_1, we derive:

$$\begin{array}{lll}
& \mathsf{UI}(LV, l' = h \vee l' = 0) \wedge \widetilde{h} \neq h & \\
= & (l' = h \vee l' = 0) \wedge (\widetilde{l'} = \widetilde{h} \vee \widetilde{l'} = 0) \wedge l' = \widetilde{l'} \wedge \widetilde{h} \neq h & [\text{def } \mathsf{UI}] \\
= & ((l' = h \wedge l' = \widetilde{h}) \vee l' = 0) \wedge l' = \widetilde{l'} \wedge \widetilde{h} \neq h & [\text{prop calc}] \\
= & (\mathbf{false} \vee l' = 0) \wedge l' = \widetilde{l'} \wedge \widetilde{h} \neq h & [\text{contradiction}] \\
= & l' = 0 \wedge l' = \widetilde{l'} \wedge \widetilde{h} \neq h & [\text{prop calc}]
\end{array}$$

Here, each behaviour of the original program where $l' \neq 0$ — revealing the value of h to Low — is made infeasible. ◊

The purpose of combining a functional system design (a lifted relation) with a confidentiality specification (an obligation) is to disable functionality that potentially reveals a secret to Low. A remarkable consequence of this combination is that it *makes the system design secure*.

In extreme cases, P's functionality may be irreconcilable with θ. Then, FC admits *no behaviour whatsoever*, as Example 4.37 demonstrates.

Example 4.37. Applying θ_1 to the lifted form of program $l' = h$ yields:

$$\begin{array}{lll}
& \mathsf{UI}(LV, l' = h) \wedge \widetilde{h} \neq h & \\
= & l' = h \wedge \widetilde{l'} = \widetilde{h} \wedge l' = \widetilde{l'} \wedge \widetilde{h} \neq h & [\text{def } \mathsf{UI}] \\
= & \mathbf{false} & [\text{contradiction}]
\end{array}$$

Since $l' = h$ always reveals h to Low, no cover story for $l' = h$ prescribed by θ_1 remains in the fog space. With no supporting fog behaviours, $l' = h$ itself is rendered infeasible. ◊

Indeed, it is entirely appropriate for the combination of irreconcilable specifications of functionality and confidentiality to be a miracle, because only a miracle could ever satisfy both specifications.

4.4.2 Fixed Points of Confidentiality

An anomaly induced by conjoining $\mathsf{UI}\,(L, P)$ with θ is that observable behaviours made infeasible by θ are not necessarily made infeasible in the fog space. This anomaly is illustrated by Example 4.38.

Example 4.38. Consider a very simple system $h = 0 \vee h = 1$ and the obligation $\theta = h \neq \widetilde{h} \wedge (h = 0 \Rightarrow \widetilde{h} = 2)$. We have:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{UI}\,(\mathbf{true}, h = 0 \vee h = 1) \wedge h \neq \widetilde{h} \wedge (h = 0 \Rightarrow \widetilde{h} = 2)} \\
= & \mathsf{U}\,(h = 0 \vee h = 1) \wedge h \neq \widetilde{h} \wedge (h = 0 \Rightarrow \widetilde{h} = 2) & [\text{def } \mathsf{UI}] \\
= & (h = 0 \wedge \widetilde{h} = 1 \vee h = 1 \wedge \widetilde{h} = 0) \wedge (h = 0 \Rightarrow \widetilde{h} = 2) & [\text{def } \mathsf{U}] \\
= & h = 1 \wedge \widetilde{h} = 0 & [\text{prop calc}]
\end{array}$$

Here, the behaviour $h = 0$ is infeasible, but its counterpart $\widetilde{h} = 0$ remains in the fog space. It is as though Low cannot rule out the infeasible $h = 0$, which is incompatible with our expectation that Low knows the system's construction (Subsection 4.2.2). $\Diamond$

To rectify this anomaly, we define a new predicate transformer C over the space of lifted relations.

Definition 4.39 (C predicate transformer)**.**

$$\mathsf{C}\,(Q) \quad \triangleq \quad Q \wedge \widetilde{\mathsf{D}\,(Q)}$$

The result of applying C to Q is a lifted relation that allows all observable behaviours of Q, but makes infeasible all fog behaviours that do not correspond to any of those observable behaviours.

Lemma 4.40 (C is monotonic)**.** $Q_1 \sqsubseteq Q_2$ implies $\mathsf{C}\,(Q_1) \sqsubseteq \mathsf{C}\,(Q_2)$

$\mathbf{C}$ is not idempotent. Its application to Q has the potential to make yet other observable behaviours of Q infeasible, if the fog counterparts of those behaviours are excluded from the fog space of $\mathbf{C}(Q)$.

Example 4.41. Let $Q = h > 0 \land \widetilde{h} > 0$ and $\theta = \widetilde{h} = h - 1$. Then:

$$FC \quad = \quad Q \land \theta \quad = \quad h > 1 \land \widetilde{h} = h - 1$$

Repeatedly applying $\mathbf{C}$ to FC reveals a pattern:

$$\begin{array}{rcccl}
\mathbf{C}(FC) & = & h > 1 \land \widetilde{h} = h - 1 \land \widetilde{h} > 1 & = & h > 2 \land \widetilde{h} = h - 1 \\
\mathbf{C}^2(FC) & = & h > 2 \land \widetilde{h} = h - 1 \land \widetilde{h} > 2 & = & h > 3 \land \widetilde{h} = h - 1 \\
\mathbf{C}^n(FC) & = & h > n + 1 \land \widetilde{h} = h - 1 & &
\end{array}$$

The application of $\mathbf{C}$ to FC does not converge because $\mathbf{C}^n(FC) \sqsubset \mathbf{C}^{n+1}(FC)$ for $n \geq 0$. In the limit, we obtain **false**: every behaviour is made miraculous. $\Diamond$

Let $\mathbf{CC}(Q)$ denote the fixed point of $\mathbf{C}$ applied to Q.

Definition 4.42 ($\mathbf{CC}$ predicate transformer)**.**

$$\mathbf{CC}(Q) \quad \triangleq \quad \mu X \bullet \mathbf{C}(Q \land X)$$

Since $\mathbf{C}$ is monotonic (Lemma 4.40) and the lifted space is a complete lattice, the Knaster-Tarski theorem (Tarski, 1955) implies $\mathbf{CC}(Q)$ is well-defined for every point Q in the lifted space. Unlike $\mathbf{C}$, $\mathbf{CC}$ is idempotent.

Lemma 4.43 ($\mathbf{CC}$ is idempotent)**.** $\mathbf{CC}(\mathbf{CC}(Q)) = \mathbf{CC}(Q)$

The idempotence of $\mathbf{CC}$ is significant. Let $Y = \mathbf{CC}(\mathbf{UI}(L, P) \land \theta)$. By definition, the fog space of Y is no larger than its observable space. In particular, the observable space of Y contains only behaviours that θ maps to at least one cover story in the fog space of Y. It follows that projecting Y back to the unlifted space yields a relation $\mathbf{D}(Y)$ that conforms to θ.

Theorem 4.44 (Fixed point of confidentiality)**.**

$$\mathbf{D}\left(\mathbf{CC}\left(\mathbf{UI}\left(L,P\right),\theta\right)\right) \propto_L \theta$$

4.4.3 Reconciling Multi-Obligations

Since multi-obligations are not defined as predicates, they cannot be conjoined with lifted system designs in the same manner as obligations. The procedure for conjoining obligations can be extended to multi-obligations, by treating each instantiation of a multi-obligation as an obligation in its own right.

Given a multi-obligation $(\sigma\,,\,\tau)$ and a relation P, we can instantiate the obligation template τ with a valuation of the auxiliary variables emitted by $\sigma(P)$. The instantiated obligation θ can be reconciled against the lift of P, to identify a lifted relation Q (where $P \sqsubseteq \mathbf{D}\left(Q\right)$) such that $\mathbf{D}\left(Q\right)$ conforms to θ. Then, the procedure can be repeated with $\mathbf{D}\left(Q\right)$ in place of P and instantiations of τ generated from $\sigma(\mathbf{D}\left(Q\right))$.

This procedure needs to be applied until a relation P_ω is identified that conforms to each $\sigma(P_\omega)$-instantiation of τ. In the worst case, the derivation of P_ω may be extremely long-winded. For reasons described in Section 4.5, $P_1 \propto_L \theta$ and $P_1 \sqsubseteq P_2$ do not together imply $P_2 \propto_L \theta$, so the procedure needs to be restarted whenever P_1 is replaced by P_2.

4.5 Refinement and Confidentiality

Refinement steps that reduce non-determinism may fail to preserve confidentiality properties. Example 4.45 demonstrates how refinement can introduce insecurity into a system design.

Example 4.45. Recall from Example 4.25 the program $l' = h \vee l' = 0$ and the obligation $\theta_4 \triangleq h = 0 \Rightarrow \widetilde{h} \neq h$, where $l' = h \vee l' = 0$ conforms to θ_4. While $l' = h \vee l' = 0$ is refined by $l' = h$, the program $l' = h$ does not conform to θ_4 (Example 4.25), because Low can establish $h = 0$ if it observes $l' = 0$. ◊

This problem was first noted in print by Jacob (1989a) and was subsequently termed the "refinement paradox" by Roscoe (1995). This term is perhaps misleading, because the "paradox" is readily explained. By resolving non-determinism, refinement resolves uncertainty about the behaviours of a system, so Low's inferences about those behaviours are *never weakened* but *may be strengthened*. Therefore, naïve refinement steps may increase Low's inferences beyond the upper limit on information flow imposed by a confidentiality property, thus violating the property.

Roscoe (1995) argues the root of the so-called paradox lies in the fact that formal specification notations typically draw no distinction between two different roles that non-determinism may serve:

Under-specification is "don't-care" non-determinism, providing software engineers with freedom to choose how to implement a specification.

Unpredictability is non-determinism intended to limit Low's inferences about secret information.

It is safe to refine away under-specification non-determinism within a specification. However, care is needed when reducing the unpredictability of a system design, in order not to jeopardise confidentiality properties.

Our formulation of confidentiality in the lifted space naturally distinguishes between under-specification and unpredictability. By introducing a secondary observation space to model Low's inferences, we become able to reason about unpredictability at the semantic level. Furthermore, the effect of conjoining an obligation θ to a lifted relation Q — in the style of Section 4.4 — is to specify that P's behaviour can be no more predictable (to Low) than θ allows.

Example 4.46. Returning to the insecure refinement described in Example 4.45, we now attempt the same refinement in the lifted space.

$$\begin{aligned} &\mathbf{UI}\,(LV, l' = h \lor l' = 0) \land (h = 0 \Rightarrow \widetilde{h} \neq h) \\ \sqsubseteq\quad &\mathbf{UI}\,(LV, l' = h) \land (h = 0 \Rightarrow \widetilde{h} \neq h) && \text{[order embedding]} \end{aligned}$$

$$= \quad \textsf{\textbf{UI}}\,(LV, l' = h) \wedge (h = 0 \Rightarrow \widetilde{h} \neq h \wedge \widetilde{h} = h) \qquad [\text{def } \textsf{\textbf{UI}}]$$
$$= \quad \textsf{\textbf{UI}}\,(LV, l' = h) \wedge (h = 0 \Rightarrow \textbf{false}) \qquad [\text{contradiction}]$$
$$= \quad \textsf{\textbf{UI}}\,(LV, l' = h) \wedge h \neq 0 \qquad [\text{prop calc}]$$

Since the program fails to provide the cover story $\widetilde{h} \neq h$ when $h = 0$, the obligation excludes behaviours where $h = 0$ from taking place. $\Diamond$

The so-called refinement paradox evaporates in the consistent lifted space, because refining a lifted system with embedded obligations cannot make the system behave in a way that violates those obligation. Since obligations are never weakened by refinement (by Lemma 4.26), they remain within the system design at each step of its development.

4.6 Specifying Noninterference Properties

This section examines how many information flow security properties defined in the literature can be formulated within the framework. By doing so, we demonstrate that our framework is sufficiently expressive for capturing these recognised notions of confidentiality.

A side-effect of this work is that we link our UTP formulation of confidentiality properties to the body of existing work on information flow properties. This link could potentially be used to port existing results from the literature to our framework.

4.6.1 Noninterference

The canonical information flow property is *noninterference* (Goguen and Meseguer, 1982, 1984). Noninterference mandates that High's inputs to a system must have no effect on the system's outputs to Low; in other words, Low must be unable to rule out any possible High interaction with the system (or even that High has interacted with the system at all). Therefore, a system satisfying noninterference does not disclose to Low any information about High's interaction.

In Goguen and Meseguer's original definition of noninterference, a system is modelled as a deterministic state machine, where outputs to Low are a function of the inputs from High and Low. This model allows analysis of whether High's choice of inputs can influence the outputs visible to Low. However, this model does not support reasoning about non-deterministic systems. Moreover, this model also requires the system to be *input total*: in every state, the system must be willing to accept every input provided by the environment. This requirement is unrealistic in practice, since a system with a finite memory can only buffer a finite series of inputs pending computation.

Despite the limitations of its original definition, the notion of noninterference has been highly influential in theoretical studies of information flow security. A multitude of noninterference-like properties have been defined to replicate the intent of noninterference in a non-deterministic setting. Typically, these properties are formulated in a trace semantics, where the interactions between a system and its users are recorded as events within the trace. These properties include, but are not limited to, the following:

Noninference (O'Halloran, 1990) requires that for every trace of P featuring a high-level event, there is a Low-indistinguishable trace featuring no high-level events.

Generalised noninference (McLean, 1994a) is similar to noninference, but is weaker: it requires only that for every trace of P featuring a high-level input event, there is a Low-indistinguishable trace featuring no high-level input events.

Generalised noninterference (McCullough, 1987) requires that perturbing a trace by changing high-level inputs does not imply a change in subsequent low-level events.

Forward correctability (Johnson and Thayer, 1988) requires that a perturbation of a trace that inserts or deletes a high-level input event can be rectified by inserting or deleting high-level output events.

Separability (McLean, 1994a) is an extremely strict property, which prohibits any information flow between High and Low. A system which enforces separability between High and Low is equivalent to two physically separate ("air-gapped") sub-systems, where one sub-system interacts with High alone and the other with Low alone.

Separability is based on a proposal by Rushby (1981) for verifying security, which involves cutting known communication channels between different components of the system and establishing the resulting system can be split into separate components.[2]

4.6.2 Worked Example: Basic Security Predicates

Several frameworks for expressing a range of information flow properties in a uniform manner have been presented in the security literature (Jacob, 1991; McLean, 1994a; Focardi and Gorrieri, 1995; Zakinthinos and Lee, 1997; Mantel, 2000b). The objective of these frameworks is to consolidate the existing definitions of noninterference-like properties in the literature, in order to evaluate and compare these properties rigorously and to enable new information flow properties to be formalised.

These frameworks formulate confidentiality properties as closure conditions over an observation space (typically trace sets). We hypothesise that multi-obligations are sufficiently expressive to encode these closure conditions. We aim to substantiate this hypothesis by demonstrating how the closure conditions from one such framework can be encoded in our own framework.

The *Modular Framework for Information Flow Properties* (MAKS), introduced by Mantel (2000b) (as the *Modular Assembly Kit for Security*) and later elaborated by Mantel (2003), is capable of expressing a wide range of noninterference-like properties from the security literature in a uniform trace-based style. It defines a collection of *basic security predicates* (BSPs) to express a variety of transformations on high-level components of sys-

[2]Jacob (1990) observed this proposal is subtly flawed, because cutting known channels may hide the presence of undesired covert channels.

tem traces, capturing low-level users' uncertainty about high-level inputs and outputs. These BSPs are closure conditions that share a common structure that is similar to our structuring of multi-obligations. Therefore, we can translate the BSPs defined by Mantel (2003) into multi-obligations, in the context of the UTP theory of reactive processes.

In the MAKS, the events communicated between a system and its users are partitioned into three sets:

- $\mathcal{V}$, denoting non-confidential events visible to Low;
- $\mathcal{N}$, denoting non-confidential events not visible to Low; and
- $\mathcal{C}$, denoting confidential events not visible to Low.

We identify $\mathcal{V}$ with Low's window and identify $\mathcal{C}$ with High's window. The BSPs assume that Low can observe a projection of the system trace, so a suitable choice of Low's view would be a **VHR**-healthy view (Definition 3.41) constructed from window $\mathcal{V}$.

Example 4.47 presents multi-obligation encodings of a selection of BSPs from the MAKS. We summarise the meaning of these BSPs as follows:

- R (removal) demands that each trace of P that contains confidential events can be *perturbed* — by deleting all confidential events — to reach another trace of P.
- BSD (backwards strict deletion) demands that each trace tr of P that contains a confidential event can be perturbed — by deleting the final confidential event c in tr but not changing the events leading up to c — to reach another trace of P.
- BSI (backwards strict insertion) is the dual of BSD. It demands that each trace tr of P can be perturbed by inserting any confidential event c after all other confidential events in tr, but not changing the events leading up to the inserted c.
- $BSIA^{pc}$ (backwards strict insertion of admissible events) is a weaker form of BSI, which applies only to those traces of P that can be

Example 4.47 (Reformulations of BSPs as multi-obligations). Let $\alpha \sim_{\mathcal{V}} \widetilde{\alpha} \triangleq \alpha \restriction \mathcal{V} = \widetilde{\alpha} \restriction \mathcal{V}$ in:

$$R(\mathcal{V},\mathcal{N},\mathcal{C}) \triangleq \left(\lambda\, P \bullet \textbf{true}\,,\, (\widetilde{tr'} - \widetilde{tr}) \restriction \mathcal{C} = \langle\rangle\right)$$

$$BSD(\mathcal{V},\mathcal{N},\mathcal{C}) \triangleq \left(\lambda\, P \bullet \begin{pmatrix} c \in \mathcal{C} \\ \wedge\ (tr' - tr) = \beta \frown \langle c\rangle \frown \alpha \\ \wedge\ \alpha \restriction \mathcal{C} = \langle\rangle \end{pmatrix},\ \exists\, \widetilde{\alpha} \bullet \begin{pmatrix} (\widetilde{tr'} - \widetilde{tr}) = \beta \frown \widetilde{\alpha} \\ \wedge\ \widetilde{\alpha} \sim_{\mathcal{V}} \alpha \\ \wedge\ \widetilde{\alpha} \restriction \mathcal{C} = \langle\rangle \end{pmatrix}\right)$$

$$BSI(\mathcal{V},\mathcal{N},\mathcal{C}) \triangleq \left(\lambda\, P \bullet \begin{pmatrix} c \in \mathcal{C} \\ \wedge\ (tr' - tr) = \beta \frown \alpha \\ \wedge\ \alpha \restriction \mathcal{C} = \langle\rangle \end{pmatrix},\ \exists\, \widetilde{\alpha} \bullet \begin{pmatrix} (\widetilde{tr'} - \widetilde{tr}) = \beta \frown \langle c\rangle \frown \widetilde{\alpha} \\ \wedge\ \widetilde{\alpha} \sim_{\mathcal{V}} \alpha \\ \wedge\ \widetilde{\alpha} \restriction \mathcal{C} = \langle\rangle \end{pmatrix}\right)$$

$$BSIA^{pc}(\mathcal{V},\mathcal{N},\mathcal{C}) \triangleq \left(\lambda\, P \bullet \begin{pmatrix} c \in \mathcal{C} \\ \wedge\ (tr' - tr) = \beta \frown \alpha \\ \wedge\ \alpha \restriction \mathcal{C} = \langle\rangle \\ \wedge\ \begin{array}{l} \exists\, x, x' \bullet P \\ \quad \wedge\ (tr' - tr) = \beta \frown \langle c\rangle \end{array} \end{pmatrix},\ \exists\, \widetilde{\alpha} \bullet \begin{pmatrix} (\widetilde{tr'} - \widetilde{tr}) = \beta \frown \langle c\rangle \frown \widetilde{\alpha} \\ \wedge\ \widetilde{\alpha} \sim_{\mathcal{V}} \alpha \\ \wedge\ \widetilde{\alpha} \restriction \mathcal{C} = \langle\rangle \end{pmatrix}\right)$$

◇

Information flow property	Multi-obligation
Noninference	$R(H)$
Generalised noninference	$R(HI)$
Generalised noninterference	$BSD(HI) \sqcup BSI(HI)$
Separability (over P)	$BSD(H) \sqcup BSIA^{pc}(H)$

Table 4.1: MAKS definition of noninterference properties from Mantel (2003), where $H = (\mathcal{L}, \emptyset, \mathcal{H})$, $HI = (\mathcal{L}, \mathcal{H} \setminus \mathcal{I}, \mathcal{H} \cap \mathcal{I})$ and $\mathcal{I}$ is a subset of $\mathcal{V} \cup \mathcal{N} \cup \mathcal{C}$ denoting input events.

constructed by appending a confidential event to a trace in P. This BSP needs to be initialised with the specification of P, to filter the traces satisfying this condition.

The definition of conformance implies the perturbations of each trace *tr* must themselves be Low-indistinguishable from *tr*.

The BSPs of MAKS can be conjoined to express many (but not all) of the noninterference-like information flow properties defined in the literature. Table 4.1 reproduces the MAKS definitions of the noninterference-like properties described in Subsection 4.6.1, but in terms of the least upper bounds of the multi-obligations in Example 4.47. Full justifications of these definitions are given by Mantel (2003).

4.7 Related Work

Since the advent of the noninterference property, many researchers have studied the problem of restricting information flow from systems to their users. This section surveys some of their work, and compares it with the approach presented in this chapter.

4.7.1 The Limitations of Noninterference

Information flow properties based on noninterference typically require that a system does not leak any information about High's activities to Low. This blanket security policy often conflicts with a system's functional requirements, which may require High to be able to communicate non-confidential data to Low through the system. In these cases, noninterference from High to Low is simply not appropriate.

No consensus has been reached in the literature about which nondeterministic formulations of noninterference are most suitable for expressing the security requirements of realistic information systems (Ryan, 2001). Indeed, the strictness of noninterference-like properties has led researchers such as Ryan et al. (2001) to question whether these properties are relevant to secure software development.

The position elaborated in this thesis is that software engineers should be able to specify custom confidentiality properties that are tailored to the intricacies of the system domain, rather than choosing from a limited range of ready-made information flow security properties.

4.7.2 Security Specifications

Our notion of obligations for specifying confidentiality has much in common with the *security specifications* defined by Jacob (1988).

A security specification f is a function from Low's interactions to sets of system behaviours. A system P satisfies f if, for each interaction ψ_L with P that Low can make, $f(\psi_L)$ is a subset of Low's inference set for ψ_L.

There is a duality between obligations and security specifications: f specifies the maximum information that Low *may establish* about the system's behaviour; whereas an obligation specifies the minimum information that Low *cannot rule out* about the system's behaviour.

Security specifications do not incorporate the notion of cover story behaviours. Hence, if we wish to specify that $\widetilde{\phi}$ is a cover story for ϕ using a security specification, then it is necessary that ϕ serves as a cover story for $\widetilde{\phi}$ as well. Moreover, we can define obligations that offer a choice of which cover stories are present in a system, whereas security specifications provide no such choice.

4.7.3 Other Approaches

This chapter is primarily concerned with one flavour of confidentiality properties. Nevertheless, a variety of alternative approaches towards information flow security have been studied extensively in the literature. These approaches have advantages over our confidentiality framework, but they also have drawbacks.

Active and Passive Flows Jacob (1991) and Roscoe (1997) identify two scenarios for information leakage between users. These are *passive flows*, where Low tries to infer information about the system's behaviour without the help of other users; and *active flows*, where a treacherous High user tries to leak secret information to Low by interacting with the system according to a pre-arranged protocol. The framework set out in this chapter is intended to counter passive flows, but it does not guarantee the prevention of active flows. Nevertheless, if such a protocol could be established by High and Low then, in many situations, High could pass secrets to Low outside the boundaries of the system anyway.

Quantifying Flow Small leaks of data from high-level users to low-level users are often tolerable (and sometimes unavoidable) in realistic systems, provided that a low-level user cannot infer any significant details about confidential data from such a leak. However, our notion of confidentiality is *qualitative*: an obligation is violated if the slightest amount of data deemed secret by a confidentiality property is leaked to Low. This effect motivates defining confidentiality properties in terms of a *quantitative* measure of information flow between users. Recent research has addressed quantitative confidentiality properties using information theory (Shannon, 1948) as a foundation; Mu (2008) surveys this work and its applications.

Probability Another limitation of our treatment of confidentiality properties is that we do not address the probability distribution of a system's behaviours. This shortcoming could lead to serious security breaches, because if Low has knowledge of this distribution, then it may be able to deduce confidential data with near certainty but without violating the confidentiality property.

Some research has investigated *probabilistic* confidentiality properties which account for the likelihood of alternative high-level activities (Santen, 2006, 2008). While these properties are attractive in theory, their application in practice is not without difficulty: the probability distribution of a system's behaviours may be unknown. Even if that distribution is known, it may be intractable to determine whether a non-trivial system model satisfies a probabilistic confidentiality property (Ryan, 2001).

The security of confidential information need not rest on technical measures alone. Measures such as access control and inference control are complemented by physical security mechanisms for computer terminals, such as barriers and surveillance systems (Anderson, 2001).

Sociological factors also have a role to play in system security: for instance, ethics training or the threat of punishment may deter users from attempting to access confidential information without the appropriate clearance (Jacob, 1989a; Workman and Gathegi, 2007).

4.7.4 Refinement and Confidentiality

Most notions of confidentiality-preserving refinement in the literature are realised in two ways:

1. by limiting the space of confidentiality properties to those that are *refinement closed* — that is, if a system satisfies such a property, then all refinements of the system will also satisfy that property; or

2. by strengthening the refinement relation to ensure it preserves confidentiality properties in specifications.

Obligations are refinement closed in the lifted space. The method proposed in Section 4.4 for deriving a secure specification of a system P is refinement-closed. If P is refined in an insecure fashion with respect to θ, then that insecurity is manifested when the lifted form of P is conjoined with θ. Therefore, the notion of confidentiality-preserving refinement described in Section 4.5 is just the standard notion of refinement in the UTP, but over the space of lifted relations.

Roscoe et al. (1994) describe a noninterference-like property over CSP processes in the line of the first approach. The property requires Low's interface to a process — formulated using the concept of lazy abstraction (described in Subsection 3.5.3) — to be deterministic. If this property is met — as can be verified with a model checker — then noninterference is assured, because High's interaction with the process cannot influence Low's interactions in any way. Furthermore, no (failures-divergences) refinement of the process can induce new sources of information flow from High to Low, because Low's interface must remain deterministic. While this property is theoretically appealing, Ryan (2001) argues that it is unsuitable for a large class of systems which contain unavoidable non-determinism in their outputs to Low.

Seehusen and Stølen (2007, 2009) have proposed a further instance of the first approach. They advocate structuring a system specification as a set of trace sets. Each trace set represents under-specification, while the presence of multiple trace sets within a specification represents unpredictability. Traces can be refined away from the specification, so long as doing so makes no trace set in the specification empty. Indeed, this notion of refinement generalises the structure of a confidentiality refinement relation defined by Jacob (1992).

Instances of the second approach that are specialised to particular confidentiality properties are detailed by Mantel (2001) and Alur et al. (2006), among others. A generic approach for strengthening refinement to preserve any given confidentiality property is described by Banks and Jacob (2010a). In that work, the standard refinement ordering is combined with a confidentiality ordering, which dictates that refinement does not strengthen Low's inferences about behaviours of a system marked as secret. This notion of refinement is very strong indeed: it prohibits some refinement steps that do not jeopardise security. To make it tractable, it may be necessary to adopt a multi-user refinement relation (see Section 3.6) in place of the canonical notion of refinement.

4.8 Conclusion

This chapter has presented an abstract framework for defining confidentiality properties in a UTP style. The framework cleanly captures the essence of confidentiality properties in an abstract manner, without specialising to any particular UTP theory. By specialising the framework to a particular UTP theory, it could be deployed to formulate confidentiality properties over the program state (in the theory of designs); the trace generated by a process (in the theory of reactive processes); or the execution period of operations (in timed theories).

A further novelty of the framework lies in how it enables confidentiality properties to be integrated with a functional specification of a system. To our knowledge, our approach for uniting functionality and confidentiality concerns with a single semantics is original. In the next chapter, this novelty is applied to incorporate both functionality and confidentiality attributes within a single specification language.

Not all conceivable confidentiality properties can be expressed as obligations. The notion of multi-obligations allows us to widen the space of confidentiality properties that can be expressed within the framework. As Section 4.6 demonstrates, the addition of multi-obligations gives expressive power that suffices to capture many of the information flow properties detailed in the literature.

The abstractness of the framework undermines its practicality as a platform for developing secure software. Some significant obstacles that hinder the practical application of the framework are as follows:

Compositionality Compositionality is essential for understanding a system in terms of its individual components. But the framework is defined over the totality of a system's behaviour. To make the framework practical, it is necessary to identify how it can be applied to individual components of a system.

Localisation Confidentiality requirements are often directly related to particular aspects of a system, principally those involving interactions with high-level users. To formulate these requirements within our framework, they must be encoded *indirectly* over the space of behaviours of a system. This indirection adds undesirable complexity to specifying confidentiality properties in the framework.

Verification The conformance condition is defined purely semantically. Without proof rules, it is necessary to translate a system into its underlying semantics in order to verify it against an obligation, which is cumbersome and inefficient.

The next chapter overcomes these obstacles with a tighter integration of the confidentiality framework with the syntax and semantics of *Circus*. It demonstrates how the framework can be made tractable, but at the expense of sacrificing some of the framework's generality.

5 Specifying Confidentiality in *Circus*

5.1 Introduction

In this chapter, we specialise the confidentiality framework presented in Chapter 4 to express confidentiality properties over *Circus* processes. This work paves the way towards an *integrated* language for specifying the functionality and confidentiality attributes of systems alongside each other. The philosophy of this integrated language is to exploit the facilities provided by *Circus* as far as practicable, in order to maintain compatibility with the existing laws and refinement calculus of *Circus*.

In Section 5.2 and Section 5.3, we lift the UTP semantics of *Circus* actions and operators to model Low's inferences about the execution of those constructs in a compositional fashion. Section 5.4 presents a simple extension of the *Circus* syntax that is needed to ensure processes in our lifted language have an unambiguous semantics.

The central novelty of this chapter is a specification construct for expressing confidentiality properties within the body of a *Circus* process. We define *confidentiality annotations* in Section 5.5, in terms of the lifted semantics of *Circus* and the obligations described in Chapter 4. These annotations can be used in tandem with the specification facilities of *Circus* to formulate a variety of confidentiality properties over the state and behaviour of *Circus* processes.

In Section 5.6, we show how confidentiality annotations can be applied to express a diverse variety of confidentiality properties in terms of a process's interactions with its environment. This work culminates with a technique for *superposing* confidentiality annotations with existing *Circus* processes, enabling the specification of noninterference-like properties in our integrated language.

In Section 5.7, we discuss how the integrated language may be extended to accommodate the declassification of secret information. Finally, in Section 5.8, we draw comparisons between our integrated language and recent work on accommodating security concerns within formal methods.

This chapter emphasises the theoretical aspects of the integrated language. We defer the study of a tractable method for developing systems from specifications expressed in the language until the next chapter.

5.2 Lifting the *Circus* Semantics

The first step towards defining our integrated language is to codify Low's inferences about the behaviour of *Circus* actions at the semantic level. We accomplish this step by lifting the UTP semantics of *Circus* actions, following the approach described in Subsection 4.2.1. This lifted semantics supplies the foundation we need for defining confidentiality annotations in Section 5.5.

5.2.1 Lifting *Circus* Actions

Since *Circus* actions are reactive designs, we model Low's interactions (and hence inferences) with a *Circus* process in terms of the reactive views defined in Subsection 3.5.1. The indistinguishability relation formed by applying the **IR** predicate transformer (Definition 4.16) to the reactive interface $\mathcal{R}(\mathcal{L})$ (Definition 3.43) corresponding to Low's window $\mathcal{L}$ is equivalent to:

$$\left(\begin{array}{ll} & ok = \widetilde{ok} \wedge ok' = \widetilde{ok'} \\ \wedge & wait = \widetilde{wait} \wedge wait' = \widetilde{wait'} \\ \wedge & (tr' - tr) \upharpoonright \mathcal{L} = (\widetilde{tr'} - \widetilde{tr}) \upharpoonright \mathcal{L} \\ \wedge & ref \cap \mathcal{L} = \widetilde{ref} \cap \mathcal{L} \wedge ref' \cap \mathcal{L} = \widetilde{ref'} \cap \mathcal{L} \end{array}\right)$$

We can simplify this indistinguishability relation in the context of *Circus* actions. Each *Circus* action A is **CSP3**- and **CSP4**-healthy; that is, A does not depend upon *ref*, nor does it restrict ref' on termination (Oliveira et al., 2009). Hence, our *Circus* indistinguishability relation need only constrain the relation between ref' and $\widetilde{ref'}$ on non-termination.

Definition 5.1 (*Circus* indistinguishability relation)**.**

$$\mathcal{I}(\mathcal{L}) \quad \triangleq \quad \left(\begin{array}{ll} & ok = \widetilde{ok} \land ok' = \widetilde{ok'} \\ \land & wait = \widetilde{wait} \land wait' = \widetilde{wait'} \\ \land & (tr' - tr) \upharpoonright \mathcal{L} = (\widetilde{tr'} - \widetilde{tr}) \upharpoonright \mathcal{L} \\ \land & wait' \Rightarrow ref' \cap \mathcal{L} = \widetilde{ref'} \cap \mathcal{L} \end{array} \right)$$

The $\mathcal{I}(\mathcal{L})$ relation codifies the expectation that Low's observation of the behaviour of a *Circus* process amounts to a projection of its trace through $\mathcal{L}$, together with knowledge of whether the process has started correctly, is waiting for interaction from the environment, or has terminated. Henceforth, we adopt $\mathcal{I}(\mathcal{L})$ as our model of Low's observational abilities of *Circus* actions and processes.

By lifting the semantics of *Circus* actions and conjoining $\mathcal{I}(\mathcal{L})$, we can model Low's inferences about an action's behaviour in terms of the fog behaviours. Definition 5.2 presents a specialised form of the lifting function **UI** (Definition 4.17) which incorporates $\mathcal{I}(\mathcal{L})$.

Definition 5.2 (**UC** predicate transformer)**.**

$$\mathbf{UC}\,(\mathcal{L}, A) \quad \triangleq \quad \mathbf{U}\,(A) \land \mathcal{I}(\mathcal{L})$$

We call $\mathbf{UC}\,(\mathcal{L}, A)$ the lift of the *Circus* action A with respect to $\mathcal{L}$. Reassuringly, there exists an order-embedding between *Circus* actions and **UC**-lifted actions, which is a special form of Theorem 4.4.

Lemma 5.3 (**UC** order-embedding)**.**

$$A_1 \sqsubseteq A_2 \quad \text{if and only if} \quad \mathbf{UC}\,(\mathcal{L}, A_1) \sqsubseteq \mathbf{UC}\,(\mathcal{L}, A_2)$$

Furthermore, Lemma 4.6 applies to lifted *Circus* actions too.

Law 5.4 (Recovering actions)**.** For all *Circus* actions,

$$\mathbf{D}\,(\mathbf{UC}\,(\mathcal{L}, A)) \quad = \quad \mathbf{D}\,(\mathbf{U}\,(A)) \quad = \quad A$$

5.2.2 Lifting Reactive Designs

While *Circus* actions are reactive designs, their lifted counterparts are not reactive, because **UC** does not commute with **R3**:

$$\begin{array}{rcl} \mathbf{UC}\,(\mathcal{L}, \mathbf{R3}(P)) & = & \mathbf{U}\,(I\!I_{rea} \lhd wait \rhd P) \wedge \mathcal{I}(\mathcal{L}) \\ & \neq & I\!I_{rea} \lhd wait \rhd (\mathbf{U}\,(P) \wedge \mathcal{I}(\mathcal{L})) \\ & = & \mathbf{R3} \circ \mathbf{UC}\,(\mathcal{L}, P) \end{array}$$

Nevertheless, the **UC**-lifts of *Circus* actions can be reshaped into a form with the character of reactive designs. Given an action A, its fog counterpart $\widetilde{A}$ (with alphabet $\widetilde{x}, \widetilde{x'}$) satisfies the following healthiness conditions:

$$\begin{array}{rcl} \widetilde{\mathbf{R1}}(A) & = & A \wedge \widetilde{tr} \leq \widetilde{tr'} \\ \widetilde{\mathbf{R2}}(A) & = & A[\langle\rangle, \widetilde{tr'} - \widetilde{tr} / \widetilde{tr}, \widetilde{tr'}] \\ \widetilde{\mathbf{R3}}(A) & = & (\widetilde{I\!I_{rea}} \lhd \widetilde{wait} \rhd A) \end{array}$$

Naturally, $\widetilde{\mathbf{R1}}$, $\widetilde{\mathbf{R2}}$ and $\widetilde{\mathbf{R3}}$ are idempotent and commute with each other. Moreover, $\widetilde{\mathbf{R1}}$ and $\widetilde{\mathbf{R2}}$ commute with **R1** and **R2**. However, **R3** and $\widetilde{\mathbf{R3}}$ do not commute with each other in general, because $\mathbf{R3}(A)$ behaves as $I\!I_{rea}$ when *wait* holds, but $\widetilde{\mathbf{R3}}(A)$ behaves as $\widetilde{I\!I_{rea}}$ when $\widetilde{wait}$ holds. To sidestep this mismatch, we insist that $wait = \widetilde{wait}$ and define a compound healthiness condition:

$$\begin{array}{rcl} \widehat{\mathbf{R3}}(A) & \triangleq & \mathbf{R3}(A) \wedge \widetilde{\mathbf{R3}}(A) \wedge wait = \widetilde{wait} \\ & = & \big((I\!I_{rea} \wedge \widetilde{I\!I_{rea}}) \lhd wait \rhd A\big) \wedge wait = \widetilde{wait} \end{array}$$

$\widehat{\mathbf{R3}}$ is idempotent and commutes with **R1**, $\widetilde{\mathbf{R1}}$, **R2** and $\widetilde{\mathbf{R2}}$. We call the composition of these healthiness conditions $\widehat{\mathbf{R}}$.

Definition 5.5 ($\widehat{\mathbf{R}}$ healthiness condition)**.**

$$\widehat{\mathbf{R}}(A) \quad \triangleq \quad \mathbf{R1} \circ \widetilde{\mathbf{R1}} \circ \mathbf{R2} \circ \widetilde{\mathbf{R2}} \circ \widehat{\mathbf{R3}}(A)$$

Since the exterior $\mathcal{I}(\mathcal{L})$ term of $\mathbf{UC}\,(\mathcal{L}, A)$ implies $wait = \widetilde{wait}$, we can express $\mathbf{UC}\,(\mathcal{L}, A)$ — where A is $\mathbf{R}$-healthy — in terms of $\widehat{\mathbf{R}}$.

Lemma 5.6 (Lifted reactive process)**.** Provided A is $\mathbf{R}$-healthy:

$$\mathbf{UC}\,(\mathcal{L}, A) \quad = \quad \widehat{\mathbf{R}}\,(\mathbf{U}\,(A)) \wedge \mathcal{I}(\mathcal{L})$$

Theorem 5.7 builds on Lemma 5.6 by reformulating the interior $\mathbf{U}\,(A)$ term as a design, which is justified because $\mathcal{I}(\mathcal{L})$ implies the condition $ok = \widetilde{ok} \wedge ok' = \widetilde{ok'}$.

Theorem 5.7 (Lifted reactive design)**.** Provided A is a reactive design:

$$\mathbf{UC}\,(\mathcal{L}, A) \quad = \quad \widehat{\mathbf{R}}\left(\neg\, \mathbf{U}\,(A)^{ff}_{ff} \vdash \mathbf{U}\,(A)^{tt}_{ff}\right) \wedge \mathcal{I}(\mathcal{L})$$

where B^{bc}_{de} denotes $B[b, c, d, e / ok', \widetilde{ok'}, wait, \widetilde{wait}]$.

The semantics of $\mathbf{UC}\,(\mathcal{L}, A)$ from Theorem 5.7 is clarified by Lemma 5.8, which expresses A in the form $\mathbf{R}\,(Pre \vdash Post)$.

Lemma 5.8 (Unfolded lifted reactive design)**.** Provided Pre is a condition:

$$\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post)) \quad = \quad \widehat{\mathbf{R}}\left(Pre \vee \widetilde{Pre} \vdash \mathbf{U}\,(Pre \Rightarrow Post)\right) \wedge \mathcal{I}(\mathcal{L})$$

Lemma 5.8 can be instantiated with the semantics of the *Circus* specification statement, to derive a lifted specification statement presented in Lemma 5.9. The lifted forms of assignments, assumptions and coercions are all derivable from Lemma 5.9.

Lemma 5.9 (Lifted specification statement)**.**

$$\begin{aligned} &\mathbf{UC}\,(\mathcal{L}, w : [Pre, Post]) = \\ &\quad \widehat{\mathbf{R}}\left(Pre \vee \widetilde{Pre} \vdash \mathbf{U}\,(Pre \Rightarrow (Post \wedge \neg\, wait' \wedge tr' = tr \wedge u' = u))\right) \wedge \mathcal{I}(\mathcal{L}) \end{aligned}$$

where u denotes all variables outside of the frame w.

Reasoning about Low's inferences about a *Circus* action's behaviour can be simplified by studying the lifted postcondition of that action. Example 5.10 demonstrates how this principle can be applied.

Example 5.10. Consider the action $OO = on \to h := 0 \;\Box\; \mathit{off} \to h := 1$. Its semantics is:

$$\mathbf{R}\left(\mathbf{true} \vdash \left(\begin{array}{c} v' = v \wedge tr' = tr \wedge (on, Sync), (\mathit{off}, Sync) \notin ref' \\ \lhd\, wait' \,\rhd \\ u' = u \wedge \left(\begin{array}{c} h' = 0 \wedge tr' = tr \frown \langle (on, Sync) \rangle \\ \vee\, h' = 1 \wedge tr' = tr \frown \langle (\mathit{off}, Sync) \rangle \end{array}\right) \end{array}\right)\right)$$

where u denotes the list of state variables excluding h; and $s_1, s_2 \notin S$ abbreviates $s_1 \notin S \wedge s_2 \notin S$.

Suppose $\mathcal{L} = \{(on, Sync)\}$. Unfolding the postcondition of $\mathbf{UC}\,(\mathcal{L}, OO)$ shows how its fog behaviours are reduced in the presence of $\mathcal{I}(\mathcal{L})$:

$$\begin{array}{ll} & \mathbf{U}\left(\begin{array}{c} v' = v \wedge tr' = tr \wedge (on, Sync), (\mathit{off}, Sync) \notin ref' \\ \lhd\, wait' \,\rhd \\ u' = u \wedge \left(\begin{array}{c} h' = 0 \wedge tr' = tr \frown \langle (on, Sync) \rangle \\ \vee\, h' = 1 \wedge tr' = tr \frown \langle (\mathit{off}, Sync) \rangle \end{array}\right) \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \\ = & \left[\mathcal{I}(\mathcal{L}) \text{ implies } wait' = \widetilde{wait'}\right] \\ & \left(\begin{array}{c} \mathbf{U}\,(v' = v \wedge tr' = tr \wedge (on, Sync), (\mathit{off}, Sync) \notin ref') \\ \lhd\, wait' \,\rhd \\ \mathbf{U}\left(u' = u \wedge \left(\begin{array}{c} h' = 0 \wedge tr' = tr \frown \langle (on, Sync) \rangle \\ \vee\, h' = 1 \wedge tr' = tr \frown \langle (\mathit{off}, Sync) \rangle \end{array}\right)\right) \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \\ = & \left[\mathcal{I}(\mathcal{L}) \text{ implies } (tr' - tr) \upharpoonright \mathcal{L} = (\widetilde{tr'} - \widetilde{tr}) \upharpoonright \mathcal{L}\right] \\ & \left(\begin{array}{c} \mathbf{U}\,(v' = v \wedge tr' = tr \wedge (on, Sync), (\mathit{off}, Sync) \notin ref')\ (1) \\ \lhd\, wait' \,\rhd \\ \mathbf{U}\,(u' = u \wedge h' = 0 \wedge tr' = tr \frown \langle (on, Sync) \rangle)\ (2) \\ \vee\, \mathbf{U}\,(u' = u \wedge h' = 1 \wedge tr' = tr \frown \langle (\mathit{off}, Sync) \rangle)\ (3) \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \end{array}$$

At the semantic level, this result models Low as being able to distinguish between the labelled alternative behaviours and therefore being able to infer information about the state: In case (1), Low observes deadlock, so it infers $h' = h$. In (2), Low observes termination and the *on* event, so it infers $h' = 0$. In (3), Low observes termination without observing the *off* event, so it infers the occurrence of *off* and $h' = 1$. ♢

5.2.3 Addressing Divergence

Although A and $\widetilde{A}$ may diverge individually, Lemma 5.8 makes clear that **UC** $(\mathcal{L}, A)$ may only diverge in its own right — i.e. $\neg\, ok' \wedge \neg\, \widetilde{ok'}$ — if neither of the preconditions of A or $\widetilde{A}$ are satisfied, or if the lifted process has already diverged. If **UC** $(\mathcal{L}, A)$ does diverge, the exterior $\mathcal{I}(\mathcal{L})$ term forces the diverging behaviour of $\widetilde{A}$ to be Low-indistinguishable from the behaviour of A. This effect is not problematic, because all diverging behaviours of A are present in **UC** $(\mathcal{L}, A)$ (via the order embedding).

Of course, the possibility of divergence evaporates if A has **true** as its precondition. In practice, divergence is undesirable (Hoare, 1985a) and tools (such as FDR) for establishing that CSP constructs are divergence-free are readily available.

5.3 Composing Lifted Actions

To analyse Low's inferences about a *Circus* process, it is helpful to *divide* the process into multiple parts, so that each part can be *conquered* by analysing it separately. However, we need to compose those separate analyses, in order to study Low's inferences about the whole process.

Circus actions can be composed using the *Circus* operators. However, the definitions of the *Circus* operators given by Oliveira et al. (2009) are unsuitable for composing lifted actions, because they do not include the fog variables. In this section, we rectify this problem by lifting the definitions of these operators to make them suitable for composing lifted actions. Moreover, these lifted operators are also needed for composing confidentiality properties with lifted actions, as we see later.

For each *Circus* operator $\oplus$, we denote its lifted counterpart by $\widehat{\oplus}$. We now describe three principal conditions to guide the definition of $\widehat{\oplus}$. These conditions ensure that $\widehat{\oplus}$ can be used to compose lifted actions in the same way that $\oplus$ composes *Circus* actions.

Condition 5.11. Lifted operators respect the order-embedding between *Circus* actions and lifted actions:

$$\bigoplus i \bullet A_i \quad = \quad \mathbf{D}\left(\widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)\right)$$

This condition ensures that applying $\widehat{\oplus}$ to the lifts of actions A_1 and A_2 yields a lifted action with exactly the same observable behaviours as the *Circus* action $A_1 \oplus A_2$.

Condition 5.12. Decomposing a lifted action into parts and combining those parts with lifted operators respects the refinement ordering:

$$\mathbf{UC}\left(\mathcal{L}, \bigoplus i \bullet A_i\right) \quad \sqsubseteq \quad \widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)$$

Splitting a lifted composite action $\mathbf{UC}\,(\mathcal{L}, A_1 \oplus A_2)$ into its constituents and composing those lifted constituents with $\widehat{\oplus}$ preserves correctness, but not necessarily equivalence. The construct $\widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)$ models Low as being able to observe the execution of each sub-action A_i individually. This means $\widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)$ represents an over-approximation of the information Low can deduce about the behaviour of $\bigoplus i \bullet A_i$.

This condition is a compromise. It would be preferable to insist on the stronger condition that $\mathbf{UC}\,(\mathcal{L}, \bigoplus i \bullet A_i) = \widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)$. However, the definitions of lifted operators respecting that condition would be excessively complicated and intractable to analyse.

Condition 5.13. Lifted operators are monotonic with respect to $\sqsubseteq$:

$$\begin{array}{l} \mathsf{UC}\,(\mathcal{L}, A_i) \sqsubseteq \mathsf{UC}\,(\mathcal{L}, A'_i) \quad \text{for each } i \\ \quad \text{implies} \quad \left(\widehat{\bigoplus} i \bullet \mathsf{UC}\,(\mathcal{L}, A_i)\right) \sqsubseteq \left(\widehat{\bigoplus} i \bullet \mathsf{UC}\,(\mathcal{L}, A'_i)\right) \end{array}$$

This condition is essential to support piecewise refinement of lifted specifications. It enables us to refine a lifted action B in isolation, with the assurance that any context in which B appears is also refined.

Theorem 5.14. Each lifted operator defined in this section satisfies Conditions 5.11, 5.12 and 5.13.

5.3.1 Sequential Composition

Closed forms of some lifted operators can be derived by unfolding the lifts of the respective *Circus* operators. For instance, Lemma 5.15 presents the outcome of unfolding the sequence operator.

Lemma 5.15 (Unfolded sequence)**.**

$$\mathsf{UC}\,(\mathcal{L}, A_1 ; A_2) \quad \sqsubseteq \quad \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & \mathsf{UC}\,(\mathcal{L}, A_1)\,[x_0, \widetilde{x_0} / x', \widetilde{x}'] \\ \wedge & \mathsf{UC}\,(\mathcal{L}, A_2)\,[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array} \right)$$

We obtain a definition of lifted sequential composition by generalising Lemma 5.15: we replace $\mathsf{U}\,(A_1)$ and $\mathsf{U}\,(A_2)$ with arbitrary points in the lifted space B_1 and B_2.

Definition 5.16 (Lifted sequential composition)**.**

$$B_1 \mathbin{\widehat{;}} B_2 \quad \triangleq \quad \exists\, x_0, \widetilde{x_0} \bullet B_1[x_0, \widetilde{x_0} / x', \widetilde{x}'] \wedge B_2[x_0, \widetilde{x_0} / x, \widetilde{x}]$$

Since the $\widehat{;}$ operator is just relational composition over the $x', \widetilde{x}'$ and $x, \widetilde{x}$ variables, it respects the three conditions. Moreover, Law 5.17 shows that the laws of sequential composition (such as associativity) also apply to the lifted sequential composition operator.

Law 5.17 (U and $\widehat{;}$)**.** $\mathsf{U}\,(A_1 ; A_2) = \mathsf{U}\,(A_1) \mathbin{\widehat{;}} \mathsf{U}\,(A_2)$

Whenever a **UC**-lifted construct comprises an action in sequence with a coercion, assignment or *Skip*, it can be restructured as Lemma 5.18 shows.

Lemma 5.18. Provided that either A_1 or A_2 terminate instantaneously, leaving the trace unchanged:

$$\mathbf{UC}(\mathcal{L}, A_1 ; A_2) \quad = \quad \mathbf{UC}(\mathcal{L}, A_1) \mathbin{\widehat{;}} \mathbf{UC}(\mathcal{L}, A_2)$$

5.3.2 Internal Choice

A closed form for lifted internal choice is easily derived by unfolding.

Lemma 5.19 (Unfolded internal choice)**.**

$$\mathbf{UC}(\mathcal{L}, A_1 \sqcap A_2) \quad \sqsubseteq \quad \mathbf{UC}(\mathcal{L}, A_1) \vee \mathbf{UC}(\mathcal{L}, A_2)$$

Here, the refinement step cuts down the fog behaviours associated with A_1 and A_2 individually, but all behaviours of A_1 and A_2 are retained. Again, we generalise Lemma 5.19 to define a lifted internal choice operator which, in this case, is identical to the *Circus* internal choice.

Definition 5.20 (Lifted internal choice)**.**

$$B_1 \mathbin{\widehat{\sqcap}} B_2 \quad \triangleq \quad B_1 \vee B_2$$

5.3.3 Prefixing

A prefixed action $c \rightarrow A$ is identical to $(c \rightarrow Skip) ; A$. This leads to a straightforward definition of the lifted forms of prefixing and input prefixing, based on the lifted sequence operator.

Definition 5.21 (Lifted prefixing)**.**

$$\begin{aligned} c.e \mathbin{\widehat{\rightarrow}} B \quad &\triangleq \quad \mathbf{UC}(\mathcal{L}, c.e \rightarrow Skip) \mathbin{\widehat{;}} B \\ c?e \mathbin{\widehat{\rightarrow}} B \quad &\triangleq \quad \mathbf{UC}(\mathcal{L}, c?e \rightarrow Skip) \mathbin{\widehat{;}} B \end{aligned}$$

The lifted reactive design form of lifted prefixing — presented in Lemma 5.22 — is derived by instantiating Theorem 5.7.

Lemma 5.22 (Unfolded prefixing)**.**

$$\begin{array}{l} \mathsf{UC}\,(\mathcal{L}, c.e \rightarrow \mathit{Skip}) \quad = \\ \widehat{\mathbf{R}} \left(\mathbf{true} \vdash \mathsf{U}\,(v' = v) \wedge \left(\begin{array}{c} \mathsf{U}\,(tr' = tr \wedge c.e \notin \mathit{ref}') \\ \lhd\ \mathit{wait}'\ \rhd \\ \mathsf{U}\,(tr' = tr ^\frown \langle (c,e) \rangle) \end{array} \right) \right) \wedge \mathcal{I}(\mathcal{L}) \end{array}$$

Lemma 5.22 indicates that, if Low could observe $c.e \rightarrow \mathit{Skip}$ in isolation, then Low could infer whether the event $c.e$ has taken place — from wait and wait' — even if that event is absent from Low's window.

If Low can observe the event $c.e$, then the lifted form of $c.e \rightarrow A$ can be decomposed by separating the event instance from the action, as Lemma 5.23 shows.

Lemma 5.23 (Decomposing prefixing)**.** Provided $(c,e) \in \mathcal{L}$:

$$\mathsf{UC}\,(\mathcal{L}, c.e \rightarrow A) \quad = \quad \mathsf{UC}\,(\mathcal{L}, c.e \rightarrow \mathit{Skip})\ \widehat{;}\ \mathsf{UC}\,(\mathcal{L}, A)$$

5.3.4 Guarded Actions

Lemma 5.24 reformulates lifted guarded actions as lifted reactive designs.

Lemma 5.24 (Unfolded guard)**.**

$$\begin{array}{l} \mathsf{UC}\,(\mathcal{L}, g \,\&\, A) \quad \sqsubseteq \\ \widehat{\mathbf{R}} \left(\left(\mathsf{U}\,(g) \Rightarrow \neg\, \mathsf{U}\,(A)^{ff}_{ff} \right) \vdash \left(\begin{array}{l} \mathsf{U}\,(g) \wedge \mathsf{U}\,(A)^{tt}_{ff} \\ \vee\, \mathsf{U}\,(\neg\, g \wedge tr' = tr \wedge \mathit{wait}') \end{array} \right) \right) \wedge \mathcal{I}(\mathcal{L}) \end{array}$$

Here, the postcondition of $\mathsf{UC}\,(\mathcal{L}, g \,\&\, A)$ is strengthened by refinement, to impose the constraint $\mathsf{U}\,(g) \vee \mathsf{U}\,(\neg\, g)$ upon the fog space. This refinement is necessary to obtain a closed form purely in terms of $\mathsf{U}\,(A)$.

Definition 5.25 (Lifted guarded action)**.**

$$g \mathbin{\widehat{\&}} B \quad \triangleq \quad \widehat{\mathbf{R}}\left(\left(\mathbf{U}(g) \Rightarrow \neg\, B^{ff}_{ff}\right) \vdash \left(\begin{array}{l} \mathbf{U}(g) \wedge B^{tt}_{ff} \\ \vee\, \mathbf{U}(\neg\, g \wedge tr' = tr \wedge wait') \end{array}\right)\right)$$

Under this definition, we are effectively modelling Low as being able to determine whether g is **true** or **false**. Depending on the process in which the guarded action is embedded, this may over-approximate Low's inferences about the process state.

5.3.5 External Choice

Lemma 5.26 shows how lifted external choice can be refined to derive an expression in terms of the lifted actions $\mathbf{U}(A_1)$ and $\mathbf{U}(A_2)$.

Lemma 5.26 (Unfolded external choice)**.**

$$\begin{array}{l} \mathbf{UC}(\mathcal{L}, A_1 \,\Box\, A_2) \quad \sqsubseteq \\ \quad \widehat{\mathbf{R}}\left(\begin{array}{l} \neg\, \mathbf{U}\left(A_{1f}^{\,f}\right) \wedge \neg\, \mathbf{U}\left(A_{2f}^{\,f}\right) \\ \vdash \left(\begin{array}{ll} & \mathbf{U}(tr' = tr \wedge wait') \wedge \mathbf{U}(A_1)^{tt}_{ff} \wedge \mathbf{U}(A_2)^{tt}_{ff} \\ \vee & \mathbf{U}(\neg\,(tr' = tr \wedge wait')) \wedge \left(\mathbf{U}(A_1)^{tt}_{ff} \vee \mathbf{U}(A_2)^{tt}_{ff}\right) \end{array}\right) \end{array}\right) \\ \quad \wedge\, \mathcal{I}(\mathcal{L}) \end{array}$$

In summary, the consequences of the refinement in Lemma 5.26 are:

- Weakening the precondition means that both A_1 and $\widetilde{A_1}$ (or A_2 and $\widetilde{A_2}$) must diverge for the lifted external choice to diverge. This contrasts with $\mathbf{UC}(\mathcal{L}, A_1 \,\Box\, A_2)$, where divergence in either the observable space or the fog space suffices to make the entire lifted action divergent. Since divergence is to be avoided (Subsection 5.2.3), this point is unlikely to have importance in practice.
- Strengthening the postcondition models Low as being able to determine whether the external choice has resolved. Since Low can

perceive deadlock (by definition of $\mathcal{I}(\mathcal{L})$), this means that Low can detect whether any events have been performed, even if those events are outside its window. This over-approximation of Low's inferences, while perhaps counter-intuitive, has little impact on the application of lifted constructs.

Lemma 5.26 motivates the following definition for the lifted external choice operator.

Definition 5.27 (Lifted external choice)**.**

$$B_1 \mathbin{\widehat{\Box}} B_2 \quad \triangleq \quad \widehat{\mathbf{R}} \left(\begin{array}{l} \neg B_1{}^{ff}_{ff} \wedge \neg B_2{}^{ff}_{ff} \\ \vdash \left(\begin{array}{rl} & \mathbf{U}\,(tr' = tr \wedge wait') \wedge B_1{}^{tt}_{ff} \wedge B_2{}^{tt}_{ff} \\ \vee & \mathbf{U}\,(\neg\,(tr' = tr \wedge wait')) \wedge \left(B_1{}^{tt}_{ff} \vee B_2{}^{tt}_{ff}\right) \end{array} \right) \end{array} \right)$$

5.3.6 Parallel Composition

In CSP and *Circus*, a parallel composition $A_1 \llbracket cs \rrbracket A_2$ diverges whenever A_1 or A_2 diverge, provided those actions agree on their previous synchronsiations. Lemma 5.28 lifts Oliveira's definition of the precondition of a parallel composition.

Lemma 5.28 (Unfolded parallel precondition)**.**

$$PREPAR(\mathbf{U}\,(A_1)\,, \mathbf{U}\,(A_2)) \quad \sqsubseteq \quad \mathbf{U}\left(\neg\,(A_1 \llbracket cs \rrbracket A_2)^f_f\right)$$

where $PREPAR(\mathbf{U}\,(A_1)\,, \mathbf{U}\,(A_2))$ abbreviates:

$$\begin{array}{rl} & \neg\ \exists\, 1.tr, 2.tr, \widetilde{1.tr}, \widetilde{2.tr} \bullet \left(\begin{array}{rl} & \mathbf{U}\,(A_1)^{ff}_{ff} \mathbin{\widehat{;}} \mathbf{U}\,(1.tr' = tr) \\ \wedge & \mathbf{U}\,(A_2)_{ff} \mathbin{\widehat{;}} \mathbf{U}\,(2.tr' = tr) \\ \wedge & \mathbf{U}\,(1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs) \end{array} \right) \\ \wedge & \neg\ \exists\, 1.tr, 2.tr, \widetilde{1.tr}, \widetilde{2.tr} \bullet \left(\begin{array}{rl} & \mathbf{U}\,(A_1)_{ff} \mathbin{\widehat{;}} \mathbf{U}\,(1.tr' = tr) \\ \wedge & \mathbf{U}\,(A_2)^{ff}_{ff} \mathbin{\widehat{;}} \mathbf{U}\,(2.tr' = tr) \\ \wedge & \mathbf{U}\,(1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs) \end{array} \right) \end{array}$$

As with lifted external choice, weakening the precondition is necessary to derive an expression in terms of $\mathbf{U}(A_1)$ and $\mathbf{U}(A_2)$. Here, this weakening ensures the lifted parallel composition diverges only if both the observable trace and the fog trace of either lifted action leads to divergence, and the other lifted action is prepared to synchronise on the projections of those traces. Again, this point is of no importance if the lifted actions are divergence-free.

Lemma 5.28 is a stepping stone for the following derivation of a lifted reactive design modelling the parallel composition of two lifted actions:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{UC}(\mathcal{L}, A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)} \\
\sqsubseteq & \widehat{\mathbf{R}}\left(\begin{array}{l} PREPAR(\mathbf{U}(A_1), \mathbf{U}(A_2)) \\ \vdash \mathbf{U}\left(\left(\begin{array}{l} A_{1}{}^{t}_{f} ; U1(out\alpha A_1) \\ \wedge A_{2}{}^{t}_{f} ; U2(out\alpha A_2) \end{array}\right) ; M_{\parallel}(cs)\right) \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \\
& \hfill \text{[Theorem 5.7; Lemma 5.28 (weaken precondition)]} \\
= & \widehat{\mathbf{R}}\left(\begin{array}{l} PREPAR(\mathbf{U}(A_1), \mathbf{U}(A_2)) \\ \vdash \left(\begin{array}{l} \mathbf{U}(A_1)^{tt}_{ff} \mathbin{\widehat{;}} \mathbf{U}(U1(out\alpha A_1)) \\ \wedge \mathbf{U}(A_2)^{tt}_{ff} \mathbin{\widehat{;}} \mathbf{U}(U2(out\alpha A_2)) \end{array}\right) \mathbin{\widehat{;}} \mathbf{U}(M_{\parallel}(cs)) \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \\
& \hfill \text{[Law 4.7; Law 5.17]}
\end{array}$$

We interpret the relabelled variables $i.x, \widetilde{i.x}$ as members of $x, \widetilde{x}$ respectively, so $\mathbf{U}$ extends over them. By this interpretation, lifting the relabelling functions $U1$ and $U2$ is straightforward:

$$\mathbf{U}(Ui(out\alpha A)) \quad = \quad x.i = x' \wedge \widetilde{x.i} = \widetilde{x}'$$

Expanding the lifted form of the merge function $M_{\parallel}(cs)$ is not illuminating, so we refrain from doing so.

Finally, Definition 5.29 presents a parallel operator for lifted actions.

Definition 5.29 (Lifted parallel composition)**.**

$$\begin{array}{l} B_1 \ \widehat{[\![ns_1 \mid cs \mid ns_2]\!]} \ B_2 \quad \triangleq \\ \widehat{\mathbf{R}} \left(PREPAR(B_1, B_2) \vdash \left(\begin{array}{ll} & B1^{tt}_{ff} \mathbin{\widehat{;}} U1(out\alpha B_1) \\ \wedge & B2^{tt}_{ff} \mathbin{\widehat{;}} U2(out\alpha B_2) \end{array} \right) \mathbin{\widehat{;}} \mathbf{U}\left(M_{\|}(cs)\right) \right) \end{array}$$

5.3.7 Hiding

Hiding is the final operator we consider. Again, we derive a lifted form by unfolding $\mathbf{UC}\,(\mathcal{L}, A \setminus cs)$.

Lemma 5.30 (Unfolded hiding)**.**

$$\mathbf{UC}\,(\mathcal{L}, A \setminus cs) \quad = \quad (HID(\mathbf{U}\,(A)) \wedge \mathcal{I}(\mathcal{L})) \mathbin{\widehat{;}} \mathbf{UC}\,(\mathcal{L}, Skip)$$

where $HID(B)$ denotes:

$$\widehat{\mathbf{R}} \left(\exists\, s, \widetilde{s} \bullet \left(\begin{array}{ll} & B[s, \widetilde{s}, (cs \cup ref'), (cs \cup \widetilde{ref'}) / tr', \widetilde{tr'}, ref', \widetilde{ref'}] \\ \wedge & \mathbf{U}\,((tr' - tr) = (s - tr) \upharpoonright (EVENT - cs)) \end{array} \right) \right)$$

As usual, we adapt Lemma 5.30 to yield Definition 5.31.

Definition 5.31 (Lifted hiding)**.**

$$B \mathbin{\widehat{\setminus}} cs \quad \triangleq \quad (HID(B) \wedge \mathcal{I}(\mathcal{L})) \mathbin{\widehat{;}} \mathbf{UC}\,(\mathcal{L}, Skip)$$

Lifted hiding distributes through lifted operators in the same manner that *Circus* hiding distributes through *Circus* operators.

5.4 Lifting *Circus* Processes

This section addresses two issues. First, we identify a method for specifying the windows of users in a *Circus* specification. Second, we extend the *Circus* syntax to enable the specification of lifted processes.

5.4.1 Encoding Windows

It is natural to model Low's window as a subset of the channels of the process. Within a *Circus* specification, we can use the *Circus* keyword **channelset** to assign a label to that window.

Following Oliveira et al. (2009), a channel set specification can be readily translated to the corresponding set of all events visible to Low. Let $\delta(c)$ denote the type of channel c; if c does not communicate typed values, then we take $\delta(c) = \{Sync\}$. We then have:

$$\mathcal{L} \quad = \quad \{(c, e) \mid c \in L \wedge e \in \delta(c)\}$$

Example 5.32. To specify that Low can observe events on the *on* and *off* channels of a process, we write:

channel on, off
channelset $Low \triangleq \{\!|\ on, off\ |\!\}$

This channel set corresponds to the window $\{(on, Sync), (off, Sync)\}$. $\Diamond$

5.4.2 Lifting and Closure

In the UTP, it is desirable for program operators defined in some theory to be closed with respect to the theory: for instance, if P_1 and P_2 are objects of the theory, then so should be $P_1 \oplus P_2$. However, by applying the lifted *Circus* operators defined in Section 5.3 to **UC**-lifted actions, we can formulate lifted constructs that are not closed with respect to **UC**.

Example 5.33. The construct $\mathbf{UC}(\mathcal{L}, h := 0) \mathrel{\widehat{\sqcap}} \mathbf{UC}(\mathcal{L}, h := 1)$ cannot be derived by applying **UC** to any single *Circus* action. As Figure 5.1 shows, it is stronger than $\mathbf{UC}(\mathcal{L}, h := 0 \sqcap h := 1)$, but weaker than each of $\mathbf{UC}(\mathcal{L}, h := 0)$ and $\mathbf{UC}(\mathcal{L}, h := 1)$. $\Diamond$

Constructs such as $\mathbf{UC}(\mathcal{L}, h := 0) \mathrel{\widehat{\sqcap}} \mathbf{UC}(\mathcal{L}, h := 1)$ are not part of the **UC**-image of the space of *Circus* actions. Instead, they inhabit a larger space of predicates with the joint alphabet of observable and fog variables.

$$\begin{tikzcd}
A_1 & & A_2 & & \mathbf{UC}\,(\mathcal{L}, A_1) & & \mathbf{UC}\,(\mathcal{L}, A_2) \\
 & & & & & \mathbf{UC}\,(\mathcal{L}, A_1) \mathbin{\widehat{\sqcap}} \mathbf{UC}\,(\mathcal{L}, A_2) \arrow[ul] \arrow[ur] & \\
 & A_1 \sqcap A_2 \arrow[uul] \arrow[uur] & & & & \mathbf{UC}\,(\mathcal{L}, A_1 \sqcap A_2) \arrow[u] &
\end{tikzcd}$$

Figure 5.1: A comparison of the *Circus* refinement lattice (left side) with the refinement lattice of the lifted space (right side).

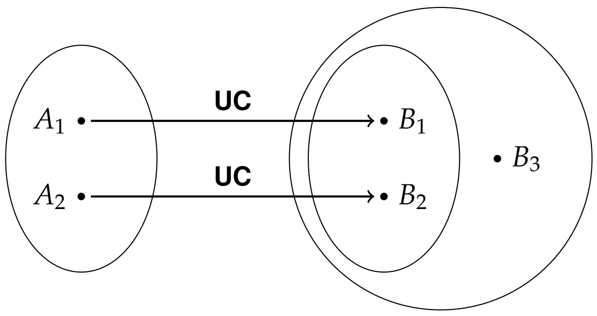

Figure 5.2: An illustration of the relationship between the *Circus* space (left side) and the lifted space (right side).

Figure 5.2 visualises the relationship between the *Circus* space, the **UC** space and this lifted space.

Example 5.34. In Figure 5.2, B_1 and B_2 are the **UC**-projections of A_1 and A_2. However, B_3 (which could denote $B_1 \mathbin{\widehat{\oplus}} B_2$) lies outside the **UC**-image of the space of *Circus* actions. $\Diamond$

Results which apply to the lifted **UC** space may fail to hold over the larger lifted space, in general. However, we shall restrict our attention to constructs that can be formed by combining **UC**-lifted actions with the lifted operators. This restriction guarantees the three principal conditions of the lifted operators are maintained for the constructs that we work with. It also ensures these constructs are expressible in the form $\widehat{\mathbf{R}}\,(Pre \vdash Post)$

via the definitions of the lifted operators. A new UTP theory would be needed to define this restricted space of acceptable lifted constructs formally, but this topic is left for future work.

For our purposes, the restriction on the form of lifted constructs is sufficient to avoid practical difficulties stemming from the lack of **UC**-closure. In particular, the techniques presented in Chapter 6 can be applied to lifted constructs both inside and outside the **UC** space.

An important consequence of moving outside the **UC** space is that a compound *Circus* construct — such as $A_1 \sqcap A_2$ — can be translated to the lifted space in multiple ways. For instance:

- $\mathbf{UC}\,(\mathcal{L}, A_1 \sqcap A_2)$ gives an exact representation of Low's inferences about $A_1 \sqcap A_2$; while
- $\mathbf{UC}\,(\mathcal{L}, A_1) \mathbin{\widehat{\sqcap}} \mathbf{UC}\,(\mathcal{L}, A_2)$ models Low's inferences about A_1 and A_2 separately. Segmenting lifted actions in this way makes it easier to analyse Low's inferences, albeit at the cost of over-approximation.

The difference between these forms has important consequences for verification and refinement, which are described in detail in Chapter 6. Hence, when we interpret a *Circus* specification in the lifted space, we need to pay special attention to distinguishing between these forms.

5.4.3 Blocks

We now present the *block*, which is a syntactic construct for specifying how *Circus* actions should be translated to the lifted space. Blocks are little more than concrete syntax for **UC**; we apply them to *Circus* actions to encode the granularity of lifted actions within the body of a process. By delineating the boundaries between lifted actions explicitly, blocks describe the translation of *Circus* processes into the lifted space.

Definition 5.35 (Block).

$$\langle L : A \rangle \quad \triangleq \quad (\mathbf{UC}\,(\mathcal{L}, A) \lhd \ell = L \rhd A)$$

where $\mathcal{L}$ denotes the window declared by the channel set L.

Each block is labelled with the name of a channel set, denoting a user's window. The metavariable ℓ can be instantiated with a window label, to focus attention on a specific user. This means that blocks can be used to encode the inferences of multiple users within a single process specification.

We say that blocks that are labelled with the same channel set are members of the same *family*. In the presentation that follows, we typically consider only a single Low user and omit an explicit window declaration from blocks. As a shorthand, we define:

$$\langle A \rangle \quad \triangleq \quad \langle \ell : A \rangle$$

Taking Low's window $\mathcal{L}$ to be implicit, Definition 5.35 gives us:

$$\langle A \rangle \quad = \quad \mathbf{UC}\,(\mathcal{L}, A)$$

The block construct sets the language apart from standard *Circus*. To maintain a level of compatibility with *Circus* specifications, we adhere to a list of syntactic conventions:

1. Each action of a process must be nested within a block.
2. Blocks of the same family cannot be nested within each other.
3. Outside of blocks, we use the syntax of *Circus* operators to denote the lifted forms of those operators.

Example 5.36. Recalling Example 5.33, we have:

$$\begin{aligned} \langle A_1 \sqcap A_2 \rangle \quad &= \quad \mathbf{UC}\,(\mathcal{L}, A_1 \sqcap A_2) \\ \langle A_1 \rangle \sqcap \langle A_2 \rangle \quad &= \quad \mathbf{UC}\,(\mathcal{L}, A_1)\ \widehat{\sqcap}\ \mathbf{UC}\,(\mathcal{L}, A_2) \end{aligned}$$

Henceforth, we tend to use the block notation in preference to **UC**, to aid readability. $\Diamond$

When dealing with processes that interact with multiple users, it may be convenient to model the inferences of different users at different levels of granularity. The atomicity of lifted constructs — from the perspective of different users — may be specified compactly by nesting blocks of different families.

Example 5.37. Let *Alice* label Alice's window, and *Bob* label Bob's window, within the following specification:

$$\langle\, \mathit{Alice} \,:\, \langle\, \mathit{Bob} \,:\, A_1 \,\rangle \sqcap \langle\, \mathit{Bob} \,:\, A_2 \,\rangle\,\rangle$$

If we are reasoning about Alice's inferences (i.e. $\ell = \mathit{Alice}$), we read this specification as $\langle\, \mathit{Alice} \,:\, A_1 \sqcap A_2 \,\rangle$. Whereas if we are reasoning about Bob, we read it as $\langle\, \mathit{Bob} \,:\, A_1 \,\rangle \sqcap \langle\, \mathit{Bob} \,:\, A_2 \,\rangle$. ◊

5.4.4 On Equivalence Laws

As a consequence of the order embedding between *Circus* actions and lifted actions (Lemma 5.3), all equivalence laws over *Circus* actions apply also to actions nested within blocks. However, Condition 5.12 implies that standard equivalence laws between *Circus* actions do not necessarily apply across blocks. For instance, while sequence distributes through internal choice within a *Circus* action:

$$\langle\, (A_1 \sqcap A_2) \,;A_3 \,\rangle \quad = \quad \langle\, (A_1 \,;A_3) \sqcap (A_2 \,;A_3) \,\rangle$$

it does not distribute across blocks, in general:

$$\langle\, A_1 \sqcap A_2 \,\rangle \;;\; \langle\, A_3 \,\rangle \quad \neq \quad \langle\, A_1 \,;A_3 \,\rangle \sqcap \langle\, A_2 \,;A_3 \,\rangle$$

Nevertheless, many algebraic properties of the *Circus* operators also apply to the lifted operators; for instance:

$$(\langle\, A_1 \,\rangle \sqcap \langle\, A_2 \,\rangle) \;;\; \langle\, A_3 \,\rangle \quad = \quad (\langle\, A_1 \,\rangle \;;\; \langle\, A_3 \,\rangle) \sqcap (\langle\, A_2 \,\rangle \;;\; \langle\, A_3 \,\rangle)$$

The task of identifying a general collection of equivalence laws is left as future work.

5.5 Confidentiality Annotations

This section introduces confidentiality annotations — or CAs for short — which are algebraic constructs for incorporating obligations within lifted processes. A CA may be embedded at any point in the body of a lifted process, alongside lifted actions, to express a confidentiality property over the process state when the CA is invoked. By specifying confidentiality properties in this way, they become amenable to compositional reasoning: we can consider the effect of CAs on a process in terms of their effect on the behaviour of individual *Circus* actions.

5.5.1 Defining Confidentiality Annotations

The fog variables of a **UC**-lifted action track the information that Low can infer about the action's behaviour. The relation between the observable variables and fog variables can be employed for a secondary purpose: to capture the meaning of CAs within the lifted semantics directly.

Definition 5.38 formalises CAs with respect to the lifted semantics.

Definition 5.38 (Confidentiality annotation)**.**

$$\langle\, L : \theta \,\rangle \quad \triangleq \quad \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge (\ell = L \Rightarrow \mathsf{Conf}\,(\theta))$$

where $\mathsf{Conf}\,(\theta) \triangleq (\mathit{ok} \wedge \neg\, \mathit{wait} \Rightarrow \theta)$.

As with blocks, we abbreviate $\langle\, \ell : \theta \,\rangle$ to $\langle\, \theta \,\rangle$ if we wish to consider only a single Low user:

$$\langle\, \theta \,\rangle \quad = \quad \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta)$$

With respect to the observable behaviour of processes, a CA behaves as the lifted *Skip*: it terminates instantaneously, leaving the process trace and state unchanged.

Example 5.39. When invoked, the CA $\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$ specifies that the process must not reveal to Low that the state satisfies the condition $h = 0$, with states where $h > 0$ serving as cover stories. $\lozenge$

We can combine CAs with lifted *Circus* constructs using the lifted operators defined in Section 5.3. Hence, we can use CAs to impose confidentiality properties over the process state at intermediate points of the process's execution. Even though CAs are localised to a specific region of the process body, their effects diffuse throughout the process.

Typically, we place CAs in sequence with blocks, to impose confidentiality properties on the process state at specific points of its execution. For instance, to specify the obligation θ applies to the process state when the action A terminates, we would write:

$$\ldots\,;\,\langle\, A \,\rangle\,;\langle\, \theta \,\rangle\,;\,\ldots$$

CAs may be used in other ways. For instance, a process may be offered a choice between performing the action A or satisfying a CA:

$$\ldots\,;\,(\langle\, A \,\rangle \sqcap \langle\, \theta \,\rangle)\,;\,\ldots$$

5.5.2 Properties of Confidentiality Annotations

CAs inherit many of the properties of obligations. This is because the semantics of CAs induces an order isomorphism between the $\sqsubseteq$-ordering over obligations and the $\sqsubseteq$-ordering over CAs.

Theorem 5.40 (CAs and $\sqsubseteq$)**.** Provided θ_1, θ_2 reference only state variables:

$$\theta_1 \sqsubseteq \theta_2 \quad \text{if and only if} \quad \langle\, \theta_1 \,\rangle \sqsubseteq \langle\, \theta_2 \,\rangle$$

Hence, in the lifted semantics, the refinement ordering represents better confidentiality as well as better functionality.

From Definition 4.28, it follows that the sequential composition of CAs can be simplified to a CA embedding the least upper bound of their respective obligations.

Law 5.41 (CA least upper bound).

$$\langle\, \theta_1 \,\rangle \,;\, \ldots \,;\, \langle\, \theta_n \,\rangle \quad = \quad \langle\, \bigsqcup \{\theta_1, \ldots, \theta_n\} \,\rangle$$

Law 5.41 implies that CAs in sequence are commutative.

Law 5.42 (CA commutativity). $\langle\, \theta_1 \,\rangle \,;\, \langle\, \theta_2 \,\rangle = \langle\, \theta_2 \,\rangle \,;\, \langle\, \theta_1 \,\rangle$

Following Definition 4.29, a choice between multiple CAs is equivalent to the greatest lower bound of their respective obligations.

Law 5.43 (CA greatest lower bound).

$$\langle\, \theta_1 \,\rangle \sqcap \cdots \sqcap \langle\, \theta_n \,\rangle \quad = \quad \langle\, \bigsqcap \{\theta_1, \ldots, \theta_n\} \,\rangle$$

The semantics of CAs is closely connected with the semantics of lifted *Circus* coercions. Indeed, the CA is a generalisation of the lifted coercion, as Law 5.44 shows.

Law 5.44 (CA and coercion). $\langle\, [\, C \,] \,\rangle = \langle\, \mathbf{U}\,(C) \,\rangle$

Applying $\mathbf{D}$ to a CA yields a *Circus* coercion, as Law 5.45 shows.

Law 5.45 (CA and $\mathbf{D}$). $\mathbf{D}\,(\langle\, \theta \,\rangle) = [\, \mathbf{D}\,(\theta) \,]$

There is a kind of duality between CAs and *Circus* coercions. A lifted coercion $\langle\, [\, C \,] \,\rangle$ demands that both the process state and the fog state satisfy C, thus placing a *lower bound* on Low's inferences about the state. In contrast, CAs place an *upper bound* on Low's inferences.

It is often convenient to rewrite the sequential composition of a CA and a *Circus* action as a single compound action. Law 5.46 and Law 5.47 are concerned with folding a CA with a block that immediately precedes or succeeds it.

Law 5.46 (CA left sequence). $\langle\, \theta \,\rangle\, ; \langle\, A \,\rangle = \langle\, A \,\rangle \wedge \mathsf{Conf}\,(\theta)$

Law 5.47 (CA right sequence). $\langle\, A \,\rangle\, ; \langle\, \theta \,\rangle = \langle\, A \,\rangle \wedge \mathsf{Conf}'\,(\theta)$

where $\mathsf{Conf}'\,(\theta) \triangleq (ok' \wedge \neg\, wait' \Rightarrow \theta')$

5.5.3 Passive Confidentiality Annotations

A CA $\langle\, \theta \,\rangle$ is *passive* in a state ψ if $[\, \psi \Rightarrow \theta \,]$ holds; that is, θ does not constrain the fog space associated with ψ in any way. When passive, a CA's behaviour is just $\langle\, Skip \,\rangle$.

If a CA within a process body is passive in every state in which it can be invoked by the process, then we call that CA *innocuous* in the context of that process. CAs can also be innocuous if they are never invoked: for instance, $\langle\, Stop \,\rangle$ and $\langle\, Chaos \,\rangle$ are left zeros (with respect to sequential composition) for CAs, just as they are for lifted *Circus* actions.

An innocuous CA has no effect whatsoever upon lifted actions. Hence, innocuous CAs can be freely inserted or removed within a process, without changing its meaning.

Example 5.48. Suppose the CA from Example 5.39 follows the lifted command $\langle\, h := 1 \,\rangle$. Let:

$$B1 \quad \triangleq \quad \langle\, h := 1 \,\rangle\, ; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$$

The CA is innocuous in this context, as calculation shows:

$$\begin{array}{ll@{\qquad}r}
\multicolumn{2}{l}{\langle\, h := 1 \,\rangle\, ; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle} & \\
= & \langle\, h := 1 \,\rangle\, ; \langle\, [\, h = 1 \,] \,\rangle\, ; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & \text{[coercion introduction]} \\
= & \langle\, h := 1 \,\rangle\, ; \langle\, h = 1 \wedge \widetilde{h} = 1 \,\rangle\, ; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & \text{[Law 5.44]} \\
= & \langle\, h := 1 \,\rangle\, ; \langle\, h = 1 \wedge \widetilde{h} = 1 \wedge (h = 0 \Rightarrow \widetilde{h} > 0) \,\rangle & \text{[lub (Law 5.41)]} \\
= & \langle\, h := 1 \,\rangle\, ; \langle\, [\, h = 1 \,] \,\rangle & \text{[Law 5.44]} \\
= & \langle\, h := 1 \,\rangle & \text{[coercion removal]}
\end{array}$$

$\Diamond$

5.5.4 Active Confidentiality Annotations

A CA $\langle\, \theta \,\rangle$ is *active* in a state ψ if θ constrains the fog space associated with ψ. Operationally, the $\mathsf{Conf}\,(\theta)$ predicate prunes the fog of ψ, leaving only those fog states prescribed as cover stories by θ for ψ. This pruning commits the process design to supporting at least one (Low-indistinguishable) fog behaviour that passes through the CA in a cover story state.

The commitment made by a CA is broken if none of the cover story states prescribed by θ are present in the fog when the CA is invoked. *Breaking the commitment symbolises a conflict between the functionality of the process and its confidentiality properties.* This conflict becomes manifest in the following example.

Example 5.49. Suppose $B0$ names the CA from Example 5.39 following the command $h := 0$:

$$B0 \quad \triangleq \quad \langle\, h := 0 \,\rangle \;; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$$

Intuitively, the demands of functionality ($\widetilde{h} = 0$) and confidentiality ($\widetilde{h} > 0$) that $B0$ places on the fog space are irreconcilable. The consequences of this conflict are brought into stark relief by calculation:

$$\begin{array}{ll} \multicolumn{2}{l}{\langle\, h := 0 \,\rangle \;; \langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle} \\ = \quad \langle\, h := 0 \,\rangle \;; \langle\, h = 0 \wedge \widetilde{h} = 0 \wedge (h = 0 \Rightarrow \widetilde{h} > 0) \,\rangle & \text{[as Example 5.48]} \\ = \quad \langle\, h := 0 \,\rangle \;; \langle\, \mathbf{false} \,\rangle & \text{[contradiction]} \end{array}$$

This CA prunes *every* fog state. ◊

In a very real sense, the CA $\langle\, \mathbf{false} \,\rangle$ specifies the impossible. By Law 5.44 and the definition of coercions, we have:

$$\langle\, \mathbf{false} \,\rangle \quad = \quad \langle\, [\, \mathbf{false} \,] \,\rangle \quad = \quad \langle\, \mathbf{R}(\neg\, ok) \,\rangle$$

Recall from Section 2.6 that $\mathbf{R}(\neg\, ok) = \mathbf{R}\,(\mathbf{true} \vdash \mathbf{false})$ is the reactive design miracle. As in Section 4.4, miracles give meaning to process specifications where confidentiality properties are inconsistent with functionality properties. *By behaving miraculously, a CA disables insecure functionality.*

Example 5.50. The specification $\langle\, h := 0 \,\rangle \,;\langle\, \mathbf{false} \,\rangle$ from Example 5.49 can be simplified as follows:

$$\begin{aligned}
&\langle\, h := 0 \,\rangle \,;\langle\, \mathbf{false} \,\rangle \\
=\quad &\langle\, h := 0 \,\rangle \,;\, \langle\, [\,\mathbf{false}\,] \,\rangle && \text{[as above (Law 5.44)]} \\
=\quad &\langle\, h := 0;\, [\,\mathbf{false}\,] \,\rangle && \text{[Lemma 5.18]} \\
=\quad &\langle\, [\,\mathbf{false}\,] \,\rangle && \text{[property of coercions]}
\end{aligned}$$

which, of course, is unimplementable. ◊

The commitment made by a CA can also be broken later in a process's execution, again inducing miraculous behaviour, as Example 5.51 shows.

Example 5.51. Suppose that Low can observe values transmitted on the *out* channel. Consider the process fragment:

$$\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle \,;\, \langle\, out!h \rightarrow Skip \,\rangle$$

Were Low to observe $out!0$, Low could infer $h = 0$ with certainty. Hence, this specification behaves miraculously if $h = 0$. ◊

5.5.5 Discussion

Example 5.48 and Example 5.49 represent the extreme ends of a scale of the effects of CAs. The interplay between *Circus* actions and active CAs is often more subtle.

Example 5.52. Suppose a process is offered the choice between the (infeasible) $B0$ and (feasible) $B1$. Its behaviour is given by calculation:

$$\begin{aligned}
&B0 \sqcap B1 \\
=\quad &\mathbf{UC}\,(\mathcal{L}, \mathbf{R}(\neg\, ok)) \sqcap \mathbf{UC}\,(\mathcal{L}, h := 1) && \text{[Example 5.48, Example 5.49]}
\end{aligned}$$

$$= \quad \mathbf{UC}\,(\mathcal{L}, h := 1) \qquad \text{[lattice top]}$$
$$= \quad B1$$

The miraculous $B0$ is bypassed by the internal choice, because $B1$ is strictly weaker than $B0$. $\Diamond$

As Example 5.52 shows, the presence of miracles within a process body does not necessarily render the whole process miraculous. While a miracle may disable part of a process body, the process may remain feasible, so long as the process can "backtrack" from the miracle by resolving non-determinism. For instance, the *Circus* action $c \rightarrow [\,\mathbf{false}\,]$ cannot refuse to engage in an event c. Yet it can never actually accept c, because it would need to satisfy the reactive design miracle $[\,\mathbf{false}\,]$ were it to do so (Woodcock, 2010).

As Woodcock (2010) and Wei et al. (2010) point out, $c \rightarrow [\,\mathbf{false}\,]$ is a *Circus* action that lies outside the space of CSP processes, because it violates the axioms of the failures-divergences model of CSP. The same effect in the lifted space may be induced by CAs. For instance, $\langle\, c \rightarrow \mathit{Skip} \,\rangle\, ; \langle\, \mathbf{false} \,\rangle$ is equivalent to $\langle\, c \rightarrow [\,\mathbf{false}\,] \,\rangle$, so c is prevented from occurring.

We may interpret the meaning of miracles within process designs — whether induced by CAs or otherwise — in two ways:

1. A miraculous process specification cannot be implemented as a program; therefore, it should be deemed invalid. This interpretation reflects the philosophy of the conformance condition defined in Chapter 4: a process that fails to conform to the confidentiality properties in its specification does not satisfy its specification.

2. Alternatively, we may embrace specifications with miraculous elements, since they provide the designers of a process with greater flexibility in making implementation choices. Provided these miraculous elements can be avoided by the process, or are finessed away by refinement, their presence can be tolerated.

We adopt the second interpretation, because it is more conducive to the spirit of the UTP (Hoare, 1984a) and refinement calculi (Morgan, 1994). Furthermore, if we were to reject miraculous specifications as invalid, then we could only treat CAs as a form of documentation.

Remark 5.53. An interesting application of miraculous CAs is to direct the process's behaviour according to Low's inferences about its state. Let *Normal* stand for an arbitrary action in the specification:

$$\langle\, C \Rightarrow \neg\, \widetilde{C} \,\rangle \,;\, \langle\, \mathit{Normal} \,\rangle \quad \Box \quad \langle\, C \,\&\, \mathit{alarm} \rightarrow \mathit{Skip} \,\rangle$$

If Low can establish the state satisfies C, then the CA $\langle\, C \Rightarrow \neg\, \widetilde{C} \,\rangle$ is violated and this specification reduces to:

$$\langle\, \mathbf{false} \,\rangle \quad \Box \quad \langle\, \mathit{alarm} \rightarrow \mathit{Skip} \,\rangle$$

which forces the *alarm* to be raised *urgently*, because the miraculous $\langle\, \mathbf{false} \,\rangle$ forces the external choice to be resolved instantly. (This effect is a consequence of the semantics of external choice; it is explained in more detail by Woodcock (2010).) Alternatively, if the state satisfies $\neg\, C$, then the process behaves as $\langle\, \mathit{Normal} \,\rangle$. However, this specification does not prevent "false alarms": the environment may choose to raise the alarm in a state satisfying C, even if the CA is satisfied.

5.6 Event-Based Confidentiality Annotations

Up to now, we have studied how CAs can specify confidentiality properties over the state of a process. However, it is often the interactions that a process performs with its environment (or its other users) which are confidential. This section presents techniques for using CAs in order to express confidentiality properties over those interactions.

5.6.1 Inputs and Outputs

Be they inputs or outputs, the data values that a process communicates with its environment may be secret. It is straightforward to specify confidentiality properties over those values using CAs.

The semantics of an input prefixing $c?i \rightarrow A$ declares $i?$ as a local variable in the context of A. This variable can be incorporated into a CA within the scope of the input prefixing.

Example 5.54. The following process fragment specifies that Low cannot deduce if a value transmitted on the c channel is a member of the set S:

$$\langle\, c?i \rightarrow h := i? \,\rangle \; ; \langle\, h \in S \Rightarrow \widetilde{h} \notin S \,\rangle$$

This CA permits Low to infer the occurrence of the event. $\Diamond$

With an output event $c!E$, the variables within the expression E can just be referenced by a CA directly.

5.6.2 Occurrence of Events

Other kinds of confidentiality properties — such as the noninterference properties discussed in Section 4.6 — are concerned with restricting Low's inferences about whether a process has (or has not) engaged in particular events. In the literature, these properties are typically formulated over the traces over a process. Since the process trace is a product of the whole process, we cannot encode trace-based properties using CAs directly.

Fortunately, the symbiosis between the state and behaviour of *Circus* processes allows information about the control flow of the process to be recorded with state variables. In essence, we can use a local variable to record whether a particular interaction is performed by the process. This variable can then be referenced by a CA to indirectly encode what Low is entitled to infer about those interactions. This scheme is illustrated briefly by Example 5.55.

Example 5.55. Recall the specification *OO* given in Example 5.10, joined with the CA from Example 5.39:

$$\langle\, on \rightarrow h := 0 \;\Box\; \mathit{off} \rightarrow h := 1 \,\rangle \;;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$$

This specification stipulates that Low cannot be sure that *on* has occurred: it demands that, if *on* does occur, then Low must be unable to rule out the occurrence of *off* instead. ◊

A process need not record its control flow in terms of its state variables, but by introducing local variables, it can always be transformed into an equivalent process that does. In this way, we can formulate various confidentiality properties over the interactions a process performs.

Example 5.56. Consider the following choice construct:

$$\langle\, sec \rightarrow A_1 \;\Box\; cov \rightarrow A_2 \,\rangle$$

Suppose that *sec* is a high-level event whose occurrence is deemed to be secret. If we regard the occurrence of *cov* (implying the non-occurrence of *sec*) as a suitable cover story, then we can formulate a confidentiality property by extending the choice construct with a local variable as follows:

$$\mathbf{var}\; h \bullet \left(\begin{array}{l} \langle\, sec \rightarrow h := 0 \;\Box\; cov \rightarrow h := 1 \,\rangle \;; \\ \langle h = 0 \Rightarrow \widetilde{h} > 0 \rangle \,; \\ \langle\, h = 0 \,\&\, A_1 \;\Box\; h = 1 \,\&\, A_2 \,\rangle \end{array} \right)$$

It is important to invoke the CA immediately after *sec* takes place, for otherwise it would not take effect if A_1 deadlocks. ◊

With this scheme, it is necessary to extend the process with a separate CA wherever a confidential interaction may take place. Moreover, in a realistic *Circus* process, the secret and cover story events need not be located within the same choice construct, so determining how the process should be extended with a CA may require some creativity.

We now propose a compositional method for structuring the specification of confidentiality properties over the events of a process.

5.6.3 Superposition

CAs over the events of a process can be packaged together in a *confidentiality specification*. The purpose of a confidentiality specification is to monitor the activity of a process and to invoke CAs whenever the process engages in a confidential interaction.

We formalise a confidentiality specification as a recursive construct that is always ready to engage in a set of high-level events. Here, we focus our attention on confidentiality specifications expressible in the form:

$$\mu X \bullet \mathbf{var}\, h : E \bullet \langle \Box\, e : E \bullet c.e \to h := e \rangle \,;\, \langle \theta \rangle \,; X$$

where E denotes the type of channel c; h is a variable of type E; and θ is an obligation over h and $\widetilde{h}$.

Remark 5.57. More sophisticated kinds of confidentiality specifications could be defined with a local state, to record the activity of the process in detail and to impose CAs when particular patterns arise.

Suppose we place the confidentiality specification above in parallel composition (synchronising on channel c) with a lifted process. Then, whenever the process accepts an event $c.e$, the same event invokes the CA in the confidentiality specification with h set to e. In this way, the confidentiality property defined by the CA is imposed upon the process.

We call this method *superposition*, as it is closely related to the superposition method for program structuring proposed by Chandy and Misra (1988), Back and Sere (1996) and others. Superposition promotes separation of concerns: it enables us to specify the confidentiality requirements of a system without regard to its functionality, and vice versa.

The following two examples show how the confidentiality properties described in Example 5.54 and Example 5.56 can be reformulated as confidentiality specifications.

Example 5.58. Continuing from Example 5.54, a confidentiality specification over the values communicated over channel c is as follows:

$$\mu X \bullet \mathbf{var}\, h \bullet \langle\, c?i \rightarrow h := i? \,\rangle \,;\langle\, h \in S \Rightarrow \widetilde{h} \notin S \,\rangle \,; X$$

Again, this specification does not prohibit Low from inferring that events on c have taken place. ◊

Example 5.59. The confidentiality specification for the choice construct described in Example 5.56 is:

$$CS = \mu X \bullet \mathbf{var}\, h \bullet \left\langle \begin{array}{ll} & sec \rightarrow h := sec \\ \Box & cov \rightarrow h := cov \end{array} \right\rangle ;\langle\, h = sec \Rightarrow \widetilde{h} = cov \,\rangle \,; X$$

The superposed specification is:

$$\langle\, sec \rightarrow A_1 \,\Box\, cov \rightarrow A_2 \,\rangle \,|[\, \{\!|\, sec, cov \,|\!\} \,]|\, CS$$

This superposed specification is not identical to the specification given in Example 5.56. If A_1 or A_2 engage in sec events, then the confidentiality specification will apply to those events as well. ◊

We can apply superposition to express noninterference-like confidentiality properties (Subsection 4.6.1), where the occurrence of a high-level event is secret, and the non-occurrence of that event is the cover story. Example 5.60 illustrates this technique, in a manner reminiscent of Roscoe's lazy abstraction (Subsection 3.5.3).

Example 5.60. Let Q denote a lifted process which communicates sec events. The process:

$$QQ \quad = \quad Q \,|||\, \langle\, \mu X \bullet cov \rightarrow X \,\rangle$$

behaves as Q, but in addition may perform a cov event at any time. Taking the confidentiality specification from Example 5.59, the superposition:

$$(QQ \,|[\, \{\!|\, sec, cov \,|\!\} \,]|\, CS) \setminus \{\!|\, cov \,|\!\}$$

specifies that Low cannot infer that QQ — and therefore Q — has performed a *sec* event. In this way, we impose the noninference property (described in Subsection 4.6.1) on *sec* events of Q. ◊

5.6.4 Discussion

Superposing a confidentiality specification to a process achieves the same effect as modifying the process by attaching a separate CA to each high-level event. Hence, a superposition of a confidentiality specification over a lifted process can be re-expressed as a single integrated process specification, by applying the step laws of parallel composition.

It would be worthwhile to create a library of reusable confidentiality specifications, to encapsulate common "design patterns" for encoding confidentiality requirements. Without needing to understand their technical detail, software engineers could compose these confidentiality specifications with *Circus* processes, to realise secure process designs.

5.7 Declassification

Up to now, we have treated the confidentiality of information as a static property. However, the secrecy of information may change over time. Information ceases to be secret when it is *declassified*.

Declassification may ease the implementation of the system. For instance, if information is deemed to be valueless to an adversary — such as old passwords or expired cryptographic keys — then it may be acceptable to reveal that information to the adversary. In other cases, the ability to declassify data is often essential to avoid conflicts with a system's functionality. For instance, the questions of an examination paper should be declassified to candidates at the moment when the exam starts, while sample answers may be declassified after the exam ends.

Declassification is not an issue for the confidentiality framework presented in Chapter 4. In that framework, if an obligation does not constrain the fog space associated with a particular behaviour of a system, then that behaviour is implicitly unclassified. However, once a CA is invoked, the confidentiality property it imposes remains in force in perpetuity.

In this section, we discuss a simple approach for declassifying confidentiality properties within lifted processes. This approach relies on expressing CAs in a way such that they can be disabled by other constructs within a process specification. However, we have not solved the problem of defining a semantics for these constructs that is consistent with the reactive design model of *Circus*, which is a significant shortcoming of the approach.

5.7.1 Relaxing Confidentiality Annotations

Recall from Subsection 4.2.1 the dual purposes of the fog space: to model Low's inferences; and to capture confidentiality properties. As we have modelled them, these concerns are not separable from each other, which impedes the declassification of confidentiality properties.

When a CA is invoked, it cuts away fog behaviours that do not represent acceptable cover stories. By cutting down the fog space, the CA destroys information about Low's inferences encoded in the semantics of lifted constructs. Once lost, this information cannot be put back into the lifted space at any later point in the process's execution. For this reason, the semantics of CAs rules out a general scheme for declassifying them once they have taken effect.

It is possible to prevent CAs from taking effect when they are invoked. Consider a CA of the form $\langle\, c \Rightarrow \theta \,\rangle$ where c denotes a Boolean variable. This CA is innocuous if c is **false**, but is equivalent to $\langle\, \theta \,\rangle$ if c is **true**. Hence, the CA is *contingent* on c, because c controls whether the CA imposes θ upon the process state.

Making a CA contingent on c does not solve the problem of relaxing that CA after its invocation. If c is an ordinary state variable, then setting its value after the CA is invoked does not affect the CA. However, if c is not a state variable, but a variable whose value is determined by the control flow of the process, then it could be used to declassify the CA once the process has reached a particular point in its execution.

5.7.2 A Declassification Construct

We postulate a special specification construct, **dec** c, which can coerce the value of c according to whether the construct is invoked by the process.

Definition 5.61 (**dec** statement)**.**

$$\mathbf{dec}\, c \quad \triangleq \quad Skip \wedge c \Leftrightarrow wait'$$

The statement **dec** c specifies that the truth of c is contingent on whether the statement is invoked by the process. Here, c is not an ordinary state variable: c is **false** if the process invokes **dec** c during its execution; and c is **true** if the process halts without executing **dec** c. Hence, the invocation of **dec** c *retroactively* declassifies any CA contingent on c.

Example 5.62. Here is a fragment of a specification for a (very simple) examination management system:

$$\mathbf{var}\; dq, ds \bullet \left(\begin{array}{l} set\_paper?q.s \rightarrow \\ \quad \langle\, Student \,:\, dq \Rightarrow \theta Questions(q?) \,\rangle\,; \\ \quad \langle\, Student \,:\, ds \Rightarrow \theta Solutions(s?) \,\rangle\,; \\ \ldots \\ \langle\, \mathbf{dec}\, dq\,; start\_exam!q \rightarrow Skip \,\rangle\;; \\ \langle\, Run\_Exam \,\rangle\;; \\ \langle\, \mathbf{dec}\, ds\,; end\_exam!s \rightarrow Skip \,\rangle \end{array} \right)$$

Here, a confidentiality property is applied to the question paper (q) and solution sheet (s) upon creation. The question paper is declassified immediately before *start_exam*, while the solution is declassified on *end_exam*. Without declassification, the CAs would prevent *start_exam* and *end_exam* from taking place. ◊

Unfortunately, the **dec** construct is not a reactive design, because it is not **R3**-healthy:

$$\mathbf{R3}(\mathbf{dec}\,c) \quad = \quad (I\!I_{rea} \lhd wait \rhd \mathbf{dec}\,c) \quad \neq \quad \mathbf{dec}\,c$$

Furthermore, the sequential composition of **dec** c and a reactive design A is not a reactive design, because the constructs:

$$\begin{aligned} A\,;\mathbf{dec}\,c \quad &= \quad A \wedge c = wait' \\ \mathbf{dec}\,c\,;A \quad &= \quad A \wedge c = wait \end{aligned}$$

are again not **R3**-healthy.

Since **dec** breaks the reactive design model of *Circus*, we lose the assurance that the laws of *Circus* actions continue to hold in the presence of **dec** constructs. In order to deploy **dec**-like constructs in a rigorous manner, it would be necessary to redefine the semantic model of *Circus* and to re-prove existing results. Hence, we must abandon **dec**, because to do otherwise would force us to sacrifice the semantic basis of the language. We leave the task of defining a workable semantics for a declassification construct to future work.

An alternative strategy for accommodating declassification would be to extend the lifted semantics with another alphabet of fog variables, for the sole purpose of tracking Low's inferences about a process. Then, to declassify a preceding CA, the fog variables can be overwritten with these backup variables, to nullify the effect of the CA. However, this approach also has its drawbacks; most notably, the extended alphabet would necessitate redefining the lifting functions, as well as the semantics of lifted operators and CAs. In turn, tracking the extra variables would complicate analysis of specifications at the semantic level.

5.8 Related Work

The lifted semantics for *Circus* presented in this chapter has much in common with Morgan's *shadow semantics* (Morgan, 2009, 2012), which extends the refinement calculus for sequential programs (Morgan, 1994). The shadow semantics has been applied by McIver (2009) and McIver and Morgan (2010) to derive secure implementations of security-critical algorithms and communication protocols by stepwise refinement.

In the shadow semantics, the program state is divided into a secret variable h and a non-secret variable v. It is assumed that Low can monitor the value of v (but not h) at each point in the program's execution. In addition, a special "shadow set" state variable H is defined to contain all values of h that are consistent with Low's observation of the program's execution. The semantics of each program construct is defined to update v, h and H as appropriate.

In some respects, the lifted semantics of *Circus* actions is a generalisation of the shadow semantics. An analogue of the shadow set is achieved (over all the observational variables) by incorporating the $\mathcal{I}(\mathcal{L})$ relation within the lifting function for *Circus* actions. While Morgan (2009) defines a shadow semantics for sequential programs, our approach is applicable to the wider domain of *Circus* actions. This makes our approach extensible to other languages in the *Circus* family, or indeed to any language with a UTP semantics.

We make an important deviation from the philosophy of the shadow semantics by not differentiating between secret and non-secret variables at the semantic level. On the contrary, confidentiality annotations allow software engineers to specify exactly which properties of the state — and by extension, the behaviour — of processes are secret. This is arguably a more flexible (albeit more involved) approach for integrating confidentiality properties within formal methods.

McIver (2009) has formulated a relative of CAs for the shadow semantics. A *visibility annotation* cuts down the shadow associated with a to only the actual value of a, thereby modelling that value as revealed to Low. Visibility annotations can be formulated in terms of CAs:

$$\textbf{reveal}\, a \quad \triangleq \quad \langle\, a = \widetilde{a}\, \rangle$$

We hypothesise that specialised annotation constructs may be useful for specifying other kinds of security properties besides confidentiality. Indeed, a family of annotation-like operators to specify a class of availability properties over CSP and timed *Circus* processes has been proposed by Woodcock (2010) and Wei et al. (2010). However, such prospects are beyond the scope of this thesis, and we do not pursue them further.

5.9 Conclusion

This chapter has presented an extension of the *Circus* language which integrates confidentiality properties with *Circus* processes. This extended language inherits the semantic basis of the confidentiality framework of Chapter 4. In particular, we have retained the notion of using miracles to denote inconsistencies between functionality and confidentiality. Hence, processes in the extended language are incapable of engaging in functionality that would (directly or indirectly) reveal information classed as secret to low-level users.

The confidentiality annotation is a versatile and elegant construct for specifying confidentiality properties over both the state and behaviour of *Circus* processes. This construct sets the language apart from the framework of Chapter 4 by the compositional manner in which confidentiality properties are joined with the functional specification facilities of *Circus*. In turn, the specification of confidentiality properties can be localised to security-critical regions of the process body. In this way, the language overcomes two of the problems described in Section 4.8.

While we have taken *Circus* as the formal foundation of our language, we are confident that its underlying principles could be translated to other formalisms. Nevertheless, the generality of *Circus* makes our language (in its current form) suitable for embedding confidentiality within sequential programs and concurrent processes alike.

There remains the problem, described in Section 4.8, of verifying that the functionality of a process in the extended language is consistent with its embedded CAs. We address this final problem in the next chapter.

6 Secure Software Development

6.1 Introduction

Chapter 5 focused on extending *Circus* with a syntax and semantics for specifying confidentiality properties. In this chapter, we turn our attention towards an engineering method to guide and support the development of process designs expressed in this extended language. This method builds upon the *Circus* refinement strategy, but with additions to maintain faithfulness with the lifted semantics.

Recall from Section 4.5 that, in lifted UTP theories, refinement preserves confidentiality properties encoded as obligations, because unfulfilled obligations induce miraculous behaviour. Section 6.2 explores this issue in the context of lifted *Circus* processes. It argues that confidentiality annotations deserve special attention during the development of a process, to avoid jeopardising the implementability of the process.

Before embarking upon the development of a process in our extended language, it is expedient to verify that its functionality is consistent with its embedded CAs. Section 6.3 presents a procedure for calculating Low's inferences about the state of the process at each point of its execution. In Section 6.4, this procedure is distilled into laws defined over lifted *Circus* actions. In Section 6.5, we describe how this procedure can be used to verify that a *Circus* process satisfies its CAs.

In Section 6.6, we tailor the *Circus* refinement strategy to accommodate verified processes in the lifted language. This *secure refinement strategy* enables software engineers to apply the laws of the *Circus* refinement calculus to lifted processes, safe in the knowledge that refinement steps maintain the consistency of the process.

The final development step is to translate a concrete-level process design into executable code. However, there are important caveats which mean that assurances about the security of the process design may not be applicable to the code. We discuss these issues in Section 6.7.

In this chapter, the development strategy is primarily concerned with safeguarding confidential information from a single adversarial user. The techniques presented here are readily extendible to cover multiple low-level users, who may be encoded within a process specification using blocks of different families (Subsection 5.4.3). Indeed, the case study of Chapter 7 involves a development where the confidentiality properties apply to multiple users' inferences.

6.2 Confidentiality-Preserving Refinement

Recall from Section 5.5 that a CA makes miraculous those behaviours of a process that would disclose secret information to Low. Since a process is compelled not to behave miraculously, the process is forced to satisfy the confidentiality property embodied by the CA.

Suppose a lifted process Q is refined in a way that removes all cover story behaviours prescribed by a CA embedded in Q. The refined process Q' violates the CA and, by definition, the CA becomes miraculous. In turn, by behaving miraculously, the CA disables all behaviours of Q' that would leak secret information. The refinement of Q to Q' is still legitimate, because miracles occupy the top of the lattice of (lifted) reactive designs. However, such refinements *cannot make a process insecure*, because processes cannot behave miraculously. For the reasons described in Section 4.5, it follows that *refinement is confidentiality-preserving* in the lifted semantics.

6.2.1 Refinement and Feasibility

While refinement is confidentiality-preserving, it must be used with care. In the worst case, a refinement step may come at the price of sacrificing the feasibility of the process, making it impossible to implement (and therefore useless). This phenomenon is made clear by Example 6.1.

Example 6.1. Recall from Example 5.48 and Example 5.49 that:

$$\begin{array}{lclcl} B0 & \triangleq & \langle\, h := 0 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & = & \langle\, \mathbf{R}(\neg\, ok) \,\rangle \\ B1 & \triangleq & \langle\, h := 1 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & = & \langle\, h := 1 \,\rangle \end{array}$$

Now consider the specification:

$$B01 \quad \triangleq \quad \langle\, h := 0 \sqcap h := 1 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$$

$B01$ is feasible in isolation: Low cannot distinguish between $h := 0$ and $h := 1$ by observing the behaviour of $B01$ alone. Both $B0$ and $B1$ are refinements of $B01$, but only $B1$ is a feasible refinement of $B01$. ◊

Refinement is not guaranteed to maintain the feasibility of processes containing CAs. This is because removing cover stories may induce conflicts between CAs and their surrounding contexts, making those contexts miraculous. On the other hand, refinement may also induce miracles into any process with partial specification constructs such as specification statements and coercions (Nelson, 1989; Morgan, 1994). The next example replicates the phenomenon shown in Example 6.1, but using *Circus* coercions in place of CAs.

Example 6.2 (adapted from Zeyda et al. (2003)). Working in the *Circus* semantics, the action $h := 1$ can be rewritten using a coercion:

$$\begin{array}{lll} (h := 0 \sqcap h := 1)\,;\,[\,h = 1\,] & & \\ = \quad (h := 0\,;\,[\,h = 1\,]) \sqcap (h := 1\,;\,[\,h = 1\,]) & & [\text{distributivity}] \\ = \quad \mathbf{R}(\neg\, ok) \sqcap h := 1 & & [\text{Example 2.4}] \\ = \quad h := 1 & & [\text{lattice top}] \end{array}$$

While refining the $h := 0 \sqcap h := 1$ action to $h := 0$ is feasible in isolation, the resulting specification — $h := 0\,;\,[\,h = 1\,]$ — is miraculous, as we saw in Example 2.4. $\Diamond$

To support confidentiality-preserving refinement with a practical development method, we need to employ special measures to verify whether a refinement step, applied to one part of a process, maintains the feasibility of the process as a whole. These measures — which are the main topic of this chapter — depend on analysing the information that Low can infer about the process state at each point during the process's execution.

6.2.2 Atomic and Composite Non-Determinism

Information flow to Low can be modelled at different levels of granularity. We introduce these levels of granularity by drawing an analogy with Morgan's shadow semantics, which was discussed in Section 5.8.

The shadow semantics grants Low the ability to monitor the control flow of a program. Non-deterministic specification constructs in the shadow semantics are either *atomic* or *composite*:

Atomic constructs are considered to execute instantaneously, so Low cannot observe how non-determinism is resolved during their execution. These constructs are needed for specifying security-critical elements of a program, such as conditional branches that depend on the values of secret variables.

Composite constructs enable Low to monitor how they resolve non-determinism. Hence, refining away non-determinism within a composite construct does not reveal more information about the construct's behaviour to Low.

We can express atomic and composite non-deterministic constructs in the lifted semantics. The atomic construct denoting non-deterministic choice between A_1 and A_2 is represented by:

$$\langle A_1 \sqcap A_2 \rangle$$

while the corresponding composite construct is represented by:

$$\langle A_1 \rangle \sqcap \langle A_2 \rangle$$

Morgan (2009) defines *ignorance-preserving refinement* over the shadow semantics, which mandates the shadow set H can never be decreased by refining any component of a program. In other words, a refinement is ignorance-preserving if it reveals no more information to Low about the secret variable h. Under ignorance-preserving refinement, composite constructs can be refined into atomic constructs, because doing so can only increase Low's ignorance about a program's behaviour. However, Morgan forbids the refinement of atomic constructs in any way.

Owing to ignorance-preserving refinement, a program design in the shadow semantics *implicitly* specifies the maximal information about h that is permitted to flow to Low. On the contrary, CAs provide an *explicit* specification of the maximal information flow to Low. Therefore, we do not insist that refinement is ignorance-preserving, because CAs disable insecure functionality of a process. It follows that atomic choice can be refined to composite choice in our semantics:

$$\langle A_1 \sqcap A_2 \rangle \quad \sqsubseteq \quad \langle A_1 \rangle \sqcap \langle A_2 \rangle$$

which is a consequence of Condition 5.12.

By making this refinement, we sever the indistinguishability relation between the observable behaviours of A_1 and the fog behaviours of A_2 (and vice versa). Intuitively, the specification $\langle A_1 \rangle \sqcap \langle A_2 \rangle$ discloses to Low how the process resolves the choice between A_1 and A_2. (It is as though Low can observe the "program counter" to determine which of A_1 and A_2 is executed.)

A consequence of this refinement is that $\langle A_1 \rangle \sqcap \langle A_2 \rangle$ reveals at least as much information to Low as does $\langle A_1 \sqcap A_2 \rangle$. For this reason, this refinement may violate a CA elsewhere within a process, so it is not guaranteed to preserve the feasibility of the process.

6.2.3 Equivalence

There are some lifted constructs whose atomic and composite forms are equivalent, as Example 6.3 shows.

Example 6.3. Recall from Example 5.10 that a lifted action can be unfolded according to the $\mathcal{I}(\mathcal{L})$ relation. By extension, given that L includes the *on* or *off* channels, we have:

$$\langle\, L \,:\, on \rightarrow h := 0 \sqcap off \rightarrow h := 1 \,\rangle \quad = \quad \left(\begin{array}{cc} & \langle\, L \,:\, on \rightarrow h := 0 \,\rangle \\ \sqcap & \langle\, L \,:\, off \rightarrow h := 1 \,\rangle \end{array} \right)$$

Since Low can distinguish an *on* event from an *off* event (by observing termination), it follows that Low can always determine how the choice is resolved within this specification. ◊

Example 6.3 motivates the following law.

Law 6.4 (Split choice)**.** Provided L includes the channels c_1 and c_2:

$$\langle\, L \,:\, c_1 \rightarrow A_1 \sqcap c_2 \rightarrow A_2 \,\rangle \quad = \quad \langle\, L \,:\, c_1 \rightarrow A_1 \,\rangle \sqcap \langle\, L \,:\, c_2 \rightarrow A_2 \,\rangle$$

Where possible, it is advantageous to refine an atomic construct into a composite construct, because we can analyse the components of the composite construct in isolation from each other. Moreover, in cases where atomic and composite constructs are semantically equivalent, substituting one for the other cannot change the feasibility of the process.

6.3 Backwards Propagation

As we observed in Chapter 4, confidentiality properties are not closed under composition. Thus, it can be tricky to determine if a CA is violated within a process body, because the CA imposes a confidentiality constraint that extends over the whole of a process.

This section describes a technique for *compacting* our model of Low's inferences about the behaviour of lifted *Circus* actions into indistinguishability relations over the process state. These indistinguishability relations have the same form as obligations, so they can be combined with a CA to identify which cover stories prescribed by the CA are not discarded by the action's behaviour. In turn, we can *propagate* this information through a process, to calculate the confidentiality constraints that apply over the process state at each point in the process's execution.

Section 6.5 explains how propagation can help to warn software engineers if the functionality and confidentiality attributes of a process specification are mutually inconsistent.

6.3.1 Weakest Reactive Preconditions

In predicate transformer semantics (Dijkstra, 1976), **wp** $(P, Post)$ denotes the *weakest precondition* which guarantees that program P terminates in a state satisfying $Post$. Following Hoare and He (1998), **wp** can be defined in the UTP theory of relations as follows:

$$\mathbf{wp}\,(P, Post) \quad \triangleq \quad \forall\, x' \bullet P \Rightarrow Post$$

The following predicate transformer describes all initial states of a *Circus* action A such that *every* normal execution of A started in any such state — regardless of whether A diverges, deadlocks or terminates — is guaranteed to reach an observable state satisfying the postcondition $Post$:

$$\begin{aligned} \mathbf{wrp}\,(A, Post) \quad &\triangleq \quad \mathbf{wp}\,(A, (ok \wedge \neg\, wait \Rightarrow Post)) \\ &= \quad \forall\, x' \bullet (ok \wedge \neg\, wait \wedge A) \Rightarrow Post \end{aligned}$$

wrp $(A, Post)$ is called the *weakest reactive precondition* (for normal behaviour) by Cavalcanti and Woodcock (2003); we adopt that name here.

6.3.2 Masking States

Propagation involves the translation of Low-indistinguishable process behaviours into a notion of Low-indistinguishable process states. We say that a fog state $\widetilde{\psi}$ *masks* a state ψ through A if and only if, for each behaviour of A started in ψ, there is a counterpart Low-indistinguishable fog behaviour of $\widetilde{A}$ started in $\widetilde{\psi}$.

Definition 6.5 (Masking)**.** $\widetilde{\psi}$ masks ψ through A (with respect to $\mathcal{L}$) if:

$$\left[A \wedge \psi \Rightarrow \exists\, \widetilde{x}, \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, A) \wedge \widetilde{\psi} \right]$$

Given that $\widetilde{\psi}$ masks ψ, if Low's interaction is consistent with A starting in ψ, then Low cannot rule out the possibility that A started in $\widetilde{\psi}$ instead.

Example 6.6. Given that Low can observe *on* and *off* events, consider:

$$\langle\, g \,\&\; (on \rightarrow Skip \sqcap off \rightarrow Skip) \;\Box\; \neg\, g \,\&\; on \rightarrow Skip \,\rangle$$

Here, each state satisfying the condition g masks each state satisfying $\neg\, g$, because *on* may be performed in either case. Notice that symmetry does not apply here, because the occurrence of *off* would reveal to Low that g holds. ◊

We can adapt **wrp** to encode Low's inferences about the behaviour of a *Circus* action in terms of its initial state. The predicate:

$$\mathbf{wrp}\left(A, \exists\, \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, A)\right)$$

describes the *weakest masking relation* over the initial states of action A: it maps ψ to $\widetilde{\psi}$ if and only if $\widetilde{\psi}$ masks ψ.

We now define a specialised form of the weakest masking relation over lifted constructs. In order to simplify the presentation of the results to follow, it hides the $ok, \widetilde{ok}, wait, \widetilde{wait}$ variables.

Definition 6.7 (**wmr** predicate transformer)**.**

$$\mathbf{wmr}\,(B) \quad \triangleq \quad \left(\forall\, x' \bullet \mathbf{D}\,(B!) \Rightarrow \exists\, \widetilde{x'} \bullet B!\right)$$

where $B!$ denotes $B[\mathbf{true}, \mathbf{true}, \mathbf{false}, \mathbf{false} / ok, \widetilde{ok}, wait, \widetilde{wait}]$.

wmr (B) encodes the maximum information that Low can deduce about the initial state of B, regardless of the interaction that Low makes with B. Hence, **wmr** (B) may potentially over-approximate Low's inferences: not every behaviour of B may reveal all the information to Low modelled by **wmr** (B). Example 6.8 clarifies the nature of this over-approximation.

Example 6.8. Consider the block $ST = \langle\, s!a \to Skip \sqcap t!b \to Skip \,\rangle$. Given that Low's window includes the s and t channels, we have:

$$\textbf{wmr}\,(ST) \quad = \quad a = \widetilde{a} \wedge b = \widetilde{b}$$

Intuitively, Low may learn by observing ST either the value of a (on channel s) or the value of b (on channel t), depending on how the non-determinism in ST is resolved. However, before ST starts, there is no way of knowing how that non-determinism is resolved. Hence, **wmr** over-approximates Low's inferences about the initial state of ST.

Now consider the block $TT = \langle\, t!a \to Skip \sqcap t!b \to Skip \,\rangle$. We have:

$$\textbf{wmr}\,(TT) \quad = \quad a = \widetilde{a} \vee b = \widetilde{b}$$

By observing a value transmitted on channel t, Low learns only that either a or b have that value, in the absence of other information. $\Diamond$

Since **wmr** $(\langle\, A \,\rangle)$ expresses an upper bound on Low's inferences about A's initial state, it is not monotonic with respect to the $\sqsubseteq$ ordering. This non-monotonicity of **wmr** is demonstrated by the next example.

Example 6.9. The ST block (Example 6.8) is refined by $\langle\, s!a \to Skip \,\rangle$. This block reveals the value of a to Low, but tells Low nothing about b:

$$\textbf{wmr}\,(\langle\, s!a \to Skip \,\rangle) \quad = \quad a = \widetilde{a}$$

We have $ST \sqsubseteq \langle\, s!a \to Skip \,\rangle$, but **wmr** $(ST) \not\sqsubseteq$ **wmr** $(\langle\, s!a \to Skip \,\rangle)$, so **wmr** is not monotonic. $\Diamond$

6.3.3 Propagating Obligations

We now turn our attention to propagating obligations backwards through blocks. Consider a block placed in sequence with a CA:

$$B\,;\langle\,\theta\,\rangle$$

Should B terminate, then $\langle\,\theta\,\rangle$ takes effect and constrains the fog space associated with the final state of B according to the obligation θ.

We extend the **wmr** predicate transformer to propagate θ backwards through B. Let $\mathbf{bwQ}\,(B,\theta)$ denote the masking relation over the initial states of B, but strengthened with an obligation parameter.

Definition 6.10 (bwQ predicate transformer).

$$\mathbf{bwQ}\,(B,\theta) \quad \triangleq \quad \forall\, x' \bullet \Big(\mathbf{D}\,(B!) \Rightarrow \exists\, \widetilde{x'} \bullet B! \wedge \mathsf{Conf}'\,(\theta)\Big)$$

If a *Circus* action A starts in a state ψ and terminates in a state ψ' satisfying the postcondition $\mathbf{D}\,(\theta')$, then $\mathbf{bwQ}\,(\langle\,A\,\rangle\,,\theta)$ relates ψ to only those initial fog states that mask ψ through $\langle\,A\,\rangle$ and lead to Low-indistinguishable behaviours terminating in fog states marked as cover stories for ψ' by θ'. In fact, **bwQ** is more general: it can be applied to arbitrary lifted constructs to identify an indistinguishability relation between initial states.

The masking relation $\mathbf{bwQ}\,(B,\theta)$ is itself an obligation, which fuses Low's inferences about the behaviour of B with the confidentiality property imposed upon those inferences by $\langle\,\theta\,\rangle$ whenever B terminates.

Example 6.11. Suppose ST is named in sequence with a CA:

$$\langle\, s!a \rightarrow Skip \sqcap t!b \rightarrow Skip\,\rangle\;;\langle\, a = 0 \Rightarrow \widetilde{a} > 0\,\rangle$$

Intuitively, the CA prevents the $s!a$ event from taking place if $a = 0$. This intuition is justified by calculation:

$$\begin{array}{ll} & \mathbf{bwQ}\,((s!a \rightarrow Skip \sqcap t!b \rightarrow Skip), (a = 0 \Rightarrow \widetilde{a} > 0)) \\ = & a = \widetilde{a} \wedge b = \widetilde{b} \wedge (a = 0 \Rightarrow \widetilde{a} > 0) \qquad \text{[by Example 6.8]} \end{array}$$

$$= \quad a = \widetilde{a} \land b = \widetilde{b} \land a \neq 0 \qquad \text{[prop calc]}$$

No cover stories are prescribed by this back-propagated obligation when $a = 0$. $\Diamond$

Theorem 6.12 shows the result of applying **bwQ** to a lifted reactive design.

Theorem 6.12 (Unfolding **bwQ**).

$$\begin{aligned}&\mathbf{bwQ}\,(\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post)), \theta) \quad = \\ &\quad \forall x' \bullet \left(\begin{array}{l} Pre! \land ok' \land Post! \land tr \leq tr' \\ \quad \Rightarrow \exists \widetilde{x'} \bullet (\widetilde{Pre}! \Rightarrow \widetilde{Post!}) \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'(\theta) \end{array}\right) \\ &\land \quad \widetilde{Pre}! \Rightarrow Pre\end{aligned}$$

If B always exhibits miraculous behaviour in some initial states, then Low can rule out the possibility that B starts in those states. The initial states where execution of B is feasible are given by $\mathbf{bwR}\,(B)$.

Definition 6.13 (**bwR** predicate transformer).

$$\mathbf{bwR}\,(B) \quad \triangleq \quad \exists x', \widetilde{x'} \bullet B!$$

Whenever B is a non-miraculous construct, we have $\mathbf{bwR}\,(B) = \mathbf{true}$. Theorem 6.14 shows the result of applying **bwR** to a lifted reactive design.

Theorem 6.14 (Unfolding **bwR**).

$$\begin{aligned}&\mathbf{bwR}\,(\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post))) \quad = \\ &\exists x', \widetilde{x'} \bullet \mathbf{U}\,(tr \leq tr') \land ((Pre! \lor \widetilde{Pre}!) \Rightarrow ok' \land \mathbf{U}\,(Pre! \Rightarrow Post!)) \land \mathcal{I}(\mathcal{L})!\end{aligned}$$

We define a predicate transformer **bw**, which conjoins **bwQ** with **bwR** to exclude initial states which lead to miraculous behaviour from back-propagated obligations.

Definition 6.15 (**bw** predicate transformer)**.**

$$\mathbf{bw}\,(B,\theta) \quad \triangleq \quad \mathbf{bwR}\,(B) \wedge \mathbf{bwQ}\,(B,\theta)$$

Informally, the obligation **bw** (B,θ) encodes the maximum information that Low can infer about the initial state of B via its interaction with B. Moreover, if B terminates, then **bw** (B,θ) incorporates Low's inferences about the final state reached by B derived from the subsequent behaviour of the process, as encoded by θ.

bw is monotonic in its second argument with respect to the $\sqsubseteq$ ordering.

Lemma 6.16 (Monotonicity of **bw**)**.** $\theta_1 \sqsubseteq \theta_2$ implies **bw** $(B,\theta_1) \sqsubseteq$ **bw** (B,θ_2)

However, **bw** is not necessarily monotonic in its first argument with respect to $\sqsubseteq$, because **wmr** is not monotonic with respect to $\sqsubseteq$ (Example 6.9).

6.3.4 Forwards Propagation

An alternative to backwards propagation would be to propagate CAs and Low's inferences *forwards* through blocks. Forwards propagation is analogous to calculating the *strongest postcondition* that a program satisfies if started in any initial state satisfying a specified precondition.

Example 6.17. The counterpart to **wmr** for forwards propagation is:

$$\forall\, x \bullet \mathbf{D}\,(B!) \wedge \neg\, wait' \Rightarrow \exists\, \widetilde{x} \bullet B!$$

Applying this predicate transformer to ST (Example 6.8) yields:

$$\begin{array}{l} ok' \wedge \neg\, wait' \Rightarrow \\ \left(\begin{array}{ll} & \widetilde{ok'} \wedge \neg\, \widetilde{wait'} \\ \wedge & \left(\begin{array}{ll} & \mathsf{last}\,(tr') = (s,a) \wedge \mathsf{last}(\widetilde{tr'}) = (s,\widetilde{a}) \wedge a = \widetilde{a} \\ \vee & \mathsf{last}\,(tr') = (t,b) \wedge \mathsf{last}(\widetilde{tr'}) = (t,\widetilde{b}) \wedge b = \widetilde{b} \end{array} \right) \end{array} \right) \end{array}$$

Unlike **wmr** (ST), this relation distinguishes the cases where Low learns *a* or *b*, since the tr' variable retains information about how the non-determinism is resolved. $\Diamond$

As Example 6.17 shows, forwards propagation retains more detailed information about Low's inferences than does backwards propagation. However, forwards propagations are far more laborious to calculate. This result correlates with Dijkstra and Scholten's observation that weakest preconditions are simpler to calculate than strongest postconditions (Dijkstra and Scholten, 1990, §12).

It is pragmatic to adopt backwards propagation as the basis of the verification method presented in subsequent sections. However, there is no fundamental reason why this verification method could not instead be defined in the style of forwards propagation.

6.4 Laws of Backwards Propagation

We can apply **bw** to any lifted construct, be it a lifted *Circus* action, a CA, or a composite construction of lifted actions, lifted operators and CAs.

As should be expected, back-propagating an obligation θ_1 through a CA embedding θ_2 yields the least upper bound of θ_1 and θ_2.

Law 6.18 (bw CA). $\mathbf{bw}\,(\langle\, \theta_1 \,\rangle, \theta_2) = \theta_1 \wedge \theta_2$

Calculating the backwards propagation of obligations through lifted actions can be difficult, especially with actions with a complex semantics. In this section, we propose tactics for dividing this calculation into manageable parts. We also identify laws for back-propagating obligations through the basic *Circus* constructs.

6.4.1 Heuristics

Calculating $\mathbf{bw}\,(B, \theta)$ from first principles can be tedious. Lemma 6.19 shows this calculation can be simplified by breaking B into parts and applying **bw** to each part.

Lemma 6.19 (Over-approximating **bw**).

$$\mathbf{bw}\,(B,\theta) \quad \sqsubseteq \quad \mathbf{bw}\,(B \wedge C,\theta) \wedge \mathbf{bw}\,(B \wedge \neg\, C,\theta)$$

Lemma 6.19 may over-approximate $\mathbf{bw}\,(B,\theta)$, but it is often desirable to calculate $\mathbf{bw}\,(B,\theta)$ exactly. An exact calculation can be obtained by partitioning B according to Low's abilities to distinguish between the behaviours of B. Theorem 6.20 specifies the scheme of these calculations.

Theorem 6.20 (Splitting **bw**).

$$\mathbf{bw}\,(B,\theta) \quad = \quad \mathbf{bw}\,(B \wedge C,\theta) \wedge \mathbf{bw}\,(B \wedge \neg\, C,\theta)$$

provided the following conditions hold:

$$\mathbf{bwR}\,(B \wedge C) = \mathbf{true} \qquad \text{(SC1)}$$
$$\mathbf{bwR}\,(B \wedge \neg\, C) = \mathbf{true} \qquad \text{(SC2)}$$
$$B = B \wedge \mathcal{I}(\mathcal{L}) \qquad \text{(SC3)}$$
$$C \wedge \mathcal{I}(\mathcal{L}) = \mathbf{U}\,(C) \wedge \mathcal{I}(\mathcal{L}) \qquad \text{(SC4)}$$

Side-conditions 1 and 2 of Theorem 6.20 require that at least one behaviour of B satisfies C and at least one behaviour of B does not satisfy C. Side-condition 3 stipulates that B is constructed from **UC**-lifted actions; this condition is trivially satisfied if B is a block or composed from blocks using the lifted *Circus* operators. Side-condition 4 demands C holds for each behaviour ϕ if and only if C holds for all behaviours that are Low-indistinguishable from ϕ.

We specialise Theorem 6.20 in three ways by instantiating C.

Traces If the projections of the traces of two behaviours through Low's window $\mathcal{L}$ are different, then Low can distinguish between those behaviours. This insight motivates the following lemma.

Lemma 6.21 (**bw** traces). Let $\mathcal{W} \subseteq \mathcal{L}$ and $HT = (tr' - tr) \upharpoonright \mathcal{W} \in S$, where S denotes a set of traces drawn from events in $\mathcal{W}$. Then, provided side-conditions 1, 2 and 3 of Theorem 6.20 are upheld:

$$\textsf{\textbf{bw}}\,(B, \theta) \quad = \quad \textsf{\textbf{bw}}\,(B \wedge HT, \theta) \wedge \textsf{\textbf{bw}}\,(B \wedge \neg\, HT, \theta)$$

Refusals If the process deadlocks, it *may* refuse any of the events in Low's window. If Low attempts to engage in those events and the process refuses them, then Low will perceive that refusal. By observing these refusals, Low may be able to infer some information about the state of the process at deadlock and, consequently, its state prior to deadlocking.

Lemma 6.22 (**bw** refusals)**.** Let $\mathcal{W} \subseteq \mathcal{L}$ and $HR = \mathcal{W} \subseteq ref'$. Then, provided side-conditions 1,2 and 3 of Theorem 6.20 are upheld:

$$\textsf{\textbf{bw}}\,(B, \theta) \quad = \quad \textsf{\textbf{bw}}\,(B \wedge HR, \theta) \wedge \textsf{\textbf{bw}}\,(B \wedge \neg\, HR), \theta)$$

Moreover, if B is a lifted *Circus* action, we also have (by **CSP4**-healthiness):

$$\textsf{\textbf{bw}}\,(B, \theta) \quad = \quad \textsf{\textbf{bw}}\,\big(B \wedge (wait' \Rightarrow HR), \theta\big) \wedge \textsf{\textbf{bw}}\,\big(B \wedge (wait' \Rightarrow \neg\, HR), \theta\big)$$

Termination Since $\mathcal{I}(\mathcal{L})$ models Low as being able to distinguish terminating from non-terminating behaviours, we also have Lemma 6.23. Again, Low may exploit knowledge of (non-)termination to infer information about the initial state of a block.

Lemma 6.23 (**bw** termination)**.** Provided side-conditions 1,2 and 3 of Theorem 6.20 are upheld:

$$\textsf{\textbf{bw}}\,(B, \theta) \quad = \quad \textsf{\textbf{bw}}\,\big(B \wedge wait', \theta\big) \wedge \textsf{\textbf{bw}}\,\big(B \wedge \neg\, wait', \theta\big)$$

Naturally, these lemmas can be used together, to partition lifted constructs more finely than on traces, refusals or termination alone.

6.4.2 Propagating through *Circus* Constructs

bw inherits many of the properties of the **wp** predicate transformer. Following the style of the weakest precondition calculus, we now define laws for back-propagating obligations through lifted *Circus* constructs.

Since these laws are defined over individual lifted actions, they cannot be applied to more complex blocks. Nevertheless, by Condition 5.12, such a block can be broken apart (by refinement) into a composite construct, featuring simpler blocks that are amenable to these laws.

Primitives As should be expected, the lifted *Skip* is an identity for backwards propagation.

Law 6.24 (bw *Skip*). $\textsf{bw}\,(\langle\, Skip\,\rangle\,,\theta) = \theta$

Likewise, the lifted *Stop* and the lifted *Chaos* are zeros for backwards propagation.

Law 6.25 (bw *Stop*). $\textsf{bw}\,(\langle\, Stop\,\rangle\,,\theta) = \mathbf{true}$

Law 6.26 (bw *Chaos*). $\textsf{bw}\,(\langle\, Chaos\,\rangle\,,\theta) = \mathbf{true}$

Scoping Backwards propagation distributes through a declaration of variable a simply by hiding a and $\widetilde{a}$.

Law 6.27 (bw scope).

$$\textsf{bw}\,(\mathbf{var}\; a : T \bullet B, \theta) \quad = \quad \forall\, a : T \bullet \exists\, \widetilde{a} : T \bullet \textsf{bw}\,(B, \forall\, a : T \bullet \exists\, \widetilde{a} : T \bullet \theta)$$

Commands Law 6.28 provides a general account of back-propagating obligations through specification statements.

Law 6.28 (bw specification statement).

$$\textsf{bwR}\,(\langle\, w \;:\; [Pre, Post]\,\rangle) \;=\; \textsf{U}\,\big(Pre \Rightarrow \exists\, v' \bullet Post \wedge u' = u\big)$$

$$\textsf{bwQ}\,(\langle\, w \,:\, [Pre, Post]\,\rangle\,, \theta) \quad = \quad \left(\begin{array}{l} \forall\, v' \bullet (Pre \wedge Post \wedge u' = u) \Rightarrow \\ \quad \exists\, \widetilde{v}' \bullet (\widetilde{Pre} \Rightarrow \widetilde{Post} \wedge \widetilde{u}' = \widetilde{u}) \wedge \theta' \\ \wedge\ \widetilde{Pre} \Rightarrow Pre \end{array}\right)$$

where u denotes all state variables not listed in w.

Law 6.29 follows by specialising Law 6.28.

Law 6.29 (**bw** assumption).

$$\textsf{bw}\,(\langle\, \{\, C \,\}\,\rangle\,, \theta) \quad = \quad (C \Rightarrow \exists\, \widetilde{v}' \bullet (\widetilde{C} \Rightarrow \widetilde{v}' = \widetilde{v}) \wedge \theta'[v/v']) \wedge (\widetilde{C} \Rightarrow C)$$

As CAs are a generalisation of coercions, Law 6.30 is a direct consequence of Law 5.44 and Law 6.18.

Law 6.30 (**bw** coercion). $\textsf{bw}\,(\langle\, [\, C\,]\,\rangle\,, \theta) = \theta \wedge \textsf{U}\,(C)$

The law for assignment is typical for a weakest precondition calculus.

Law 6.31 (**bw** assignment). $\textsf{bw}\,(\langle\, a := E\,\rangle\,, \theta) = \theta[E, \widetilde{E}/a, \widetilde{a}]$

Sequence As lifted sequential composition is just relational composition, back-propagating an obligation through a sequence of blocks can be performed block by block.

Law 6.32 (**bw** sequence).

$$\begin{array}{rcl} \textsf{bwR}\left(B_1 \,\widehat{;}\, B_2\right) & = & \textsf{bwR}\left(B_1 \wedge \mathsf{Conf}'\left(\textsf{bwR}\,(B_2)\,[x', \widetilde{x}'/x, \widetilde{x}]\right)\right) \\ \textsf{bwQ}\left(B_1 \,\widehat{;}\, B_2, \theta\right) & = & \textsf{bwQ}\,(B_1, \textsf{bwQ}\,(B_2, \theta)) \end{array}$$

Guard Recall from Definition 5.25 that the lifted guard $g \,\&\, B$ reveals the truth of the condition g to Low. If g is **false**, this construct deadlocks and reveals no extra information to Low. If g is **true**, this construct behaves as B and reveals whatever information to Low that B reveals.

Law 6.33 (**bw** guard). $\textsf{bw}\,(g \,\&\, B, \theta) = (\textsf{bw}\,(B, \theta) \wedge \textsf{U}\,(g)) \vee \textsf{U}\,(\neg\, g)$

Prefixing A prefixing $c!E \to \mathit{Skip}$ which outputs the value of an expression E to the environment on channel c may reveal to Low information about the process state in terms of E, provided Low can observe c.

Law 6.34 (**bw** output prefix).

$$\mathbf{bw}\,(\langle\, L \,:\, c!E \to \mathit{Skip} \,\rangle\,, \theta) \quad = \quad \begin{cases} \theta \wedge E = \widetilde{E} & \text{if } c \in L \\ \theta & \text{otherwise} \end{cases}$$

A prefixing $c.e \to \mathit{Skip}$ which communicates a constant e to the environment does not change the state or reveal state information to Low. It is therefore an identity for backwards propagation, as Law 6.35 shows. This law can be derived from Law 6.34 by taking $E = e$.

Law 6.35 (**bw** prefix). $\mathbf{bw}\,(\langle\, c.e \to \mathit{Skip} \,\rangle\,, \theta) = \theta$

A prefixing which accepts an input value $e?$ from the environment on channel c reveals the exact value of $e?$ to Low, provided Low can observe c. Conversely, if Low cannot observe c, Low can still infer that $e?$ has the type $\delta(c)$ and that $P(e?)$ holds.

Law 6.36 (**bw** input prefix).

$$\begin{aligned} &\mathbf{bw}\,(\langle\, L \,:\, c?e : P \to e := e? \,\rangle\,, \theta) \\ &= \begin{cases} \forall\, e : \delta(c) \bullet P(e) \Rightarrow \theta[e/\widetilde{e}] & \text{if } c \in L \\ \forall\, e : \delta(c) \bullet P(e) \Rightarrow \exists\, \widetilde{e} : \delta(c) \bullet P(\widetilde{e}) \wedge \theta & \text{otherwise} \end{cases} \end{aligned}$$

Choice To back-propagate θ through a composite internal choice, it is sufficient to back-propagate θ through each construct of the choice separately and take the least upper bound.

Law 6.37 (**bw** internal choice).

$$\mathbf{bw}\,(B_1 \sqcap B_2, \theta) \quad \sqsubseteq \quad \mathbf{bw}\,(B_1, \theta) \wedge \mathbf{bw}\,(B_2, \theta)$$

It is harder to derive a general law for back-propagating θ through a composite external choice. Should B_1 and B_2 both deadlock without performing any event (or diverging), then $B_1 \Box B_2$ behaves as ${B_1}^{tt}_{ff} \wedge {B_2}^{tt}_{ff}$. This term may be miraculous if ${B_1}^{tt}_{ff}$ and ${B_2}^{tt}_{ff}$ restrict the state space in different ways — for instance, if they contain CAs — even if ${B_1}^{tt}_{ff}$ and ${B_2}^{tt}_{ff}$ are individually feasible. It follows that:

$$\mathbf{bw}\,(B_1 \Box B_2, \theta) \quad \sqsubseteq \quad \mathbf{bw}\,(B_1, \theta) \wedge \mathbf{bw}\,(B_2, \theta)$$

is *not* guaranteed in all cases. We side-step this problem by defining a backwards propagation law for a more specialised form of external choice. In this form, the choice is resolved by the environment before any state change can take place.

Law 6.38 (**bw** guarded prefixed external choice)**.** Provided at least one of the guards $g_1, g_2, \ldots, g_n$ is satisfied by each state:

$$\mathbf{bw}\,(\Box\, i \bullet g_i \,\&\, c_i \to B_i, \theta) \quad \sqsubseteq \quad \bigwedge i \bullet (\mathbf{U}\,(g_i) \wedge \mathbf{bw}\,(B_i, \theta)) \vee \mathbf{U}\,(\neg\, g_i)$$

Law 6.39 follows from Law 6.38 by taking **true** for each guard.

Law 6.39 (**bw** prefixed external choice)**.**

$$\mathbf{bw}\,(\Box\, i \bullet c_i \to B_i, \theta) \quad \sqsubseteq \quad \bigwedge i \bullet \mathbf{bw}\,(B_i, \theta)$$

6.5 Verifying Consistency

Owing to the non-compositional nature of confidentiality properties, a CA in a process specification may make other components of the process unimplementable, if the functionality of those components is inconsistent with the CA. Hence, it is expedient to ascertain that a process is feasible before proceeding with its development, to avoid unpleasant surprises if the process is indeed unimplementable.

This section describes a procedure for dividing the task of verifying that a process satisfies a CA into multiple cases, and then to conquer each case individually. This procedure is based on propagating the CA through the blocks of the process.

Propagation makes infeasible blocks readily identifiable, so this procedure enables the implementability (and therefore the security) of processes to be verified in a piecewise manner. Should the process be found to be unimplementable, then the requirements will need to be revised to resolve the conflict between functionality and confidentiality.

6.5.1 Propagated Confidentiality Annotations

Recall from Subsection 6.3.3 that backwards propagation can be applied to specification constructs of the form $\langle A \rangle ; \langle \theta \rangle$ to translate θ into an obligation over the initial state of $\langle A \rangle$. This back-propagated obligation can itself be embedded within a CA preceding $\langle A \rangle$:

$$\langle \mathsf{bw}\,(\langle A \rangle, \theta) \rangle ; \langle A \rangle ; \langle \theta \rangle$$

Inserting this CA into the specification refines the specification:

$$\begin{aligned} & \langle A \rangle ; \langle \theta \rangle \\ = \quad & \langle \mathbf{true} \rangle ; \langle A \rangle ; \langle \theta \rangle && \text{[property of CA]} \\ \sqsubseteq \quad & \langle \mathsf{bw}\,(\langle A \rangle, \theta) \rangle ; \langle A \rangle ; \langle \theta \rangle && \text{[Theorem 5.40]} \end{aligned}$$

Example 6.40 shows this refinement is non-trivial if $\langle A \rangle$ may deadlock or diverge.

Example 6.40. By Lemma 6.23, back-propagating θ through $\langle \mathit{Skip} \sqcap \mathit{Stop} \rangle$ just yields θ. Therefore:

$$\langle \mathit{Skip} \sqcap \mathit{Stop} \rangle ; \langle \theta \rangle \quad \sqsubseteq \quad \langle \theta \rangle ; \langle \mathit{Skip} \sqcap \mathit{Stop} \rangle ; \langle \theta \rangle$$

Provided θ is not vacuously **true**, the specification is strictly refined because $\langle \theta \rangle$ is enforced even if the non-deterministic choice resolves to $\langle \mathit{Stop} \rangle$. $\Diamond$

Suppose that, when $\langle A \rangle$ is started in a lifted state ψ, it terminates in a final state that violates θ. Then $\langle \mathbf{bw}(\langle A \rangle, \theta) \rangle$ is miraculous in ψ.

Example 6.41. Recall the specification from Example 6.11:

$$\langle\, s!a \rightarrow Skip \sqcap t!b \rightarrow Skip \,\rangle \,;\langle\, a = 0 \Rightarrow \widetilde{a} > 0 \,\rangle$$

We insert a CA encapsulating the back-propagation of $a = 0 \Rightarrow \widetilde{a} > 0$ through the block (see Example 6.11) into the specification, to obtain:

$$\langle\, a = \widetilde{a} \wedge b = \widetilde{b} \wedge a \neq 0 \,\rangle\,;\,\langle\, s!a \rightarrow Skip \sqcap t!b \rightarrow Skip \,\rangle\,;\langle\, a = 0 \Rightarrow \widetilde{a} > 0 \,\rangle$$

Now, the feasibility of the specification is made explicit: if it starts in a state where $a = 0$, then miraculous behaviour is assured. $\Diamond$

6.5.2 Left Justification

We now describe a procedure for transforming a lifted process design into a form where inconsistencies between the functionality and confidentiality attributes of a process can be detected by studying the CAs within the process alone. The procedure is based on inserting propagated CAs between every lifted specification construct within the process body. These CAs are inserted to record Low's inferences about the process state at each point in the process's execution, based on Low's observations of the subsequent behaviour of the process.

To formalise this procedure, we define a special format for lifted constructs, where each construct is surrounded by a pair of CAs.

Definition 6.42 (Left justification)**.** A lifted construct is *left-justified* if and only if it has the form:

$$\langle\, \theta^{\alpha} \,\rangle\,; B\,;\langle\, \theta^{\omega} \,\rangle$$

such that $\mathbf{bw}(B, \theta^{\omega}) \sqsubseteq \theta^{\alpha}$.

Of course, we could left-justify any construct by setting $\theta^{\alpha} = \mathbf{false}$. That would not be helpful for our broader goal of verifying the process is consistent with its CAs. Rather, we are interested in identifying the *weakest* obligation θ^{α} that left-justifies the construct.

Every CA is left-justified in its own right:

$$\langle\, \theta \,\rangle \quad = \quad \langle\, \mathbf{true} \,\rangle \,;\, \langle\, \theta \,\rangle \,;\, \langle\, \mathbf{true} \,\rangle \quad = \quad \langle\, \theta \,\rangle \,;\, \langle\, \mathbf{true} \,\rangle \,;\, \langle\, \mathbf{true} \,\rangle$$

It is straightforward to left-justify primitive constructs — such as specification statements and guards — with the laws of **bw** (Section 6.4).

To back-propagate a CA through a composite lifted construct, we left-justify each of its components individually, in order to derive a CA to left-justify the composite construct as a whole. We now consider the details of this procedure for specific composite constructs.

Sequence We propagate a CA through a sequence of blocks in a pattern reminiscent of the right fold operator in functional programming (Hutton, 1999). For instance, given:

$$\langle\, A_1 \,\rangle \;;\, \ldots \,;\, \langle\, A_n \,\rangle \;;\, \langle\, \theta_n \,\rangle$$

we left-justify each lifted action as follows:

$$\langle\, \theta_0 \,\rangle \,;\, \langle\, A_1 \,\rangle \;;\, \langle\, \theta_1 \,\rangle \,;\, \ldots \,;\, \langle\, \theta_{n-1} \,\rangle \,;\, \langle\, A_n \,\rangle \;;\, \langle\, \theta_n \,\rangle$$

where, for each $0 \le i < n$, $\mathbf{bw}\,(\langle\, A_{i+1} \,\rangle\,, \theta_{i+1}) \sqsubseteq \theta_i$ holds.

Remark 6.43. It is most efficient to left-justify sequences of blocks from right to left. Consider:

$$\langle\, A_1 \,\rangle \;;\, \langle\, \theta_1 \,\rangle \,;\, \langle\, A_2 \,\rangle \;;\, \langle\, \theta_2 \,\rangle$$

If $\mathbf{bw}\,(\langle\, A_2 \,\rangle\,, \theta_2) \not\sqsubseteq \theta_1$, then it is efficient to left-justify $\langle\, A_2 \,\rangle$ first, so that $\langle\, A_1 \,\rangle$ can be left-justified with respect to $\theta_1 \wedge \mathbf{bw}\,(\langle\, A_2 \,\rangle\,, \theta_2)$.

Choice When propagation reaches a composite choice construct, we distribute the CA through the choice as follows:

$$\begin{array}{lcl}(B_1 \sqcap B_2) \mathbin{;} \langle\, \theta \,\rangle & = & B_1 \mathbin{;} \langle\, \theta \,\rangle \sqcap B_2 \mathbin{;} \langle\, \theta \,\rangle \\ (B_1 \,\Box\, B_2) \mathbin{;} \langle\, \theta \,\rangle & = & B_1 \mathbin{;} \langle\, \theta \,\rangle \,\Box\, B_2 \mathbin{;} \langle\, \theta \,\rangle\end{array}$$

We left-justify B_1 and B_2 (with respect to θ) separately. Following Law 6.37 and Law 6.38, we then left-justify the choice construct as a whole with a CA containing the least upper bound of the obligations from each branch:

$$\begin{array}{lcl}(B_1 \sqcap B_2) \mathbin{;} \langle\, \theta \,\rangle & \sqsubseteq & \langle\, \mathbf{bw}\,(B_1,\theta) \wedge \mathbf{bw}\,(B_2,\theta) \,\rangle \mathbin{;} (B_1 \sqcap B_2) \mathbin{;} \langle\, \theta \,\rangle \\ (B_1 \,\Box\, B_2) \mathbin{;} \langle\, \theta \,\rangle & \sqsubseteq & \langle\, \mathbf{bw}\,(B_1,\theta) \wedge \mathbf{bw}\,(B_2,\theta) \,\rangle \mathbin{;} (B_1 \,\Box\, B_2) \mathbin{;} \langle\, \theta \,\rangle\end{array}$$

Parallel Parallel-by-merge is defined so that, upon termination, the final values of the state variables are selected from the final states reached by the concurrent actions (Hoare and He, 1998; Oliveira et al., 2009). Therefore, a CA which succeeds a parallel construct can be split into parts, depending on the merge specified by the parallel operator.

Let ns_1 and ns_2 denote disjoint subsets (but not necessarily a partition) of the state variables of a process. Then, let θ_0, θ_1 and θ_2 denote obligations that satisfy the conditions:

$$\begin{array}{l}\alpha\theta_0 \cap (ns_1 \cup \widetilde{ns_1} \cup ns_2 \cup \widetilde{ns_2}) = \varnothing \\ \alpha\theta_1 \subseteq ns_1 \cup \widetilde{ns_1} \\ \alpha\theta_2 \subseteq ns_2 \cup \widetilde{ns_2}\end{array}$$

These conditions demand the alphabets of θ_0, θ_1 and θ_2 are disjoint. From this point onwards, we consider parallel constructs of the form:

$$(B_1 \,|[\, ns_1 \mid cs \mid ns_2 \,]|\, B_2) \mathbin{;} \langle\, \theta_0 \wedge \theta_1 \wedge \theta_2 \,\rangle$$

Remark 6.44. Some obligations cannot be structured in this form. An example obligation would be $\widetilde{g} \neq \widetilde{h}$, where $ns_1 = \{g\}$ and $ns_2 = \{h\}$.

However, this obligation can be refined to $\widetilde{g} \geq 0 \wedge \widetilde{h} < 0$ to achieve an obligation structured in our required form.

Since a parallel composition terminates only if both its components terminate, we can distribute θ_1 and θ_2 according to the rule:

$$\begin{gathered}(B_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_2) \,;\langle\, \theta_0 \wedge \theta_1 \wedge \theta_2 \,\rangle \\ \sqsubseteq \\ (B_1 \,;\langle\, \theta_1 \,\rangle \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_2 \,;\langle\, \theta_2 \,\rangle) \,;\langle\, \theta_0 \wedge \theta_1 \wedge \theta_2 \,\rangle\end{gathered}$$

Since B_1 and B_2 must synchronise on events in cs, each of these components may constrain the other's space of behaviours. In some cases, the synchronisation may prevent B_1 or B_2 from engaging in cover story behaviours mandated by their respective CAs.

Example 6.45. Consider the specification, where $c_1, c_2 \notin L$:

$$\left(\begin{array}{c}\langle\, c_1 \to h := 0 \,\Box\, c_2 \to h := 1 \,\rangle \\ [\![\, \{h\} \mid \{\!|\, c_1, c_2 \,|\!\} \mid \varnothing \,]\!] \\ \langle\, c_1 \to Skip \,\rangle\end{array}\right) ;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle$$

Since the parallel branches can only synchronise on c_1, this specification can be rewritten as:

$$\begin{array}{lll}\langle\, c_1 \to h := 0 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & & \text{[semantics of parallel]} \\ = & \langle\, c_1 \to Skip \,\rangle \,;\, \langle\, h := 0 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle & \text{[Lemma 5.18]} \\ = & \langle\, c_1 \to Skip \,\rangle \,;\, \langle\, h := 0 \,\rangle \,;\langle\, \mathbf{false} \,\rangle & \text{[by Example 5.49]}\end{array}$$

We would fail to detect this insecurity in the original specification if we were to analyse the two parallel branches without accounting for the synchronisation between them. ◊

We can sidestep the problem highlighted by Example 6.45 by enlarging Low's window on B_1 and B_2 to include all channels listed by cs. In effect, we assume *pessimistically* that Low can observe all synchronisation events performed by B_1 and B_2 individually, even if they are hidden from the environment.

Example 6.46. Continuing from Example 6.45, interpreting the left branch of the parallel construct under the assumption that Low *can* observe synchronisation events over c_1 and c_2 gives:

$$\begin{array}{ll}
& \langle\, c_1 \to h := 0 \,\Box\, c_2 \to h := 1 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle \\
\sqsubseteq & (\langle\, c_1 \to h := 0 \,\rangle \,\Box\, \langle\, c_2 \to h := 1 \,\rangle) \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle \\
& \hfill \text{[Condition 5.12]} \\
= & \left(\begin{array}{ll} & \langle\, c_1 \to h := 0 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle \\ \Box & \langle\, c_2 \to h := 1 \,\rangle \,;\langle\, h = 0 \Rightarrow \widetilde{h} > 0 \,\rangle \end{array} \right) \quad \text{[distributivity]} \\
= & \langle\, c_1 \to h := 0 \,\rangle \,;\langle\, \textbf{false} \,\rangle \,\Box\, \langle\, c_2 \to h := 1 \,\rangle \,;\langle\, \textbf{true} \,\rangle \\
& \hfill \text{[as Example 5.48, Example 5.49]}
\end{array}$$

Now, the inconsistency hidden in the original specification is exposed. ◊

Remark 6.47. Assuming that Low can observe synchronisations between parallel processes is incompatible with the superposition technique described in Subsection 5.6.3. As superposition is used to impose confidentiality properties over the synchronisation events, it would be self-defeating to assume Low can observe those events. This problem could be solved by reforming a superposed process into a single process (with CAs embedded alongside blocks) before applying back-propagation.

We then left-justify $B_1 \,;\langle\, \theta_1 \,\rangle$ and $B_2 \,;\langle\, \theta_2 \,\rangle$ separately:

$$\begin{array}{lll}
B_1 \,;\langle\, \theta_1 \,\rangle & \sqsubseteq & \langle\, \mathsf{bw}\left(B_1^+, \theta_1\right) \,\rangle \,; B_1 \,;\langle\, \theta_1 \,\rangle \\
B_2 \,;\langle\, \theta_2 \,\rangle & \sqsubseteq & \langle\, \mathsf{bw}\left(B_2^+, \theta_2\right) \,\rangle \,; B_2 \,;\langle\, \theta_2 \,\rangle
\end{array}$$

where B_1^+ and B_2^+ denote B_1 and B_2 respectively, but with the events in cs modelled as visible to Low.

Finally, the parallel composition construct can be left-justified by taking the least upper bound of the left-justified CAs:

$$\begin{array}{c}\langle\, \theta_0 \wedge \textbf{bw}\left(B_1^+, \theta_1\right) \wedge \textbf{bw}\left(B_2^+, \theta_2\right) \rangle\, ; \\ (\langle\, \textbf{bw}\left(B_1^+, \theta_1\right) \rangle\, ; B_1 \, ; \langle\, \theta_1 \,\rangle \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; \langle\, \textbf{bw}\left(B_2^+, \theta_2\right) \rangle\, ; B_2 \, ; \langle\, \theta_2 \,\rangle) \, ; \\ \langle\, \theta_0 \wedge \theta_1 \wedge \theta_2 \,\rangle \end{array}$$

Since θ_0 references state variables that are left unchanged by the parallel construct, it is important that θ_0 is included within the left-justifying CA.

Recursion In *Circus*, a recursive construct can take the form $\mu X \bullet F(X)$, or other forms that can be transformed to this form (Oliveira et al., 2009). In order to left-justify the lifted loop body $\langle\, F(Skip) \,\rangle$ with respect to θ, our strategy is to identify an *invariant obligation* θ_I (where $\theta \sqsubseteq \theta_I$) that is maintained by $\langle\, F(Skip) \,\rangle$.

Definition 6.48 (Invariant obligation)**.** The obligation θ_I is an invariant obligation for $\langle\, F(Skip) \,\rangle$ if back-propagating θ_I through $\langle\, F(Skip) \,\rangle$ yields an obligation no stronger than θ_I; that is:

$$\textbf{bw}\left(\langle\, F(Skip) \,\rangle, \theta_I\right) \quad \sqsubseteq \quad \theta_I$$

We can take the obligation θ_0 from the CA following the recursive construct and calculate:

$$\theta_{i+1} \quad = \quad \theta_i \wedge \textbf{bw}\left(\langle\, F(Skip) \,\rangle, \theta_i\right)$$

until we identify an θ_n such that $\theta_n = \theta_{n+1}$, whereupon (by Definition 6.48) we know that θ_n is an invariant for the loop body.

Since **bw** is not monotonic in its first argument, the conjunction of θ_i in θ_{i+1} ensures that θ_{i+1} is no weaker than θ_i. However, this method is not guaranteed to identify a finite n such that $\theta_n = \theta_{n+1}$ in the general case. In such cases, we may resort to intuition in order to identify an invariant obligation, by looking for patterns in the sequence of obligations $\theta_0, \theta_1, \theta_2, \ldots$. Nor is θ_n necessarily the weakest (and hence most desirable) invariant obligation for the loop body. We leave the problem of devising more sophisticated techniques for identifying invariant obligations for future work.

6.5.3 Full Justification

We say that a process is *fully justified* if every lifted construct in its body is left-justified. Full justification entails that each component within a composition is left-justified and, moreover, that each composite construct is also left-justified in its own right.

The purpose of fully justifying a process specification is to reveal miraculous behaviour stemming from inconsistencies between functionality and confidentiality. These inconsistencies are indicated by the presence of the miraculous CA $\langle\, \mathbf{false} \,\rangle$ within a fully justified process. This CA is generated by backwards propagation if execution of the constructs that follow it could leak information to Low in violation of a subsequent CA.

Example 6.49. Assuming Low can observe events on channel s, left-justifying the specification fragment:

$$ST1 \quad = \quad \langle\, s!a \rightarrow Skip \sqcap t!b \rightarrow Skip \,\rangle \; ; \langle\, a \neq \widetilde{a} \,\rangle$$

yields the infeasible specification $\langle\, \mathbf{false} \,\rangle \, ; ST1$. ◇

Fully justifying a process is a sound method for verifying its feasibility, because left-justification is a sound method for verifying the feasibility of individual constructs. Hence, the absence of the CA $\langle\, \mathbf{false} \,\rangle$ in a fully justified process body implies the process is feasible. However, full justification is only complete as a verification method if no approximations are made: if Low's inferences are over-approximated too conservatively, then fully justifying a process could generate spurious inconsistencies.

6.5.4 Resolving Inconsistency

Each back-propagated CA reflects Low's inferences about the behaviour of the constructs executed after that CA is invoked. Therefore, whenever backwards propagation generates the CA $\langle\, \mathbf{false}\, \rangle$, we should focus our attention on the constructs following that CA, in order to understand the conflict between functionality and confidentiality.

The presence of miraculous CAs within a process does not necessarily imply the process development is doomed. For instance, if a miraculous CA is local to a non-deterministic construct, then it can be eliminated by refinement, as Example 6.50 shows.

Example 6.50. A specification fragment which refines $ST1$ is:

$$ST2 \quad = \quad (\langle\, s!a \to Skip\, \rangle \sqcap \langle\, t!b \to Skip\, \rangle)\ ; \langle\, a \neq \widetilde{a}\, \rangle$$

Left-justifying each block of $ST2$ separately gives:

$$\begin{gathered}\langle\, \mathbf{false}\, \rangle ; \langle\, s!a \to Skip\, \rangle\ ; \langle\, a \neq \widetilde{a}\, \rangle \\ \sqcap \\ \langle\, a \neq \widetilde{a} \wedge b = \widetilde{b}\, \rangle ; \langle\, t!b \to Skip\, \rangle\ ; \langle\, a \neq \widetilde{a}\, \rangle\end{gathered}$$

Since $\langle\, \mathbf{false}\, \rangle$ is the lattice top, this specification is equivalent to:

$$\langle\, a \neq \widetilde{a} \wedge b = \widetilde{b}\, \rangle ; \langle\, t!b \to Skip\, \rangle\ ; \langle\, a \neq \widetilde{a}\, \rangle$$

which is itself a fully-justified specification. ◊

In practice, it is expedient to verify the feasibility of back-propagated CAs when calculating them. If a miraculous CA is detected, then further backwards propagation is futile if the inconsistency it reveals cannot be resolved by refining away insecure components of the process.

6.5.5 Discussion

Our propagation procedure is related to the *unwinding* technique first proposed by Goguen and Meseguer (1984), which aims to simplify the task of verifying that a system satisfies noninterference. Modern formulations of unwinding transform a global confidentiality property (expressed in terms of trace sets) over a system into a set of conditions over its individual state transitions (Ryan and Schneider, 1999; Mantel, 2000a). These conditions can then be discharged using standard proof methods.

Propagation, in a sense, is a dual of unwinding. Propagation transforms a CA — a confidentiality property located in one part of a process but applying to the whole process — into a collection of CAs distributed throughout the process. The procedure for distributing these CAs entails a proof of the process's feasibility, or a result that suggests (but does not necessarily imply) the process is inconsistent.

It is useful for propagated CAs to be retained in a process design, to save the effort of re-calculating them after performing each refinement step. As we show in the next section, they provide useful guidance for ensuring that refinement steps do not violate CAs.

6.6 A Secure Refinement Strategy

For stepwise refinement to be useful, it must lead to process designs that are implementable. Therefore, once a process has been verified to be consistent with its CAs, it is highly desirable that each refinement step maintains the consistency of the process. However, we established in Section 6.2 that applying the *Circus* refinement laws to lifted processes containing CAs is not guaranteed to preserve their feasibility.

This section extends the *Circus* refinement strategy proposed by Cavalcanti et al. (2003) to accommodate process designs featuring CAs. Given a consistent process design, we justify each refinement of the process design with respect to the CAs embedded within the process, in order to maintain the process's consistency.

The secure refinement strategy set out in this section addresses action refinement, data refinement and process refinement. It is closely related to the *Circus* refinement strategy, but it describes additional checks that need to be made to ensure the consistency of processes is preserved by refinement steps.

The application of full justification to a process design greatly reduces the effort needed to justify that refinement steps maintain the consistency of the process. Using the propagated CAs within a process, we can verify the consistency of the refined process without needing to re-verify the entire process at each refinement step.

6.6.1 Action Refinement

Given a fully justified process Q, we are free to refine lifted *Circus* actions nested within blocks. Consider the following left-justified fragment of Q:

$$\langle\, \mathbf{bw}\,(\langle\, A_1 \,\rangle\,,\theta)\,\rangle\,;\,\langle\, A_1 \,\rangle\,;\langle\, \theta \,\rangle$$

Let $A_1 \sqsubseteq A_2$. If $\mathbf{bw}\,(\langle\, A_2 \,\rangle\,,\theta) \not\sqsubseteq \mathbf{bw}\,(\langle\, A_1 \,\rangle\,,\theta)$, then replacing $\langle\, A_1 \,\rangle$ with $\langle\, A_2 \,\rangle$ means the specification fragment is no longer left-justified. Hence, the resulting process is not fully justified.

To verify the refinement is feasibility-preserving, we return the refined process to its fully justified form, so that insecurities can be identified. First, we re-apply left-justification to the block to obtain:

$$\langle\, \mathbf{bw}\,(\langle\, A_2 \,\rangle\,,\theta)\,\rangle\,;\,\langle\, A_2 \,\rangle\,;\langle\, \theta \,\rangle$$

Then, we back-propagate $\langle\, \mathbf{bw}\,(\langle\, A_2 \,\rangle\,,\theta)\,\rangle$ to the preceding blocks and apply left-justification until a fully justified process is reached.

As before, if propagation yields the CA $\langle\, \mathbf{false} \,\rangle$, then it indicates the refinement step is not feasibility-preserving. In that case, it is necessary either to rectify that infeasibility by refining the process further, or to discard the refined process and try a different development step instead.

In other cases, where $\mathbf{bw}\,(\langle\, A_2 \,\rangle\,,\theta) \sqsubseteq \mathbf{bw}\,(\langle\, A_1 \,\rangle\,,\theta)$, replacing $\langle\, A_1 \,\rangle$ with $\langle\, A_2 \,\rangle$ will maintain a fully justified process. There is no need to back-propagate $\mathbf{bw}\,(\langle\, A_2 \,\rangle\,,\theta)$, because $\mathbf{bw}$ is monotonic in its second argument (Lemma 6.16). Therefore, no additional verification is necessary. This observation facilitates a useful shortcut, as Example 6.51 shows.

Example 6.51. The left-justified form of the specification fragment from Example 6.41 can be refined by resolving the non-deterministic choice between the s and t actions:

$$\langle\, a = \widetilde{a} \wedge b = \widetilde{b} \wedge a \neq 0 \,\rangle \,;\, \langle\, s!a \to Skip \,\rangle \,;\, \langle\, a = 0 \Rightarrow \widetilde{a} > 0 \,\rangle$$

This block remains left-justified, because:

$$\mathbf{bw}\,(s!a \to Skip, a = 0 \Rightarrow \widetilde{a} > 0) \quad \sqsubseteq \quad a = \widetilde{a} \wedge b = \widetilde{b} \wedge a \neq 0$$

and so no further backwards propagation is necessary. ◊

6.6.2 Data Refinement

Data refinement is conventionally conducted by identifying a retrieve relation linking a concrete state space to an abstract state space (He et al., 1986). To ensure that a concrete process simulates the abstract process correctly, this relation must satisfy particular soundness conditions. Cavalcanti et al. (2003) and Oliveira (2005) define the soundness conditions that a retrieve relation must satisfy to be a *forwards simulation* over *Circus* actions.

Definition 6.52 (Forwards simulation (Oliveira, 2005)). A forwards simulation between actions A_1 and A_2 of processes P_1 and P_2, with state spaces v_1 and v_2 respectively and local state L, is a relation R between v_1, v_2 and L satisfying:

1. (Feasibility) $\forall\, v_2, L \bullet (\exists\, v_1 \bullet R)$
2. (Correctness) $\forall\, v_1, v_2, v_2', L \bullet R \wedge A_2 \Rightarrow (\exists\, v_1', L \bullet A_1 \wedge R')$

A lifted forwards simulation between lifted actions has essentially the same structure as a forward simulation between *Circus* actions.

Definition 6.53 (Lifted forwards simulation). A lifted forwards simulation between lifted constructs B_1 and B_2 of processes P_1 and P_2, with state spaces $v_1, \widetilde{v_1}$ and $v_2, \widetilde{v_2}$ respectively and local state $L, \widetilde{L}$, is a relation R between $v_1, \widetilde{v_1}$, $v_2, \widetilde{v_2}$ and $L, \widetilde{L}$ satisfying:

1. (Feasibility) $\forall v_2, \widetilde{v_2}, L, \widetilde{L} \bullet (\exists v_1, \widetilde{v_1} \bullet R)$
2. (Correctness) $\begin{array}{l} \forall v_1, \widetilde{v_1}, v_2, \widetilde{v_2}, v_2', \widetilde{v_2'}, L, \widetilde{L} \bullet \\ \quad R \wedge B_2 \quad \Rightarrow \quad (\exists v_1', \widetilde{v_1'}, L, \widetilde{L} \bullet B_1 \wedge R') \end{array}$

Corollary 6.54 is a consequence of the order-embedding between the space of *Circus* actions and the space of lifted actions (Lemma 5.3).

Corollary 6.54. The relation $\mathbf{U}(R)$ is a lifted forwards simulation between $\langle A_1 \rangle$ and $\langle A_2 \rangle$ exactly if R is a forwards simulation between A_1 and A_2.

With Corollary 6.54, we can apply the existing simulation laws given by Oliveira et al. (2009) for data-refining lifted *Circus* actions. Of course, if these data refinements entail the reduction of non-determinism, they need not yield a fully justified process.

When data refinement is applied to a abstract process featuring CAs, those CAs need to be translated into CAs over the concrete state space. Given an abstract obligation θ_1, we derive a concrete obligation θ_2 by projecting θ_1 through the simulation relation R:

$$\theta_2 \quad = \quad \forall v_1, \widetilde{v_1} \bullet \mathbf{U}(R) \Rightarrow \theta_1$$

θ_2 is the weakest obligation such that, for each state ψ_2 and fog state $\widetilde{\psi_2}$ that satisfy θ_2, *all* projections of $\psi_2 \wedge \widetilde{\psi_2}$ through $\mathbf{U}(R)$ satisfy θ_1.

Example 6.55. Recall the process fragment $B01$ from Example 6.1:

$$B01 \quad \triangleq \quad \langle h := 0 \sqcap h := 1 \rangle \,;\langle h = 0 \Rightarrow \widetilde{h} > 0 \rangle$$

If we data-refine each component of this fragment, with the simulation $(g = 0 \lhd h > 0 \rhd g = 1)$, then we obtain the concrete process fragment:

$$\langle g := 0 \sqcap g := 1 \rangle \,;\langle g = 0 \Rightarrow \widetilde{g} = 1 \rangle$$

which is feasible. $\Diamond$

Inconsistencies may arise if data refinement is applied to CAs in isolation from the functionality surrounding them, as Example 6.56 shows.

Example 6.56. Taking $B01$ as before, let $R = (g = 0 \lhd h < 10 \rhd g = 1)$. Projecting the CA of $B01$ through R yields:

$$\begin{array}{ll} & \forall h, \widetilde{h} \bullet \mathbf{U}\,(g = 0 \lhd h < 10 \rhd g = 1) \Rightarrow (h = 0 \Rightarrow \widetilde{h} > 0) \\ = & \forall h, \widetilde{h} \bullet (\mathbf{U}\,(g = 0 \lhd h < 10 \rhd g = 1) \wedge h = 0) \Rightarrow \widetilde{h} > 0 \\ = & \forall \widetilde{h} \bullet (g = 0 \wedge (\widetilde{g} = 0 \lhd \widetilde{h} < 10 \rhd \widetilde{g} = 1)) \Rightarrow \widetilde{h} > 0 \\ = & g = 0 \Rightarrow \widetilde{g} = 1 \end{array}$$

R is a forwards simulation between $h := 0 \sqcap h := 1$ and the concrete action $g := 0$. This means the data refinement of $B01$ with respect to R is:

$$\langle\, g := 0 \,\rangle \;;\langle\, g = 0 \Rightarrow \widetilde{g} = 1 \,\rangle \quad = \quad \langle\, g := 0 \,\rangle \;;\langle\, \mathbf{false} \,\rangle$$

This refinement is clearly too strong. One solution is to restructure the CA before applying data refinement:

$$B01 \quad = \quad \langle\, h := 0 \sqcap h := 1 \,\rangle \;;\langle\, h = 0 \Rightarrow \widetilde{h} = 1 \,\rangle$$

Data-refining this restructured specification with respect to R yields:

$$\langle\, g := 0 \,\rangle \;;\langle\, g = 0 \Rightarrow (\widetilde{g} = 0 \vee \widetilde{g} = 1)\,\rangle \quad = \quad \langle\, g := 0 \,\rangle$$

Now, the CA is innocuous: the data refinement has resolved the nondeterministic choice in a secure way, so the CA plays no further part. $\Diamond$

As Example 6.55 demonstrates, applying data refinement to processes featuring CAs has the potential to create inconsistencies between functionality and confidentiality. These inconsistencies can be detected by restoring a data-refined process to a fully justified form, just as with action refinement.

6.6.3 Process Refinement

In *Circus*, process refinement is defined as action refinement over the main actions of processes, with their state variables hidden:

$$P_1 \sqsubseteq_{\mathcal{P}} P_2 \quad \triangleq \quad (\exists\, v, v' \bullet P_1.Act) \sqsubseteq (\exists\, v, v' \bullet P_2.Act)$$

where $P_1.Act$ and $P_2.Act$ denote the main actions of P_1 and P_2 respectively. This definition can easily be extended to the lifted space, by hiding the fog state variables as well.

The *Circus* refinement strategy defines *splitting laws* for achieving process refinement (Cavalcanti et al., 2003). These laws enable a process P to be decomposed into multiple processes. These sub-processes can be joined together (using parallel composition, or another operator) to form a process that refines P. Hence, a monolithic process specification can be implemented by a coalition of communicating processes.

The splitting laws for *Circus* processes are equally applicable to lifted processes. Since these laws are defined in terms of action refinement, we can apply the *Circus* refinement calculus (extended with the laws of Section 6.4) to split a lifted process into multiple lifted processes. However, lifted processes need special attention for a couple of reasons:

- If processes are joined together in parallel composition on channel set cs, then synchronisations between the processes may constrain their behaviours. As described in Subsection 6.5.2, this effect can lead to inconsistencies in the parallel process which are not detectable by analysing its constituent processes individually. This problem can be avoided by enlarging Low's window on each process to include all channels in cs.

- An impediment to process refinement may arise if CAs reference state variables spanning across a partition of the process state. In such cases, it may be necessary to strengthen a CA with an equivalent (or stronger) CA before decomposing the process.

Example 6.57. Suppose a process incorporates a CA that references state variables a and b, such as $\langle \widetilde{a} \neq a \lor \widetilde{b} \neq b \rangle$. This CA could be refined to either $\langle \widetilde{a} \neq a \rangle$ or $\langle \widetilde{b} \neq b \rangle$, in order to partition the process into two components with states a and b respectively. $\Diamond$

We do not consider the *Circus* process indexing mechanism, or the associated promotion techniques described by Cavalcanti et al. (2003).

6.7 Implementation

A program specification is ready to be implemented only if it exclusively consists of executable constructs (Morgan, 1994). Since CAs are partial constructs — i.e. they may exhibit miraculous behaviour — they are not executable and should therefore be eliminated from a process design to ensure the process is implementable.

6.7.1 Translating to *Circus*

The order embedding linking *Circus* constructs and lifted constructs can be applied to translate lifted processes to *Circus* processes. The purpose of making this translation is to enable existing techniques for implementing *Circus* to be applied.

Theorem 6.58. Provided Q is a fully justified lifted process with no miraculous CAs, $\mathbf{D}(Q)$ can be expressed as a pure *Circus* process.

The justification of Theorem 6.58 is instructive. Given a fully justified process Q such that no CA in Q is $\langle \textbf{false} \rangle$, then because Q is fully justified, the CAs in Q cannot induce miraculous behaviour. By Condition 5.11, the application of $\mathbf{D}$ to Q distributes through each the lifted operators of Q. Hence, by distributing $\mathbf{D}$ throughout the body of Q, $\mathbf{D}(Q)$ can be expressed as a process consisting of constructs of the form:

$$\mathbf{D}(\langle A \rangle) \quad \text{and} \quad \mathbf{D}(\langle \theta \rangle)$$

composed with the (unlifted) *Circus* operators. By Law 5.4 and Law 5.45, each of these constructs is itself a *Circus* action. It is important to stress this translation does *not* constitute a refinement of a lifted process, because the fog variables are absent from the alphabet of *Circus* processes.

A *Circus* process translated by applying **D** to a fully justified lifted process by this procedure does not leak secret information to Low. However, the *Circus* semantics does not guarantee that refinements of this *Circus* process are themselves secure. To guard against the danger of making the process insecure, the translation to *Circus* should be made only after all development work on the process design is finished. In particular, if the process is fully deterministic, then no refinement can reduce non-determinism further, thereby disposing of the danger described in Section 4.5).

6.7.2 Compilation to Code

Oliveira (2005) has described a scheme for translating *Circus* processes into Java code. Since the processes derived by applying **D** to lifted processes are just *Circus* processes, this scheme could be applied to implement them. However, the security of an implementation generated using this scheme is open to question, because it could leak secrets to the adversary in unexpected ways:

- At the software level, Oliveira's translation scheme implements the resolution of non-determinism according to the output of a software random number generator (RNG). It is well known that the output stream of a software RNG is not truly random, but exhibits patterns (Anderson, 2001). If an adversary can detect those patterns, it may exploit them to predict how non-determinism is resolved by the process implementation, with catastrophic consequences for confidentiality.

- A confederation of processes may be distributed across a network of nodes. Encrypting the information transmitted between nodes would be necessary, but may not suffice to safeguard the confidentiality of that information. For instance, the encryption algorithm may be breakable with sufficient computational resources. Alternatively, the protocols used for authentication or key exchange may harbour subtle flaws in their design or implementation (Burrows et al., 1989; Carlsen, 1994), which an adversary may exploit to cause the protocol to deviate from the execution its designers intended.

- At the hardware level, power is consumed and electromagnetic radiation is generated in proportion to the computation being performed. Fluctuations in these *side channels* signal information about the computation in progress. If an adversary is able to monitor (or even tamper with) system hardware during operation, then it may exploit these side channels to extract information about the system's state or behaviour. These attacks are not merely hypothetical: Kocher et al. (1999) applied a form of power analysis to read cryptographic keys from smart cards, based on knowledge of the cryptographic algorithm being executed.

These characteristics of physical systems are not modelled in the semantics of *Circus* processes. In the words of Ryan and Schneider (1998):

> *"[A] formal model is necessarily an abstraction of the real system. As such it is always possible that some significant aspect of the real system is missed."*

Since we have not formalised these characteristics, we can only justify a physical computing artefact that implements a process design as being secure if we assume the adversary cannot exploit these characteristics. In principle, these characteristics could be modelled by extending *Circus* (and its semantics), but that endeavour is beyond the scope of this thesis.

6.8 Conclusion

This chapter has detailed a confidentiality-sensitive extension of the *Circus* refinement strategy for developing abstract process designs into concrete processes. Since refinement in the lifted semantics is confidentiality-preserving but not feasibility-preserving, this strategy emphasises providing guidance to software engineers to maintain the feasibility of each process design in the refinement chain.

Backwards propagation is an effective device for executing the strategy, as it provides a platform for verifying the feasibility of a process and conducting refinement steps that preserve feasibility. Fully justifying a process is sufficient for verifying that the process is feasible. However, the rules given in Subsection 6.5.2 for propagating CAs through lifted constructs are not guaranteed to be complete, because they over-approximate Low's inferences in some circumstances. Therefore, in those cases where full justification fails to prove that a process is feasible, software engineers may be obliged to resort to first principles in order to establish the process is indeed feasible, or to adjust the process design in order to make full justification successful.

Strictly speaking, verification can be carried out at any stage during a process's development. It is arguably best left to software engineers to choose the stage of development where verification is most appropriate. However, delaying verification may be disadvantageous, because without the guidance of propagated CAs, a process design may be reached containing infeasible elements that cannot be refined away. It would then be necessary either:

- to backtrack to an earlier design in the development and proceed along a different refinement path;
- to break the refinement chain, by weakening the functionality or confidentiality attributes of the design.
- to weaken the notion of refinement; for instance, to co-operating or independent refinement (Subsection 3.6.2).

The first two options are not ideal: the first incurs wasted effort, the second may sacrifice the design's correctness. The third option has promise if the observational abilities of all users of the process — and the relationships between those users — are known. However, *Circus* does not provide facilities to specify this information, so extensions to the language would needed to support this option.

An important topic for future work would be to automate the propagation procedure. The connection between backwards propagation and the weakest precondition semantics suggests that verifying a process satisfies its CAs may be machine-checkable. Alternatively, it may be more efficient to automate propagation in the forwards direction.

The next chapter provides a case study, as empirical evidence to validate the comprehensiveness of the refinement strategy.

7 Case Study

7.1 Introduction

The previous chapters have expounded a formal platform for developing secure software which builds on the *Circus* formal method. This chapter exemplifies how the platform can be applied in practice, by executing a case study development. It explores the general issues that emerge when functionality and confidentiality concerns are united in a system development. It also aims to reinforce the reader's understanding of the techniques presented in the previous chapters.

Even though it is small in scope, this case study aims to validate the platform with respect to its facilities for incorporating confidentiality properties into system developments. Moreover, this case study clarifies the capabilities (and weaknesses) of the platform, enabling us to identify potential avenues for enhancing its scope.

Starting from a *Circus* specification of a system's functionality in Section 7.2, we apply backwards propagation to calculate the inferences of its users in Section 7.3. Thereafter, in Section 7.4, we consider a series of custom confidentiality properties which the system is required to uphold. For each property, we show how the specification can be adjusted to incorporate the property. We also apply backwards propagation to determine whether the resulting specifications are feasible. We aim to demonstrate that backwards propagation is a practicable technique for verifying confidentiality.

Having constructed a consistent process specification incorporating both functionality and confidentiality attributes, we consider a selection of possible design decisions that could be applied to that process in Section 7.5. Again, we verify the correctness of these design decisions with respect to confidentiality by applying backwards propagation. Finally, Section 7.6 summarises some lessons learnt from this case study.

7.2 A Sealed-Bid Auction

We study a *sealed-bid auction* contested by Alice and Bob. In a sealed-bid auction, each contestant independently submits a single bid (for our purposes, a positive integer) to the auctioneer. On receiving both bids, the auctioneer declares the winner of the auction as the contestant who bid the larger amount. Should the bids be equal, the auctioneer is free to choose the winner.

A (fictional) auction house commissions a *Circus* specification of the auctioneer, which is presented in Figure 7.1. This process first initialises its state, then accepts bids from Alice and Bob in an unspecified order, before it announces the winner of the auction and halts. For the purposes of this study, the process is able to choose which bid to accept first.

Notice that this process deadlocks if either contestant fails to submit their bid. A more realistic model would represent the contestants opting either to submit a bid or declining to bid (before a deadline expires), but this detail is irrelevant to our study.

The contestants and the auction house may expect a sealed-bid auction to uphold various kinds of confidentiality requirements. However, the *Auction* process specification tells us nothing about those requirements. These requirements need to be identified and formalised, in order to incorporate them within the process development.

7.3 Modelling Contestants' Inferences

In this section, we consider how backwards propagation can be applied to capture each user's maximal knowledge about the process state, based on its subsequent observation of the process's behaviour. To avoid com-

$BDR ::= Alice \mid Bob$
$\mathbf{channel}\ bidAlice, bidBob : \mathbb{N}_1$
$\mathbf{channel}\ winner : BDR$
$\mathbf{process}\ Auction \triangleq \mathbf{begin}$
$\quad \mathbf{state}\ ASt \triangleq [a, b : \mathbb{N}]$
$\quad Init \triangleq [ASt' \mid a' = 0 \land b' = 0]$
$\quad BidA \triangleq bidAlice?n \to a := n?$
$\quad BidB \triangleq bidBob?n \to b := n?$
$\quad Submit \triangleq (BidA \,;\, BidB) \sqcap (BidB \,;\, BidA)$

$$Declare \triangleq \left(\begin{array}{ll} & a > b \,\&\, winner!Alice \to Stop \\ \Box & a < b \,\&\, winner!Bob \to Stop \\ \Box & a = b \,\&\, \sqcap c : BDR \bullet winner!c \to Stop \end{array} \right)$$

$\quad \bullet\ Init \,;\, Submit \,;\, Declare$
$\mathbf{end}$

Figure 7.1: First model of auctioneer, capturing functionality alone.

plicating matters at this stage, we defer the introduction of confidentiality properties into the development until the next section.

We expect that Alice can observe her own bid (on the *bidAlice* channel), Bob can observe his own bid (on *bidBob*), and both contestants can observe the *winner* channel. In a sealed-bid auction, the contestants submit their bids in private, so we assume that Alice cannot observe *bidBob* and Bob cannot observe *bidAlice*. We specify Alice and Bob's windows as channel sets, as detailed in Subsection 5.4.1:

$\mathbf{channelset}\ A \triangleq \{\!|\ bidAlice, winner\ |\!\}$
$\mathbf{channelset}\ B \triangleq \{\!|\ bidBob, winner\ |\!\}$

Our next task is to lift the *Auction* process. We shall focus on the initialisation, submission and declaration phases of the auction. Hence, we elevate the actions *Init*, *Submit* and *Declare* to the status of blocks. Let:

$$Init_0 \triangleq \langle\, Init \,\rangle \qquad Submit_0 \triangleq \langle\, Submit \,\rangle \qquad Declare_0 \triangleq \langle\, Declare \,\rangle$$

To model Alice and Bob's inferences about the behaviour of *Auction*, we apply full propagation to the lifted process. Since *Auction* is structured sequentially, it is expedient to propagate backwards from *Declare*, as explained in Remark 6.43.

7.3.1 Declaration

The backwards propagation laws given in Subsection 6.4.2 do not extend to blocks with the structure of $Declare_0$. Rather than refining $Declare_0$ into multiple blocks — which would over-approximate Alice and Bob's inferences about the winner of the auction (Subsection 6.2.2) — we instead apply the heuristics described in Subsection 6.4.1 to calculate the state information revealed to Alice and Bob by the behaviour of $Declare_0$.

Since both Alice and Bob can observe *winner* events, their interactions with $Declare_0$ are identical. We break $Declare_0$ apart as follows:

$$
\begin{aligned}
&\mathbf{bw}\,(Declare_0, \theta)\\
= &\left(\begin{array}{ll} & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle\rangle, \theta)\\ \wedge & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle WA\rangle, \theta)\\ \wedge & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle WB\rangle, \theta)\end{array}\right) \qquad \text{[Lemma 6.21]}\\
= &\left(\begin{array}{ll} & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle\rangle \wedge WA \in ref', \theta)\\ \wedge & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle\rangle \wedge WB \in ref', \theta)\\ \wedge & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle WA\rangle, \theta)\\ \wedge & \mathbf{bw}\,(Declare_0 \wedge tr' - tr = \langle WB\rangle, \theta)\end{array}\right)\\
& \qquad \text{[Lemma 6.22]}
\end{aligned}
$$

where WA denotes $(winner, Alice)$ and WB denotes $(winner, Bob)$.

Since $Declare_0$ does not terminate successfully, the choice of θ is irrelevant, because **bw** only imposes θ on terminal states. We consider each part in isolation:

$$
\begin{aligned}
\mathbf{bw}\left(Declare_0 \wedge tr' - tr = \langle\rangle \wedge WA \in ref', \theta\right) &= a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b}\\
\mathbf{bw}\left(Declare_0 \wedge tr' - tr = \langle\rangle \wedge WB \in ref', \theta\right) &= a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}
\end{aligned}
$$

$$\begin{array}{lcl}\mathbf{bw}\left(Declare_0 \wedge tr' - tr = \langle WA \rangle, \theta\right) & = & a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b} \\ \mathbf{bw}\left(Declare_0 \wedge tr' - tr = \langle WB \rangle, \theta\right) & = & a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b}\end{array}$$

Hence, the weakest obligation which left-justifies $Declare_0$ is:

$$bw0 \quad = \quad (a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}) \wedge (a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b})$$

If $a = b$, the non-deterministic choice in $Declare_0$ reveals to the contestants either $a \leq b$ or $a \geq b$. Since we cannot control demonic non-determinism, we must be prepared to accept either case. (The same problem manifests in Example 6.8.) By way of compensation, $bw0$ over-approximates the contestants' inferences about the process state if $a = b$:

$$\begin{array}{lll} & a = b \Rightarrow bw0 & \\ = & a = b \Rightarrow \widetilde{a} \geq \widetilde{b} \wedge \widetilde{a} \leq \widetilde{b} & \text{[prop calc]} \\ = & a = b \Rightarrow \widetilde{a} = \widetilde{b} & \text{[property of } \leq \text{]}\end{array}$$

and so Alice and Bob are modelled as being able to deduce if their bids are equal, even though a single run of $Declare_0$ does not reveal that information to them.

7.3.2 Submission

To back-propagate an obligation through $Submit_0$, we can treat $Submit_0$ as a composite choice (in the sense of Subsection 6.2.2) between $BidA\,;BidB$ and $BidB\,;BidA$. This approach is justified by instantiating Law 6.37:

$$\mathbf{bw}\,(Submit_0, \theta) \quad \sqsubseteq \quad \left(\begin{array}{ll} & \mathbf{bw}\,(\langle\, BidA\,;BidB\,\rangle\,, \theta) \\ \wedge & \mathbf{bw}\,(\langle\, BidB\,;BidA\,\rangle\,, \theta)\end{array}\right)$$

We consider Alice and Bob's inferences about this block separately. With Alice's window, assuming the scope of $n?$ extends beyond the prefixing, the block $\langle A : BidA \,; BidB \rangle$ can be broken down as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\langle A : bidAlice?n \rightarrow a := n? \,; bidBob?n \rightarrow b := n? \rangle} \\ = & \begin{array}{l}\langle A : bidAlice?n \rightarrow Skip \rangle \;; \\ \langle A : a := n? \,; bidBob?n \rightarrow b := n? \rangle\end{array} \qquad \text{[Lemma 5.23]} \\ = & \begin{array}{l}\langle A : bidAlice?n \rightarrow Skip \rangle \;; \langle A : a := n? \rangle \;; \\ \langle A : bidBob?n \rightarrow Skip \rangle \;; \langle A : b := n? \rangle\end{array} \qquad \text{[Lemma 5.18, twice]} \end{array}$$

Hence, an obligation θ can be back-propagated through this specification fragment by appealing to the laws in Section 6.4. With Alice's window, the back-propagation of θ through $BidA$ is:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{bw}\,(\langle A : bidAlice?n \rightarrow Skip \rangle \;; \langle A : a := n? \rangle, \theta)} \\ = & \mathbf{bw}\,(\langle A : bidAlice?n \rightarrow Skip \rangle, \mathbf{bw}\,(\langle A : a := n? \rangle, \theta)) & \text{[Law 6.32]} \\ = & \mathbf{bw}\left(\langle A : bidAlice?n \rightarrow Skip \rangle, \theta[n?, \widetilde{n?}/a, \widetilde{a}]\right) & \text{[Law 6.31]} \\ = & \forall\, n? : \mathbb{N}_1 \bullet \theta[n?, \widetilde{n?}/a, \widetilde{a}][n?/\widetilde{n?}] & \text{[Law 6.36]} \\ = & \forall\, a : \mathbb{N}_1 \bullet \theta[a/\widetilde{a}] & \text{[renaming]} \end{array}$$

Here, the renaming of $\widetilde{a}$ to a is justified because $a = \widetilde{a}$; that is, Alice knows the exact value of her own bid.

With Alice's window, the back-propagation of θ through $BidB$ is:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{bw}\,(\langle A : bidBob?n \rightarrow Skip \rangle \;; \langle A : b := n? \rangle, \theta)} \\ = & \mathbf{bw}\left(\langle A : bidBob?n \rightarrow Skip \rangle, \theta[n?, \widetilde{n?}/b, \widetilde{b}]\right) & \text{[as before]} \\ = & \forall\, n? : \mathbb{N}_1 \bullet \exists\, \widetilde{n?} : \mathbb{N}_1 \bullet \theta[n?, \widetilde{n?}/b, \widetilde{b}] & \text{[Law 6.36]} \\ = & \forall\, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \theta & \text{[renaming]} \end{array}$$

In line with our intuition, this calculation indicates nothing is revealed to Alice about Bob's bid b, except that $b \in \mathbb{N}_1$. Therefore, we have:

$$\mathbf{bw}\,(\langle A : BidA \,; BidB \rangle, \theta) \quad = \quad \forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \theta[a/\widetilde{a}]$$

$$\textsf{bw}\,(\langle\, A\ :\ BidB\,;BidA\,\rangle\,,\theta) \quad = \quad \forall\, a,b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \theta[a/\widetilde{a}]$$

With Bob's window, the back-propagations of θ through *BidA* and *BidB* are symmetric:

$$\begin{aligned}
\textsf{bw}\,(\langle\, B\ :\ BidA\,;BidB\,\rangle\,,\theta) &\quad = \quad \forall\, a,b : \mathbb{N}_1 \bullet \exists\, \widetilde{a} : \mathbb{N}_1 \bullet \theta[b/\widetilde{b}] \\
\textsf{bw}\,(\langle\, B\ :\ BidB\,;BidA\,\rangle\,,\theta) &\quad = \quad \forall\, a,b : \mathbb{N}_1 \bullet \exists\, \widetilde{a} : \mathbb{N}_1 \bullet \theta[b/\widetilde{b}]
\end{aligned}$$

Back-propagating $bw0$ through $\langle\, A\ :\ BidA\,;BidB\,\rangle$ yields:

$$\begin{aligned}
&\textsf{bw}\,(\langle\, A\ :\ BidA\,;BidB\,\rangle\,,bw0) \\
= \quad & \forall\, a,b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet bw0[a/\widetilde{a}] && \text{[as above]} \\
= \quad & \forall\, a,b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet (a \geq b \Rightarrow a \geq \widetilde{b}) \wedge (a \leq b \Rightarrow a \leq \widetilde{b}) \\
= \quad & \textbf{true} && \text{[pred calc]}
\end{aligned}$$

It should be no surprise that $\textsf{bw}\,(\langle\, B\ :\ BidA\,;BidB\,\rangle\,,bw0)$ is also **true**.

By substituting these results into our split form of $Submit_0$:

$$\textsf{bw}\,(Submit_0,\theta) \quad \sqsubseteq \quad \left(\begin{array}{rl} & \textsf{bw}\,(\langle\, BidA\,;BidB\,\rangle\,,\theta) \\ \wedge & \textsf{bw}\,(\langle\, BidB\,;BidA\,\rangle\,,\theta) \end{array}\right)$$

we find that $\textsf{bw}\,(Submit_0,bw0) \sqsubseteq \textbf{true}$, so $\textsf{bw}\,(Submit_0,bw0) = \textbf{true}$.

7.3.3 Initialisation

The backwards propagation of $\textsf{bw}\,(Submit_0,bw0)$ through $Init_0$ is trivial:

$$\textsf{bw}\,(Init_0,\textbf{true}) \quad = \quad \textbf{true}$$

Figure 7.2 presents a fully-justified lifted process form of *Auction* — complete with CAs modelling Alice and Bob's inferences about the process state — based on the calculations we have performed.

$$
\begin{array}{l}
\mathit{BDR} ::= \mathit{Alice} \mid \mathit{Bob} \\
\textbf{channel } \mathit{bidAlice}, \mathit{bidBob} : \mathbb{N}_1 \\
\textbf{channel } \mathit{winner} : \mathit{BDR} \\
\textbf{channelset } A \triangleq \{\!|\ \mathit{bidAlice}, \mathit{winner}\ |\!\} \\
\textbf{channelset } B \triangleq \{\!|\ \mathit{bidBob}, \mathit{winner}\ |\!\} \\
\textbf{process } \mathit{AuctionL} \triangleq \textbf{begin} \\
\quad \textbf{state } \mathit{ASt} \triangleq [a, b : \mathbb{N}] \\
\quad \mathit{Init}_0 \triangleq \langle\, [\mathit{ASt}' \mid a' = 0 \land b' = 0] \,\rangle \\
\quad \mathit{BidA} \triangleq \mathit{bidAlice}?n \to a := n? \\
\quad \mathit{BidB} \triangleq \mathit{bidBob}?n \to b := n? \\
\quad \mathit{Submit}_0 \triangleq \langle\, (\mathit{BidA} \,;\, \mathit{BidB}) \sqcap (\mathit{BidB} \,;\, \mathit{BidA}) \,\rangle \\
\quad \mathit{Declare}_0 \triangleq \left\langle \begin{array}{ll} & a > b \,\&\, \mathit{winner}!\mathit{Alice} \to \mathit{Stop} \\ \Box & a < b \,\&\, \mathit{winner}!\mathit{Bob} \to \mathit{Stop} \\ \Box & a = b \,\&\, \sqcap c : \mathit{BDR} \bullet \mathit{winner}!c \to \mathit{Stop} \end{array} \right\rangle \\
\quad \mathit{bw0} \triangleq (a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}) \land (a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b}) \\
\quad \bullet \langle\, \textbf{true} \,\rangle \,;\, \mathit{Init}_0 \,;\, \langle\, \textbf{true} \,\rangle \,;\, \mathit{Submit}_0 \,;\, \langle\, \mathit{bw0} \,\rangle \,;\, \mathit{Declare}_0 \,;\, \langle\, \textbf{true} \,\rangle \\
\textbf{end}
\end{array}
$$

Figure 7.2: Lifted and fully-justified model of auctioneer.

7.4 Specifying Confidentiality Properties

We now investigate how some example confidentiality properties can be expressed over $Auction_0$, and verify whether they are consistent with the pre-existing functionality of the process by applying backwards propagation.

In practice, it would be expected that backwards propagation should be performed *after* a specification covering functionality and confidentiality properties has been produced, to verify these properties are compatible before proceeding with the development. When we are reasonably confident that these properties are compatible, this approach would be most expedient. However, we take the opposite approach in this section for pedagogic reasons. Our goals here are to focus on how individual confidentiality requirements can be encoded; and to investigate how they may affect the feasibility of a process. These goals are best served by considering each requirement in isolation, as we proceed to do.

7.4.1 Comparing Bids

The auction house's first confidentiality requirement is:

CP1 The contestants cannot establish if Alice's bid is larger than Bob's.

We capture **CP1** by inserting a CA into the body of the process, between $Submit_0$ and $Declare_0$:

$$Init_0\,;Submit_0\,;\langle\, a > b \Rightarrow \widetilde{a} \leq \widetilde{b}\,\rangle\,;Declare_0$$

To determine whether this CA is consistent with the process, we compose it with the back-propagation $bw0$ of $Declare_0$:

$$\begin{aligned}
&\langle\, a > b \Rightarrow \widetilde{a} \leq \widetilde{b}\,\rangle\,;\langle\,(a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}) \wedge (a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b})\,\rangle \\
=\quad &\langle\, a > b \Rightarrow \widetilde{a} \leq \widetilde{b} \wedge (a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}) \wedge (a \leq b \Rightarrow \widetilde{a} \leq \widetilde{b})\,\rangle \text{[Law 5.41]} \\
=\quad &\langle\,(a \geq b \Rightarrow \widetilde{a} = \widetilde{b}) \wedge (a < b \Rightarrow \widetilde{a} \leq \widetilde{b})\,\rangle \qquad \text{[prop calc]}
\end{aligned}$$

The obligation tells us that $\mathit{Declare}_0$ does not leak information that violates **CP1**. Since this obligation is stronger than $\mathit{bw}0$, we still need to propagate it backwards through Submit_0, to ensure Submit_0 is left-justified. As in Subsection 7.3.2, we calculate:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{bw}\left(\mathit{Submit}_0, (a \geq b \Rightarrow \widetilde{a} = \widetilde{b}) \wedge (a < b \Rightarrow \widetilde{a} \leq \widetilde{b})\right)} \\ = & \forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet ((a \geq b \Rightarrow \widetilde{a} = \widetilde{b}) \wedge (a < b \Rightarrow \widetilde{a} \leq \widetilde{b}))[a/\widetilde{a}] \\ = & \mathbf{true} \qquad\qquad\qquad\qquad\qquad\qquad [\text{renaming, pred calc}] \end{array}$$

This calculation indicates that **CP1** is upheld, because Alice cannot rule out the possibility that her own bid is equal to Bob's. Again, the calculation with Bob's window is symmetric.

Since this obligation matches the back-propagation of $\mathit{bw}0$ through Submit_0, we need not back-propagate it further. Hence, we can ascertain the process embedding the CA corresponding to **CP1** is self-consistent.

7.4.2 Value of Bids

The second confidentiality requirement we consider is:

CP2 Neither contestant can deduce if the other's bid exceeds £100.

We capture **CP2** by inserting two separate CAs over Alice and Bob's inferences about each other's bids. Let $\mathit{CP2A}$ specify that Bob cannot deduce that Alice's bid exceeds £100:

$$\mathit{CP2A} \triangleq \langle\, B \,:\, a > 100 \Rightarrow \widetilde{a} \leq 100 \,\rangle$$

Likewise, let $\mathit{CP2B}$ specify the complementary condition for Alice:

$$\mathit{CP2B} \triangleq \langle\, A \,:\, b > 100 \Rightarrow \widetilde{b} \leq 100 \,\rangle$$

We cannot insert $\mathit{CP2A}$ and $\mathit{CP2B}$ into BidA and BidB without splitting Submit_0 into multiple blocks. Instead, we insert $\mathit{CP2A}$ and $\mathit{CP2B}$

immediately following $Submit_0$ in the process body:

$$Init_0 \,;Submit_0 \,;CP2A \,;CP2B \,;\langle\, bw0 \,\rangle \,;Declare_0$$

To left-justify $Submit_0$ with respect to $CP2A$ and $CP2B$, we back-propagate each CA through $Submit_0$ separately. Here, we focus on $CP2B$:

$$\begin{array}{ll}
\textsf{bw}\,(Submit_0, bw0 \wedge CP2B) & \\
\forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet (bw0 \wedge (b > 100 \Rightarrow \widetilde{b} \leq 100))[a/\widetilde{a}] & \text{[as above]} \\
\forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \left(\begin{array}{c} bw0[a/\widetilde{a}] \wedge \widetilde{b} \leq 100 \\ \lhd\, a > 100 \wedge a < b \,\rhd \\ bw0[a/\widetilde{a}] \wedge (b > 100 \Rightarrow \widetilde{b} \leq 100) \end{array}\right) & \\
 & \text{[renaming; case split]}
\end{array}$$

The case split is helpful for evaluating the truth of this predicate. Focusing on the case where Alice's bid exceeds £100 but is smaller than Bob's bid, we have:

$$\begin{array}{lll}
\multicolumn{2}{l}{a > 100 \wedge a < b \Rightarrow (a \geq b \Rightarrow a \geq \widetilde{b}) \wedge (a \leq b \Rightarrow a \leq \widetilde{b}) \wedge \widetilde{b} \leq 100} & \\
= & a > 100 \wedge a < b \Rightarrow a \leq \widetilde{b} \wedge \widetilde{b} \leq 100 & \text{[pred calc]} \\
= & a > 100 \wedge a < b \Rightarrow a \leq \widetilde{b} \wedge \widetilde{b} \leq 100 \wedge a \leq 100 & \text{[property of } \leq\text{]} \\
= & a > 100 \wedge a < b \Rightarrow \mathbf{false} & \text{[contradiction]}
\end{array}$$

Because Bob is declared the winner in this case, Alice can infer that Bob's bid is over £100. Hence, $\textsf{bw}\,(Submit_0 \,;CP2B, bw0)$ reduces to:

$$\begin{array}{lll}
\multicolumn{2}{l}{\begin{array}{l}\forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \\ \quad \neg\,(a > 100 \wedge a < b) \wedge bw0[a/\widetilde{a}] \wedge (b > 100 \Rightarrow \widetilde{b} \leq 100)\end{array}} & \text{[as above]} \\
= & \mathbf{false} & \text{[pred calc]}
\end{array}$$

This calculation tells us the specification fragment:

$$Submit_0 \,;CP2B \,;Declare_0$$

may exhibit miraculous behaviour: should Alice's bid exceed £100 but be smaller than Bob's bid, the process would fail to announce the winner of the auction, to prevent Alice's inferences from violating **CP2**. Hence, the **CP2** property is inconsistent with the functionality of the process.

In this case study, we proceed with the process development in the absence of **CP2**. In practice, if the auction house's requirements for the system included **CP2**, then it would be the responsibility of the system developers to seek clarification from the auction house to resolve the inconsistency.

7.4.3 Order of Bids

The third confidentiality requirement we consider is:

CP3 Neither contestant can learn who placed the first bid.

We formalise **CP3** by refining $Submit_0$. Applying a tactic described in Subsection 5.6.2, we introduce a local variable f into $Submit_0$ to record which contestant bids first. Then, **CP3** can be expressed with a CA referencing f, as follows:

$$\begin{aligned}
&SubmitAux \triangleq \langle f = Alice \,\&\, BidA \,; BidB \,\square\, f = Bob \,\&\, BidB \,; BidA \,\rangle \\
&CP3 \triangleq \langle \widetilde{f} = opp(f) \,\rangle \\
&Submit_1 \triangleq \mathbf{var}\, f \bullet \langle f := Alice \sqcap f := Bob \,\rangle \;; CP3 \,; SubmitAux
\end{aligned}$$

where opp is a function that maps each contestant to its opponent:

$$\begin{array}{|l}
opp : BDR \rightarrow BDR \\
\hline
opp(Alice) = Bob \\
opp(Bob) = Alice
\end{array}$$

Here, the CA $CP3$ specifies that Bob bidding first serves as a cover story for Alice bidding first ($f = Alice \Rightarrow \widetilde{f} = Bob$) and vice versa. At first glance, we may be tempted to say this CA is upheld by the process, but we still need to apply back-propagation to be sure.

To back-propagate an obligation θ through *SubmitAux*, we again resort to heuristics. Reasoning from Alice's perspective, we have:

$$
\begin{array}{l}
\mathbf{bw}\,(SubmitAux, \theta) \\
= [\,\text{Lemma 6.23; Lemma 6.21; Lemma 6.22}\,] \\
\left(\begin{array}{ll}
 & \mathbf{bw}\,(SubmitAux \wedge wait' \wedge \neg\, TA \wedge RA, \theta) \\
\wedge & \mathbf{bw}\,(SubmitAux \wedge wait' \wedge \neg\, TA \wedge \neg\, RA, \theta) \\
\wedge & \mathbf{bw}\,(SubmitAux \wedge wait' \wedge TA, \theta) \\
\wedge & \mathbf{bw}\,(SubmitAux \wedge \neg\, wait', \theta)
\end{array}\right)
\end{array}
$$

where $\mathcal{A}$ denotes the set of events in window A in the following:

$$
\begin{array}{lcl}
TA & = & \exists\, n : \mathbb{N}_1 \bullet (tr' - tr) \upharpoonright \mathcal{A} = \langle (bidAlice, n) \rangle \\
RA & = & \exists\, n : \mathbb{N}_1 \bullet (bidAlice, n) \in ref' \cap \mathcal{A}
\end{array}
$$

The first **bw** conjunct describes Alice's inferences when she attempts to submit her bid, but finds the process refuses to accept it. The only explanation for this circumstance is that the process is deadlocked, waiting for Bob to submit his bid. Hence:

$$\mathbf{bw}\,\big(SubmitAux \wedge wait' \wedge \neg\, TA \wedge RA, \theta\big) = (f = Bob \Rightarrow \widetilde{f} = Bob)$$

(Obligation θ is irrelevant here, because we are considering only those behaviours where *SubmitAux* does not terminate.)

The second **bw** construct describes Alice's inferences when the process does not refuse her bid. It reveals no information to Alice about f:

$$\mathbf{bw}\,\big(SubmitAux \wedge wait' \wedge \neg\, TA \wedge RA, \theta\big) \quad = \quad \mathbf{true}$$

The third **bw** conjunct describes Alice's inferences when the process is deadlocked, but she has submitted her bid. Again, this circumstance can only be explained by the process waiting for Bob to submit his bid.

Hence:

$$\textbf{bw}\left(SubmitAux \wedge wait' \wedge TA, \theta\right) \quad = \quad (f = Alice \Rightarrow \widetilde{f} = Alice)$$

The fourth **bw** conjunct just back-propagates θ through *SubmitAux*:

$$\textbf{bw}\left(SubmitAux \wedge \neg\, wait', \theta\right) \quad = \quad \forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet \theta[a/\widetilde{a}]$$

Assembling these four conjuncts yields:

$$\textbf{bw}\,(SubmitAux, bw0) \quad = \quad (f \in BDR \Rightarrow f = \widetilde{f})$$

We now back-propagate this obligation through $CP3$:

$$\begin{aligned}
&\textbf{bw}\left(\langle \widetilde{f} = opp(f)\,\rangle, (f \in BDR \Rightarrow f = \widetilde{f})\right) \\
= \quad & \widetilde{f} = opp(f) \wedge (f \in BDR \Rightarrow f = \widetilde{f}) && \text{[Law 6.18]}
\end{aligned}$$

and then back-propagate through the remaining block of $Submit_1$:

$$\begin{aligned}
&\textbf{bw}\left(\langle f := Alice \sqcap f := Bob\,\rangle, \widetilde{f} = opp(f) \wedge (f \in BDR \Rightarrow f = \widetilde{f})\right) \\
= \quad & \exists f, \widetilde{f} \bullet f \in BDR \wedge \widetilde{f} \in BDR \wedge \widetilde{f} = opp(f) \wedge (f \in BDR \Rightarrow f = \widetilde{f}) \\
= \quad & \exists f, \widetilde{f} \bullet f \in BDR \wedge \widetilde{f} \in BDR \wedge \widetilde{f} = opp(f) \wedge f = \widetilde{f} && \text{[prop calc]} \\
= \quad & \exists f, \widetilde{f} \bullet f \in BDR \wedge \widetilde{f} \in BDR \wedge \widetilde{f} \neq f \wedge f = \widetilde{f} && \text{[property of } opp\text{]} \\
= \quad & \textbf{false} && \text{[pred calc]}
\end{aligned}$$

This result informs us that our definition of $Submit_1$ is inconsistent with respect to its embedded CA. Intuitively, **CP3** is violated (by the unlifted *Auction* process) because if either contestant fails to submit a bid, then the other contestant can deduce the bidding order by observing:

- either the process's refusal to accept their own bid;
- or the process's refusal to supply the winner of the auction, after the contestant has submitted their own bid.

Were it the case that our model of Alice's and Bob's observations (as given by Definition 5.1) did not include a projection of the refusals of a process, then **CP3** would not induce an inconsistency with $Submit_1$.

If the auction house is prepared to relax the **CP3** requirement to:

CP4 *Under the circumstances that both bids are submitted*, neither contestant can learn who placed the first bid.

we can redefine $Submit_1$ as:

$$Submit_2 \triangleq \mathbf{var}\, f \bullet \langle f := Alice \sqcap f := Bob \rangle \; ; SubmitAux \, ; CP3$$

Here, the CA $CP3$ is invoked only after Alice and Bob have submitted their bids. Even though $Submit_2$ uses the same CA as before, this specification is weaker than $Submit_2$.

Now, we have:

$$\mathbf{bw}\left(SubmitAux \wedge \neg\, wait', \widetilde{f} = opp(f) \wedge bw0\right) \quad = \quad \widetilde{f} = opp(f)$$

and back-propagating through the remaining block of $Submit_2$ gives:

$$\mathbf{bw}\left(\langle f := Alice \sqcap f := Bob \rangle, \widetilde{f} = opp(f)\right) \quad = \quad \mathbf{true}$$

By the same argument as that given by Subsection 7.4.1, we can conclude the *AuctionLC* process (Figure 7.3) — featuring an explicit specification of **CP1** and **CP4** — is consistent with respect to our model of Alice and Bob's observational abilities.

7.5 Potential Refinement Steps

Following Section 6.6, we now consider briefly some refinement steps that could be applied to the consistent *AuctionLC* process.

$$
\begin{array}{l}
BDR ::= Alice \mid Bob \\
\mathbf{channel}\ bidAlice, bidBob : \mathbb{N}_1 \\
\mathbf{channel}\ winner : BDR \\
\mathbf{channelset}\ A \mathrel{\widehat{=}} \{\!|\ bidAlice, winner\ |\!\} \\
\mathbf{channelset}\ B \mathrel{\widehat{=}} \{\!|\ bidBob, winner\ |\!\} \\
\mathbf{process}\ AuctionLC \mathrel{\widehat{=}} \mathbf{begin} \\
\quad \mathbf{state}\ ASt \mathrel{\widehat{=}} [a, b : \mathbb{N}] \\
\quad CP1 \mathrel{\widehat{=}} \langle\, a > b \Rightarrow \widetilde{a} \leq \widetilde{b} \,\rangle \\
\quad CP4 \mathrel{\widehat{=}} \langle \widetilde{f} = opp(f) \,\rangle \\
\quad Init_0 \mathrel{\widehat{=}} \langle\, [ASt' \mid a' = 0 \land b' = 0] \,\rangle \\
\quad BidA \mathrel{\widehat{=}} bidAlice?n \to a := n? \\
\quad BidB \mathrel{\widehat{=}} bidBob?n \to b := n? \\
\quad SubmitAux \mathrel{\widehat{=}} \langle f = Alice \mathbin{\&} BidA \,; BidB \mathbin{\Box} f = Bob \mathbin{\&} BidB \,; BidA \,\rangle \\
\quad Submit_2 \mathrel{\widehat{=}} \mathbf{var}\, f \bullet \langle f := Alice \sqcap f := Bob \,\rangle \,; SubmitAux \,; CP4 \\
\quad Declare_0 \mathrel{\widehat{=}} \left\langle \begin{array}{ll} & a > b \mathbin{\&} winner!Alice \to Stop \\ \Box & a < b \mathbin{\&} winner!Bob \to Stop \\ \Box & a = b \mathbin{\&} \sqcap c : BDR \bullet winner!c \to Stop \end{array} \right\rangle \\
\quad \bullet\ Init_0 \,; Submit_2 \,; CP1 \,; Declare_0 \\
\mathbf{end}
\end{array}
$$

Figure 7.3: Lifted model of auctioneer, capturing **CP1** and **CP4**.

7.5.1 Bidding Order

We could refine $Submit_2$ so that Alice always bids first:

$$Submit_3 \triangleq \mathbf{var}\, f \bullet \langle f := Alice \rangle \;;\; \langle\, BidA\,;BidB \,\rangle \;;CP4$$

This refinement is naïve: by fixing the order of bidding, it is clear that **CP4** is violated. Formally, this inconsistency would be detected by back-propagating $CP4$ through the preceding blocks.

An alternative refinement of $Submit_2$ would offer the choice between Alice and Bob's bids to the environment:

$$Submit_4 \triangleq \mathbf{var}\, f \bullet \left\langle \begin{array}{ll} & BidAlice\,;BidBob\,;f := Alice \\ \Box & BidBob\,;BidAlice\,;f := Bob \end{array} \right\rangle ;CP4$$

Back-propagating $CP4$ through $Submit_4$ yields **true**, as did $Submit_2$. Hence, we conclude that $Submit_4$ is a feasibility-preserving refinement of $Submit_2$.

7.5.2 Deterministic Declaration

Suppose the auctioneer is instructed to declare Bob as the winner only if his bid is strictly greater than Alice's. This instruction clarifies the behaviour of $Declare_0$ by refinement:

$$Declare_1 \triangleq a \geq b \,\&\, winner!Alice \rightarrow Stop \;\Box\; a < b \,\&\, winner!Bob \rightarrow Stop$$

By observing a *winner* event, the contestants can infer whether $a \geq b$ or $a < b$, as is reflected by:

$$\mathbf{bw}\,(Declare_1, \mathbf{true}) \quad = \quad (a \geq b \Rightarrow \widetilde{a} \geq \widetilde{b}) \wedge (a < b \Rightarrow \widetilde{a} < \widetilde{b})$$

This obligation is neither weaker nor stronger than $bw0$. Back-propagating it through the CA $CP1$ in $AuctionLC$ gives the obligation:

$$(a > b \Rightarrow \widetilde{a} = \widetilde{b}) \wedge (a = b \Rightarrow \widetilde{a} \geq \widetilde{b}) \wedge (a < b \Rightarrow \widetilde{a} < \widetilde{b})$$

Now, back-propagating this obligation through $Submit_2$ yields **true**: the extra information that $Declare_1$ reveals does not violate **CP1** or **CP4**. Hence, the refinement of $Declare_0$ into $Declare_1$ is feasibility-preserving.

7.5.3 Data Refinement

One possible data refinement on *AuctionLC* would be to replace the state *ASt* with three new variables, linked by the simulation relation:

$$R \quad = \quad \left(\begin{array}{c} c = \max(a,b) \wedge (e = 1 \lhd a = b \rhd e = 0) \\ \wedge \quad (a > b \Rightarrow w = Alice) \wedge (a < b \Rightarrow w = Bob) \end{array} \right)$$

Here, c stores the value of the larger bid, while w records the identity of the contestant who made that bid. The binary-valued variable e records whether the bids are equal. It serves no purpose with respect to the functionality of the specification.

By applying this data refinement to *AuctionLC*, we can derive the process presented in Figure 7.4. Notice how this data refinement entails the restructuring of $CP1$, by projecting its embedded obligation through the simulation:

$$\begin{array}{lll} & \forall a, b, \widetilde{a}, \widetilde{b} \bullet \mathbf{U}(R) \Rightarrow (a > b \Rightarrow \widetilde{a} \leq \widetilde{b}) & \\ = & e = 0 \wedge w = Alice \Rightarrow \widetilde{e} = 1 \vee \widetilde{w} = Bob & \text{[pred calc]} \end{array}$$

Here, the purpose of e becomes apparent. It is needed to capture the exact concrete fog states that R maps to $\widetilde{a} \leq \widetilde{b}$ in the abstract space, without strengthening the CA.

An implicit design decision captured by the $BidA_D$ and $BidB_D$ actions (see Figure 7.4) is that, if Alice and Bob's bids are equal, then the winner is the contestant who bid first. However, this decision harbours yet another subtle security flaw, with respect to **CP4**. Suppose that Alice bids the minimum amount £1 but is then declared the winner. Alice can infer that Bob's bid must also be £1 and, by extension, that she must have bid first, in violation of $CP4_D$.

$$
\begin{array}{l}
\mathit{BDR} ::= \mathit{Alice} \mid \mathit{Bob} \\
\textbf{channel}\ \mathit{bidAlice}, \mathit{bidBob} : \mathbb{N}_1 \\
\textbf{channel}\ \mathit{winner} : \mathit{BDR} \\
\textbf{channelset}\ A \triangleq \{\!|\ \mathit{bidAlice}, \mathit{winner}\ |\!\} \\
\textbf{channelset}\ B \triangleq \{\!|\ \mathit{bidBob}, \mathit{winner}\ |\!\} \\
\textbf{process}\ \mathit{AuctionLCD} \triangleq \textbf{begin} \\
\quad \textbf{state}\ \mathit{CSt} \triangleq [c : \mathbb{N}, e : \{0,1\}, w : \mathit{BDR}] \\
\quad \mathit{CP1}_D \triangleq \langle\, e = 0 \land w = \mathit{Alice} \Rightarrow \widetilde{e} = 1 \lor \widetilde{w} = \mathit{Bob}\,\rangle \\
\quad \mathit{CP4}_D \triangleq \langle \widetilde{f} = \mathit{opp}(f)\,\rangle \\
\quad \mathit{Init}_D \triangleq \langle\, [\mathit{CSt}' \mid c' = 0 \land e' = 1 \land w' = \mathit{Alice}]\,\rangle \\
\quad \mathit{BidA}_D \triangleq \mathit{bidAlice}?n \rightarrow \left[\begin{array}{l|l} \Delta \mathit{CSt} & c' = \max(c, n?) \\ n? : \mathbb{N}_1 & e' = 1 \lhd c = n? \rhd e' = 0 \\ & w' = \mathit{Alice} \lhd c > n? \rhd w' = w \end{array}\right] \\
\quad \mathit{BidB}_D \triangleq \mathit{bidBob}?n \rightarrow \left[\begin{array}{l|l} \Delta \mathit{CSt} & c' = \max(c, n?) \\ n? : \mathbb{N}_1 & e' = 1 \lhd c = n? \rhd e' = 0 \\ & w' = \mathit{Bob} \lhd c > n? \rhd w' = w \end{array}\right] \\
\quad \mathit{SubmitAux}_D \triangleq \langle f = \mathit{Alice} \,\&\, \mathit{BidA}_D \,;\mathit{BidB}_D \,\Box\, f = \mathit{Bob} \,\&\, \mathit{BidB}_D \,;\mathit{BidA}_D\,\rangle \\
\quad \mathit{Submit}_D \triangleq \textbf{var}\, f \bullet \langle f := \mathit{Alice} \sqcap f := \mathit{Bob}\,\rangle\ ;\mathit{SubmitAux}_D\,;\mathit{CP4}_D \\
\quad \mathit{Declare}_D \triangleq \langle\, \mathit{winner}!w \rightarrow \mathit{Stop}\,\rangle \\
\quad \bullet\ \mathit{Init}_D\,;\mathit{Submit}_D\,;\mathit{CP1}_D\,;\mathit{Declare}_D \\
\textbf{end}
\end{array}
$$

Figure 7.4: Lifted and data-refined model of auctioneer.

Ignoring $CP1_D$ for simplicity, we can confirm the presence of this security flaw in the usual fashion. Back-propagating the obligation:

$$\mathbf{bw}\,(CP4_D, \mathbf{bw}\,(Declare_D, \mathbf{true})) \quad = \quad w = \widetilde{w} \wedge \widetilde{f} = opp(f)$$

through $\langle\, A \,:\, Submit_D \,\rangle$ and simplifying yields the following:

$$\forall\, a, b : \mathbb{N}_1 \bullet \exists\, \widetilde{b} : \mathbb{N}_1 \bullet (a \geq b \wedge a > \widetilde{b}) \vee (a > b \wedge a \geq \widetilde{b})$$

When $a = 1$ and $b = 1$, no values of $\widetilde{b} : \mathbb{N}_1$ can satisfy this condition. Hence, we have a contradiction and can therefore establish the process is insecure with respect to **CP4**.

7.6 Discussion

This case study has demonstrated that software development is made more complicated by the presence of confidentiality properties. Even a superficially simple process can harbour surprisingly deep subtleties that must be probed to acquire robust assurances of security.

We have used back-propagation as a tool to detect inconsistencies between the functionality and confidentiality requirements placed on the auctioneer. This analysis has revealed subtle security issues: in particular, the conflict between **CP3** and the *Auction* process (described in Subsection 7.4.3) may not be immediately obvious.

The verification procedure is tractable for a small case study, but the lack of tool support would obstruct its deployment in a larger case study. As we have seen, calculating back-propagations through *SubmitAux* and $Declare_0$ by hand is tricky and requires extra creativity in places. Moreover, the potential for introducing errors can only be magnified for more complex blocks.

8 Conclusions

8.1 Summary of the Thesis

This thesis presents a formal platform for specifying, verifying and developing software systems with respect to both functionality and confidentiality properties. This platform leads towards a methodology for constructing software that is "secure by design".

The platform's backbone is the Unifying Theories of Programming. We have created a generic approach for extending UTP theories to model users' observational abilities and, by extension, their inferences about a system's behaviour. This approach enables confidentiality properties to be specified alongside a functional model of a system. In this way, we reconcile the opposing concerns of functionality and confidentiality in a clean and appealing fashion.

We focus on making the approach practical by specialising it to *Circus*. We extend the *Circus* notation with facilities to embed confidentiality properties within a functional process specification. This work engenders a language that makes the specification of confidentiality properties accessible to formal methods practitioners who are non-specialists in the field of information security. These specifications can be subjected to rigorous consistency checks, to detect potential sources of insecurity.

Much of the work presented in this thesis draws inspiration from existing ideas on information flow security in the academic literature. The originality of the thesis lies in how we have distilled these ideas to their essence by characterising them in the UTP, enabling us to synthesise new connections between these ideas and the field of formal methods.

The work reported in this thesis could be ported to other formal methods with a UTP semantics, or other languages in the *Circus* family. However, it is unclear how our encoding of confidentiality properties could be replicated for languages without a UTP-like semantics.

We now review the main contributions of the thesis, discuss their significance and highlight areas for improvement.

8.1.1 Modelling User Interactions

Chapter 3 proposes a framework for exploring specifications of systems from the perspective of their users. The key component of this framework is the view, which formalises a user's interface to a system. Views can be applied to translate a model of system behaviours into a model of user interactions, which can reveal defects in a system's functionality. Conversely, views can be used to calculate a user's inferences about a system's behaviour.

This framework is generic in the UTP theory framing a system model. We have applied the framework to the UTP theories of designs and reactive processes; and, in the latter case, we have demonstrated that the framework generalises Roscoe's notion of lazy abstraction.

8.1.2 Encoding Confidentiality

Chapter 4 extends the framework of Chapter 3 to formalise information flow from systems to adversarial users. Parameterised by an indistinguishability relation between system behaviours — constructed from a view — this encoding of information flow is generic across UTP theories.

We harness the lifted space of indistinguishable behaviours to capture notions of confidentiality. Obligations are closure conditions over the lifted space, which specify an upper bound on an adversary's inferences about a system's behaviour. We realise a specification of a secure system by conjoining obligations with a lifted model of the system's functionality, to eliminate any system behaviours that violate an obligation.

The fusion of functionality and confidentiality presented in this chapter is novel. The application of miracles to denote conflicting specifications of functionality and confidentiality is especially appealing, yet is only possible because miracles are valid specification artefacts in the UTP. Moreover, miracles solve the long-standing "refinement paradox" in a clean fashion: refinement can make system designs miraculous, but it cannot make them insecure.

Obligations capture a more liberal notion of confidentiality than noninterference. They are based on the premise that software engineers would find it advantageous to tailor the specification of confidentiality properties to the system domain, as opposed to aiming to satisfy rigid noninterference properties. Even though noninterference is arguably too strict for many classes of systems, our premise can only be vindicated with empirical evidence.

8.1.3 Integrating Confidentiality with *Circus*

Chapter 5 specialises the confidentiality encoding from Chapter 4 to cover *Circus* processes. Targeting *Circus* is a pragmatic choice, owing to its UTP semantics, its unified treatment of state and behavioural aspects of systems, and its comprehensive refinement strategy.

At the semantic level, the integration is accomplished by lifting *Circus* constructs to represent information flow to the adversary, using the techniques presented in Chapter 4. This approach complicates the semantics of these constructs, but in return, it affords reasoning about information flow in a compositional fashion.

Confidentiality annotations generalise *Circus* coercions to encode confidentiality properties over processes in terms of their intermediate state. The semantics of confidentiality annotations makes the insecure behaviours of a process miraculous, thereby preventing the process from leaking secret information to an adversary. A process specification can only be strengthened by inserting confidentiality annotations, so they can be introduced at any stage of a process's development.

Much of the expressive power of confidentiality annotations is exposed by composing them with lifted *Circus* constructs, to specify confidentiality properties over the events a process performs. Taking this idea to an extreme, confidentiality annotations can be formulated as separate processes, which can be superposed with a functional process to create a secure specification of a system.

8.1.4 Verifying Security

Chapter 6 focuses on the development of processes with confidentiality annotations by stepwise refinement, from abstract specifications all the way to the implementation level.

While confidentiality annotations ensure confidentiality is preserved by refinement, ill-judged specifications or imprudent refinement steps can make a process infeasible. This effect motivates us to extend the *Circus* refinement strategy with special measures to ensure that each process design in the refinement chain is implementable.

This chapter presents backwards propagation, a technique for systematically and tractably over-approximating an adversary's inferences about the process state at each step of the process's execution. These derived inferences can be checked against the confidentiality annotations within the process body, to verify those annotations are respected throughout the process's execution.

Backwards propagation also enables software engineers to verify that refinement steps are feasibility-preserving. Rather than verifying each process design in isolation, we can record the adversary's inferences by inserting confidentiality annotations at each point of the process. Then, these annotations can be used to verify a localised refinement step, without incurring the expense of re-verifying the whole process.

8.2 Directions for Future Research

There is still much work to be done in scaling up the capabilities of the formal development platform beyond a proof of concept. We now review a selection of topics that merit further research.

8.2.1 Tool Support

Effective machine assistance is often cited as important for encouraging the industrial adoption of formal methods (Saiedian, 1996; Woodcock et al., 2009). The lack of any dedicated tool support for our platform is arguably the main impediment to its deployment.

Some possible goals of tool support for the platform would include the ability to (semi)-automatically verify that a specification of functionality and confidentiality properties is consistent; and to determine whether this consistency is maintained by refinement steps.

The design of the tools may be based on techniques in this thesis (such as backwards propagation) or may involve other techniques devised expressly with automation in mind. Work by Černý (2009) indicates that verifying a restricted space of terminating programs against a class of confidentiality properties (much like our own) can be reduced to a decision problem that can be tackled using a SMT solver. As with all automated verification techniques, restrictions on the program space are necessary to avoid the halting problem.

One pathway towards implementing effective tool support would be to adapt existing *Circus* tools, such as CRefine (Oliveira et al., 2008). Extensions to these tools would include facilities to specify confidentiality annotations; adaptations of the refinement rules to discriminate between unlifted and lifted constructs; and potentially an implementation of the backwards propagation calculus. Alternatively, if the platform is ported to a formalism such as CSP||B (Schneider and Treharne, 2005) or Event-B (Abrial, 2010), tool support for verifying confidentiality could potentially be implemented in the form of plug-ins for their respective toolsets.

8.2.2 Mechanising the Theory

All theorems, lemmas and laws presented in this thesis have been justified by hand proof. Nevertheless, it would be expedient to verify their correctness by encoding their proofs in a theorem prover. The *Saoithín* proof checker for UTP (Butterfield, 2010), or the *Isabelle/Circus* environment

(Feliachi et al., 2011), could be applied to this task.

8.2.3 Case Studies

The purpose of carrying out case studies is to evaluate the platform's suitability for the development of secure software. Owing to time constraints and the lack of tool support, the case study and examples presented in this thesis are small in scope. Larger case studies are needed to evaluate the platform comprehensively.

Two pertinent topics for larger case study developments with the platform are outlined thus:

Validating access controls A specification of an access control mechanism could be augmented with confidentiality annotations, in order to specify explicitly the confidentiality policy that (it is claimed) is modelled by that mechanism. By combining this augmented specification with a system design, a specification could be derived that cannot leak information in contravention of the confidentiality policy, regardless of whether the mechanism captures the policy correctly. An insecure implementation along the lines of McLean's "System Z" (see Section 4.1) would then be unattainable.

Security protocol development A security protocol could be formalised in lifted *Circus*, alongside a Dolev-Yao model of an adversary's abilities to tamper with the protocol (Dolev and Yao, 1983). The secrecy properties of the protocol would be specified by inserting confidentiality annotations into the protocol design. Once these secrecy properties are verified to hold, it would then be appropriate to verify that tactics for refining the protocol design — for instance, by fixing the order of tokens within messages — do not introduce insecurities into the concrete protocol design.

A complete delivery of these case studies would provide convincing evidence that, with suitable tool assistance, the platform could potentially be scaled up to accommodate real-world software engineering projects.

8.2.4 A Confidentiality Policy Language

Our encoding of confidentiality properties — as relations between two observational spaces of a system — is very general, but it is perhaps not readily comprehensible by software engineers. Moreover, interweaving confidentiality annotations with lifted *Circus* actions (as described in Section 5.6) is a somewhat esoteric means for indirectly expressing confidentiality properties over events. These tricks may confuse readers of a specification, especially if the meaning of the confidentiality annotations is not documented.

It would be worthwhile to define a dedicated language for expressing confidentiality policies in practice. Shaped by the experience grown through case studies, this language would feature idioms for expressing common confidentiality properties over the state and behaviour of processes in a clear and intuitive fashion. The language could be designed with automated analysis in mind: restricting the expressiveness of the idioms would simplify the task of building effective tool support to check those idioms.

Some degree of translation would be needed to integrate this policy specification language into our development platform (in its current form). This translation could be implemented as a tool, taking a confidentiality policy and a *Circus* process as input, and generating a process with embedded confidentiality annotations capturing the policy. The annotated process could then be developed using the platform.

8.2.5 Probabilistic Information Flow

The model of information flow considered in this thesis is *possibilistic*, rather than probabilistic: it deals only with what an adversary can infer *with certainty* about a system. The inherent limitations of possibilistic information flow models are discussed in Section 4.7.

Bresciani and Butterfield (2011) have developed a probabilistic variant of the UTP. If their work is extended to give *Circus* a probabilistic semantics, our platform could potentially be adapted to regulate information flow in a probabilistic setting. An analogue of confidentiality annotations would be embedded in specifications to prevent an adversary from deducing facts about the process state with high probability.

A further extension of this work would involve quantifying the flow of information from systems to users, in terms of Shannon's information theory. In this setting, confidentiality properties would then be defined as an upper bound on the *mutual information* between secret data and outputs to the adversary. There is potential for automatically testing (with statistical significance) whether a program satisfies these properties are satisfied: recent work by Chothia and Guha (2011) has shown how mutual information can be calculated from trial runs of a program alone.

8.2.6 Other Notions of Security

This thesis has focused on a narrow interpretation of security: specifically, limiting the flow of (qualitative) information to a specified adversary. However, information security also encompasses the *authenticity*, the *integrity* and the *availability* of information, as well as its confidentiality.

Integrity properties are concerned with guarding against unauthorised modification of data. Jacob (1987) interprets integrity properties as a dual of confidentiality properties, in the sense that they prevent information from low-level users from flowing to (and corrupting) high-level users. Such properties can be formulated within our platform, simply by inverting the low-level and high-level users.

Availability properties require that information is available to authorised users when it is needed. Techniques for ensuring availability include clustering (to provide fault tolerance), bandwidth management (to mitigate denial-of-service attacks) and failure recovery.

Confidentiality properties have received far more attention in the literature than integrity or availability (Foley, 2005). Indeed, in many classes of systems, safeguarding the integrity and availability of information has a higher priority than maintaining the confidentiality of that information (Clark and Wilson, 1987).

It is interesting to speculate about the potential for integrating specifications of desired security properties other than confidentiality into the software development process. Nevertheless, it may not be appropriate to characterise these properties in terms of information flow, so a new semantic foundation for them may be needed instead.

8.3 Coda

In their book, Hoare and He (1998) set out an agenda to unite the study of different programming paradigms, across multiple levels of abstraction. The work presented in this thesis hints at the potential of extending the UTP agenda to encompass topics in Computing Science that lie outside the standard models of software correctness. In so doing, it underscores the significance of the UTP project for building a firm foundation to address the contemporary challenges in software engineering.

Proofs

Proof of Lemma 3.4

$$\begin{array}{lll}
& \textsf{VH1}\,(V) = V & \\
= & [\,\textsf{VH1}\,(V) \Rightarrow V\,] \wedge [\,V \Rightarrow \textsf{VH1}\,(V)\,] & \text{[equivalence]} \\
= & [\,((\exists\, x_V, x'_V \bullet V) \Rightarrow V) \Rightarrow V\,] \wedge [\,V \Rightarrow ((\exists\, x_V, x'_V \bullet V) \Rightarrow V)\,] & \text{[def } \textsf{VH1}\text{]} \\
= & [\,((\exists\, x_V, x'_V \bullet V) \wedge \neg\, V) \vee V\,] \wedge [\,(V \wedge \exists\, x_V, x'_V \bullet V) \Rightarrow V\,] & \text{[prop calc]} \\
= & [\,((\exists\, x_V, x'_V \bullet V) \wedge \neg\, V) \vee V\,] & \text{[tautology]} \\
= & [\,((\exists\, x_V, x'_V \bullet V) \vee V) \wedge (V \vee \neg\, V)\,] & \text{[distributivity]} \\
= & [\,(\exists\, x_V, x'_V \bullet V) \vee V\,] & \text{[excluded middle]} \\
= & [\,\exists\, x_V, x'_V \bullet V\,] & \text{[lower bound]}
\end{array}$$

Proof of Law 3.8

$$\begin{array}{lll}
& \textsf{VH1} \circ \textsf{VH1}(V) & \\
= & (\exists\, x_V, x'_V \bullet (\exists\, x_V, x'_V \bullet V) \Rightarrow V) \Rightarrow ((\exists\, x_V, x'_V \bullet V) \Rightarrow V) & \text{[def } \textsf{VH1}\text{]} \\
= & \textbf{true} \Rightarrow ((\exists\, x_V, x'_V \bullet V) \Rightarrow V) & \text{[pred calc]} \\
= & (\exists\, x_V, x'_V \bullet V) \Rightarrow V & \text{[prop calc]} \\
= & \textsf{VH1}(V) & \text{[def } \textsf{VH1}\text{]}
\end{array}$$

Proof of Law 3.10

$$\begin{array}{lll}
& \textsf{VH1} \circ \textsf{VH2}(V) & \\
= & (\exists\, x_V, x'_V \bullet (\exists\, x', x'_V \bullet V)) \Rightarrow (\exists\, x', x'_V \bullet V) & \text{[def } \textsf{VH1}, \textsf{VH2}\text{]}
\end{array}$$

$$
\begin{array}{lll}
= & (\exists x', x'_V \bullet (\exists x_V, x'_V \bullet V)) \Rightarrow (\exists x', x'_V \bullet V) & \text{[pred calc]} \\
= & (\forall x', x'_V \bullet \neg (\exists x_V, x'_V \bullet V)) \vee (\exists x', x'_V \bullet V) & \text{[pred calc]} \\
\Rightarrow & (\exists x', x'_V \bullet \neg (\exists x_V, x'_V \bullet V)) \vee (\exists x', x'_V \bullet V) & \text{[pred calc]} \\
= & \exists x', x'_V \bullet \neg (\exists x_V, x'_V \bullet V) \vee V & \text{[pred calc]} \\
= & \mathbf{VH2} \circ \mathbf{VH1}(V) & \text{[def } \mathbf{VH1}, \mathbf{VH2}\text{]}
\end{array}
$$

Proof of Lemma 3.13

Assume that $V_1 = \mathbf{VH}(V_1)$ and $V_2 = \mathbf{VH}(V_2)$. To prove $V_1 \wedge V_2$ is **VH**-healthy, we prove that $V_1 \wedge V_2$ is **VH1**-healthy and **VH2**-healthy separately.

$$
\begin{array}{lll}
\multicolumn{2}{l}{V_1 \wedge V_2} & \\
= & \mathbf{VH1}(V_1) \wedge \mathbf{VH1}(V_2) & \text{[assumption]} \\
= & (\exists x_1, x'_1 \bullet V_1) \Rightarrow V_1) \wedge (\exists x_2, x'_2 \bullet V_2) \Rightarrow V_2) & \text{[def } \mathbf{VH1}\text{]} \\
\Rightarrow & ((\exists x_1, x'_1 \bullet V_1) \wedge (\exists x_2, x'_2 \bullet V_2)) \Rightarrow V_1 \wedge V_2 & \text{[prop calc]} \\
= & (\exists x_1, x'_1, x_2, x'_2 \bullet V_1 \wedge V_2) \Rightarrow V_1 \wedge V_2 & [V_1, V_2 \text{ disjoint}] \\
= & \mathbf{VH1}(V_1 \wedge V_2) & \text{[def } \mathbf{VH1}\text{]}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{V_1 \wedge V_2} & \\
= & \mathbf{VH2}(V_1) \wedge \mathbf{VH2}(V_2) & \text{[assumption]} \\
= & (\exists x', x'_1 \bullet V_1) \wedge (\exists x', x'_2 \bullet V_2) & \text{[def } \mathbf{VH2}\text{]} \\
= & \exists x', x'_1, x'_2 \bullet (\exists x', x'_1 \bullet V_1) \wedge (\exists x', x'_2 \bullet V_2) & [V_1, V_2 \text{ disjoint}] \\
= & \mathbf{VH2}(\mathbf{VH2}(V_1) \wedge \mathbf{VH2}(V_2)) & \text{[def } \mathbf{VH2}\text{]} \\
= & \mathbf{VH2}(V_1 \wedge V_2) & \text{[assumption]}
\end{array}
$$

Proof of Law 3.19

$\mathbf{L}\,(V, P_1 \vee P_2)$

$$
\begin{array}{lll}
= & \exists x, x' \bullet \Delta V \wedge (P_1 \vee P_2) & [\text{def } \mathbf{L}] \\
= & (\exists x, x' \bullet \Delta V \wedge P_1) \vee (\exists x, x' \bullet \Delta V \wedge P_2) & [\text{pred calc}] \\
= & \mathbf{L}\,(V, P_1) \vee \mathbf{L}\,(V, P_2) & [\text{def } \mathbf{L}]
\end{array}
$$

Proof of Law 3.20

$$
\begin{array}{lll}
\multicolumn{2}{l}{P_1 \sqsubseteq P_2} & \\
\Rightarrow & (\Delta V \wedge P_1) \sqsubseteq (\Delta V \wedge P_2) & [\wedge\text{-monotonicity}] \\
\Rightarrow & (\exists x, x' \bullet \Delta V \wedge P_1) \sqsubseteq (\exists x, x' \bullet \Delta V \wedge P_2) & [\exists\text{-monotonicity}] \\
= & \mathbf{L}\,(V, P_1) \sqsubseteq \mathbf{L}\,(V, P_2) & [\text{def } \mathbf{L}]
\end{array}
$$

Proof of Law 3.21

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{L}\,((\bigwedge i : 1..n \bullet V_i), P)} & \\
= & \exists x, x' \bullet \Delta(\bigwedge i : 1..n \bullet V_i) \wedge P & [\text{def } \mathbf{L}] \\
\Rightarrow & \bigwedge i : 1..n \bullet \exists x, x' \bullet \Delta V_i \wedge P & [\text{pred calc}] \\
= & \bigwedge i : 1..n \bullet \mathbf{L}\,(V_i, P) & [\text{def } \mathbf{L}]
\end{array}
$$

Proof of Lemma 3.24

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{G}\,(V, U)} & \\
= & \forall x_V, x'_V \bullet \Delta V \Rightarrow U & [\text{def } \mathbf{G}] \\
= & \neg\ \exists x_V, x'_V \bullet \neg\ (\Delta V \Rightarrow U) & [\text{duality}] \\
= & \neg\ \exists x_V, x'_V \bullet \Delta V \wedge \neg\ U & [\text{prop calc}] \\
= & \exists x_V, x'_V \bullet \Delta V \wedge U & [V \text{ functional } (*)] \\
= & \neg\ \forall x_V, x'_V \bullet \Delta V \Rightarrow \neg\ U & [\text{duality}] \\
= & \neg\ \mathbf{G}\,(V, \neg\ U) & [\text{def } \mathbf{G}]
\end{array}
$$

The step marked $(*)$ is justified as follows. Since V is a total function from behaviours to interactions (i.e. **VH3**-healthy), each behaviour that V maps to a U interaction is *not* a behaviour that V maps to a $\neg\, U$ interaction, and vice versa.

Proof of Theorem 3.25

$$\begin{array}{lll}
& U \sqsubseteq \mathbf{L}\,(V, P) & \\
= & [\,(\exists\, x, x' \bullet \Delta V \wedge P) \Rightarrow U\,] & [\text{def } \sqsubseteq, \mathbf{L}] \\
= & [\,(\forall\, x, x' \bullet \neg\, \Delta V \vee \neg\, P) \vee U\,] & [\text{prop calc}] \\
= & [\,\neg\, \Delta V \vee \neg\, P \vee U\,] & [x, x' \text{ not in } \alpha U] \\
= & \left[\,\neg\, P \vee (\forall\, x_V, x'_V \bullet \neg\, \Delta V \vee U)\,\right] & [x_V, x'_V \text{ not in } \alpha P] \\
= & \left[\,P \Rightarrow (\forall\, x_V, x'_V \bullet \Delta V \Rightarrow U)\,\right] & [\text{prop calc}] \\
= & \mathbf{G}\,(V, U) \sqsubseteq P & [\text{def } \sqsubseteq, \mathbf{G}]
\end{array}$$

Proof of Lemma 3.35

$$\begin{array}{lll}
& \mathbf{L}\,(V, Pre \vdash Post) & \\
= & \exists\, x, x' \bullet \Delta V \wedge (ok \wedge Pre \Rightarrow ok' \wedge Post) & [\text{def } \mathbf{L}, \vdash] \\
= & \exists\, x, x' \bullet \Delta V \wedge (\neg\, ok \vee \neg\, Pre \vee (ok' \wedge Post)) & [\text{prop calc}] \\
= & (\exists\, x, x' \bullet \Delta V \wedge (\neg\, ok \vee \neg\, Pre)) \vee (\exists\, x, x' \bullet \Delta V \wedge ok' \wedge Post) & \\
& & [\text{distributivity}] \\
= & (\neg\, ok_V \vee \exists\, x, x' \bullet \Delta V \wedge \neg\, Pre) \vee (ok'_V \wedge \exists\, x, x' \bullet \Delta V \wedge Post) & \\
& & [V \text{ is } \mathbf{VHD}] \\
= & (\neg\, ok_V \vee \mathbf{L}\,(V, \neg\, Pre)) \vee (ok'_V \wedge \mathbf{L}\,(V, Post)) & [\text{def } \mathbf{L}] \\
= & \neg\,(ok_V \wedge \neg\, \mathbf{L}\,(V, \neg\, Pre)) \vee (ok'_V \wedge \mathbf{L}\,(V, Post)) & [\text{de Morgan}] \\
= & (ok_V \wedge \neg\, \mathbf{L}\,(V, \neg\, Pre)) \Rightarrow (ok'_V \wedge \mathbf{L}\,(V, Post)) & [\text{prop calc}] \\
= & \neg\, \mathbf{L}\,(V, \neg\, Pre) \vdash_V \mathbf{L}\,(V, Post) & [\text{def } \vdash_V]
\end{array}$$

Assuming that *Pre* is a condition over the x variables alone:

$$
\begin{array}{lll}
= & (\neg\ \exists\, x, x' \bullet \Delta V \wedge \neg\, Pre) \vdash_V \mathbf{L}\,(V, Post) & \text{[def } \mathbf{L}\text{]} \\
= & (\forall\, x, x' \bullet \Delta V \Rightarrow Pre) \vdash_V \mathbf{L}\,(V, Post) & \text{[pred calc]} \\
= & (\forall\, x \bullet V \Rightarrow Pre) \vdash_V \mathbf{L}\,(V, Post) & \text{[assumption]}
\end{array}
$$

Proof of Lemma 3.47

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{LR}\,(\mathcal{W}, P)} & \\
= & \exists\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \wedge P & \text{[def } \mathbf{LR}\text{]} \\
= & \exists\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \wedge \mathbf{R}\left(\neg\, P^f_f \vdash P^t_f\right) & \text{[Theorem 2.2]} \\
= & \mathbf{R}_V\left(\exists\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \wedge \left(\neg\, P^f_f \vdash P^t_f\right)\right) & \text{[Definition 3.46]} \\
= & \mathbf{R}_V\left((\forall\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \Rightarrow \neg\, P^f_f) \vdash_V (\exists\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \wedge P^t_f)\right) & \\
 & & \text{[Lemma 3.35]} \\
= & \mathbf{R}_V\left((\forall\, x, x' \bullet \mathcal{R}\,(\mathcal{W}) \Rightarrow \neg\, P^f_f) \vdash_V \mathbf{LR}\left(\mathcal{W}, P^t_f\right)\right) & \text{[def } \mathbf{LR}\text{]}
\end{array}
$$

Proof of Theorem 3.51

For a divergence-free CSP process P, Roscoe (1997) defines the failure set of $\mathsf{Lazy}\,(\mathcal{W}, P)$ in the form:

$$\{(s \upharpoonright \mathcal{W}, r) \mid (s, r \cap \mathcal{W}) \in \mathit{failures}(P)\}$$

The failure set of a reactive design P_R (capturing the semantics of P) is defined by Cavalcanti and Woodcock (2006) as:

$\mathit{failures}(P_R) =$

$$\begin{array}{ll} & \{(tr' - tr, ref') \mid ok \wedge \neg\, wait \wedge P_R \wedge ok'\} \\ \cup & \{((tr' - tr) \frown \langle \checkmark \rangle, ref') \mid ok \wedge \neg\, wait \wedge P_R \wedge ok' \wedge \neg\, wait'\} \\ \cup & \{((tr' - tr) \frown \langle \checkmark \rangle, ref' \cup \{\checkmark\}) \mid ok \wedge \neg\, wait \wedge P_R \wedge ok' \wedge \neg\, wait'\} \end{array}$$

Since our reactive interface allows a user to monitor termination (as signalled by $wait'_V$), we need to extend $\mathcal{W}$ to include the ✓ event. Let:

$$\mathcal{W}^{\checkmark} = \mathcal{W} \cup \{\checkmark\}$$

We now study the $\mathcal{W}^{\checkmark}$-projection of the first set of $failures(P_R)$:

$$\begin{array}{ll} \multicolumn{2}{l}{\{(tr'_V - tr_V, ref'_V) \mid \mathbf{LR}\,(\mathcal{W}^{\checkmark}, ok \wedge \neg\, wait \wedge P_R \wedge ok')\}} \\ = & \{(tr'_V - tr_V, ref'_V) \mid \exists\, x, x' \bullet \mathcal{R}\,(\mathcal{W}^{\checkmark}) \wedge ok \wedge \neg\, wait \wedge P_R \wedge ok'\} \\ = & \{((tr' - tr) \upharpoonright \mathcal{W}, ref' \cap \mathcal{W}) \mid \mathcal{R}\,(\mathcal{W}^{\checkmark}) \wedge ok \wedge \neg\, wait \wedge P_R \wedge ok'\} \end{array}$$

The last proof step is justified because $\mathcal{R}\,(\mathcal{W}^{\checkmark})$ implies:

$$tr'_V - tr_V = (tr' - tr) \upharpoonright \mathcal{W}^{\checkmark} \quad \text{and} \quad ref'_V \cap \mathcal{W}^{\checkmark} \subseteq ref'$$

and refusal sets are downwards-closed.

The projections of the other two sets of $failures(P_R)$ are similar. Hence, the sets $failures(\mathbf{LR}\,(\mathcal{W}, P_R))$ and $failures(\mathsf{Lazy}\,(\mathcal{W}, P))$ are equivalent. We conclude that $\mathsf{Lazy}\,(\mathcal{W}, P)$ yields a process isomorphic to $\mathbf{LR}\,(\mathcal{W}, P_R)$.

Proof of Theorem 4.4

$$\begin{array}{lll} \multicolumn{2}{l}{P_1 \sqsubseteq P_2} & \\ = & (P_1 \sqsubseteq P_2) \wedge (\widetilde{P_1} \sqsubseteq \widetilde{P_2}) & [\text{property of } \sqsubseteq] \\ = & \left[(P_2 \Rightarrow P_1) \wedge (\widetilde{P_2} \Rightarrow \widetilde{P_1})\right] & [\text{def } \sqsubseteq] \\ = & \left[(P_2 \wedge \widetilde{P_2}) \Rightarrow (P_1 \wedge \widetilde{P_1})\right] & [\text{disjoint alphabets}] \\ = & (P_1 \wedge \widetilde{P_1}) \sqsubseteq (P_2 \wedge \widetilde{P_2}) & [\text{def } \sqsubseteq] \\ = & \mathbf{U}\,(P_1) \sqsubseteq \mathbf{U}\,(P_2) & [\text{def } \mathbf{U}] \end{array}$$

Proof of Lemma 4.6

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{D}\,(\mathsf{U}\,(P))} & \\
= & \exists\, \widetilde{x}, \widetilde{x'} \bullet P \wedge \widetilde{P} & [\text{def } \mathsf{U}, \mathsf{D}] \\
= & P \wedge \exists\, \widetilde{x}, \widetilde{x'} \bullet \widetilde{P} & [\widetilde{x}, \widetilde{x'} \text{ free in } P] \\
= & P \wedge \mathbf{true} & [\text{pred calc}] \\
= & P & [\text{unit of conjunction}]
\end{array}$$

Proof of Law 4.9

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{U}\,(P_1 \vee P_2)} & \\
= & (P_1 \vee P_2) \wedge (\widetilde{P_1} \vee \widetilde{P_2}) & [\text{def } \mathsf{U}] \\
\Leftarrow & (P_1 \wedge \widetilde{P_1}) \vee (P_2 \wedge \widetilde{P_2}) & [\text{prop calc}] \\
= & \mathsf{U}\,(P_1) \vee \mathsf{U}\,(P_2) & [\text{def } \mathsf{U}]
\end{array}$$

Proof of Law 4.10

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{D}\,(Q_1) \wedge \mathsf{D}\,(Q_2)} & \\
= & (\exists\, \widetilde{x}, \widetilde{x'} \bullet Q_1) \wedge (\exists\, \widetilde{x}, \widetilde{x'} \bullet Q_2) & [\text{def } \mathsf{D}] \\
\Leftarrow & \exists\, \widetilde{x}, \widetilde{x'} \bullet Q_1 \wedge Q_2 & [\text{prop calc}] \\
= & \mathsf{D}\,(Q_1 \wedge Q_2) & [\text{def } \mathsf{D}]
\end{array}$$

Proof of Law 4.13

$$\begin{array}{lll}
\multicolumn{2}{l}{\neg\, \mathsf{U}\,(P)} & \\
= & \neg\, (P \wedge \widetilde{P}) & [\text{def } \mathsf{U}] \\
= & \neg\, P \vee \neg\, \widetilde{P} & [\text{de Morgan}] \\
\Leftarrow & \neg\, P \wedge \neg\, \widetilde{P} & [\text{prop calc}] \\
= & \mathsf{U}\,(\neg\, P) & [\text{def } \mathsf{U}]
\end{array}$$

Proof of Law 4.14

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{D}\,(\neg\, Q)} & \\
= & \exists\, \widetilde{x}, \widetilde{x'} \bullet \neg\, Q & [\text{def } \mathbf{D}] \\
\Leftarrow & \forall\, \widetilde{x}, \widetilde{x'} \bullet \neg\, Q & [\text{pred calc}] \\
= & \neg\, \exists\, \widetilde{x}, \widetilde{x'} \bullet Q & [\text{pred calc}] \\
= & \neg\, \mathbf{D}\,(Q) & [\text{def } \mathbf{D}]
\end{array}
$$

Proof of Lemma 4.26

$$
\begin{array}{lll}
\multicolumn{2}{l}{\theta_1 \sqsubseteq \theta_2 \wedge P \propto_L \theta_2} & \\
= & \theta_1 \sqsubseteq \theta_2 \wedge \left[P \Rightarrow \exists\, \widetilde{x}, \widetilde{x'} \bullet \mathbf{UI}\,(L, P) \wedge \theta_2 \right] & [\text{def } \propto_L] \\
\Rightarrow & \theta_1 \sqsubseteq \theta_2 \wedge \left[P \Rightarrow \exists\, \widetilde{x}, \widetilde{x'} \bullet \mathbf{UI}\,(L, P) \wedge \theta_1 \right] & [\theta_1 \sqsubseteq \theta_2] \\
= & \theta_1 \sqsubseteq \theta_2 \wedge P \propto_L \theta_1 & [\text{def } \propto_L]
\end{array}
$$

Proof of Lemma 4.40

$$
\begin{array}{lll}
\multicolumn{2}{l}{Q_1 \sqsubseteq Q_2} & \\
\Rightarrow & \mathbf{D}\,(Q_1) \sqsubseteq \mathbf{D}\,(Q_2) & [\text{Law 4.12}] \\
= & \widetilde{\mathbf{D}\,(Q_1)} \sqsubseteq \widetilde{\mathbf{D}\,(Q_2)} & [\text{renaming}] \\
\Rightarrow & Q_1 \wedge \widetilde{\mathbf{D}\,(Q_1)} \sqsubseteq Q_1 \wedge \widetilde{\mathbf{D}\,(Q_2)} & [\text{monotonicity of conjunction}] \\
\Rightarrow & Q_1 \wedge \widetilde{\mathbf{D}\,(Q_1)} \sqsubseteq Q_2 \wedge \widetilde{\mathbf{D}\,(Q_2)} & [\text{because } Q_1 \sqsubseteq Q_2] \\
= & \mathbf{C}\,(Q_1) \sqsubseteq \mathbf{C}\,(Q_2) & [\text{def } \mathbf{C}]
\end{array}
$$

Proof of Lemma 4.43

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{CC}\,(\mathsf{CC}\,(Q))} & \\
= & \mu\, X \bullet \mathsf{C}\,((\mu\, X \bullet \mathsf{C}\,(Q \wedge X)) \wedge X) & [\text{def } \mathsf{CC}] \\
= & \mu\, X \bullet \mathsf{C}\,(Q \wedge X) & [\text{property of } \mu\, X] \\
= & \mathsf{CC}\,(Q) & [\text{def } \mathsf{CC}]
\end{array}$$

Proof of Theorem 4.44

Let $Q = \mathsf{UI}\,(L, P) \wedge \theta$ in:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{D}\,(\mathsf{CC}\,(Q)) \propto_L \theta} & \\
= & [\,\mathsf{D}\,(\mathsf{CC}\,(Q)) \;\Rightarrow\; \mathsf{D}\,(\mathsf{UI}\,(L, \mathsf{D}\,(\mathsf{CC}\,(Q))) \wedge \theta)\,] & [\text{def } \propto_L] \\
= & [\,\mathsf{D}\,(\mathsf{CC}\,(Q)) \;\Rightarrow\; \mathsf{D}\,(\mathsf{UI}\,(L, \mathsf{D}\,(\mathsf{CC}\,(Q))))\,] & [\theta \sqsubseteq \mathsf{CC}\,(Q)] \\
= & [\,\mathsf{D}\,(\mathsf{CC}\,(Q)) \;\Rightarrow\; \mathsf{D}\,(\mathsf{CC}\,(Q))\,] & [\text{Lemma 4.6}] \\
= & \textbf{true} & [\text{tautology}]
\end{array}$$

Proof of Lemma 5.3

This proof relies on two special properties of $\mathcal{I}(\mathcal{L})$. First, $\mathcal{I}(\mathcal{L})$ is tautologous if the fog space is unconstrained:

$$\exists\, \widetilde{x}, \widetilde{x'} \bullet \mathcal{I}(\mathcal{L}) \quad = \quad \textbf{true}$$

Second, renaming the observable variables of $\mathcal{I}(\mathcal{L})$ to the fog variables results in a tautology:

$$\widetilde{\mathcal{I}(\mathcal{L})} \quad = \quad \widetilde{ok} = \widetilde{ok} \wedge \widetilde{ok'} = \widetilde{ok'} \wedge \ldots \quad = \quad \textbf{true}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{P_1 \sqsubseteq P_2} & \\
= & P_1 \wedge \mathcal{I}(\mathcal{L}) \sqsubseteq P_2 \wedge \mathcal{I}(\mathcal{L}) & [\text{property of } \mathcal{I}(\mathcal{L})] \\
= & \mathsf{U}\,(P_1 \wedge \mathcal{I}(\mathcal{L})) \sqsubseteq \mathsf{U}\,(P_2 \wedge \mathcal{I}(\mathcal{L})) & [\text{Theorem 4.4}]
\end{array}$$

$$
\begin{array}{cll}
= & P_1 \wedge \mathcal{I}(\mathcal{L}) \wedge \widetilde{P_1} \wedge \widetilde{\mathcal{I}(\mathcal{L})} \sqsubseteq P_2 \wedge \mathcal{I}(\mathcal{L}) \wedge \widetilde{P_2} \wedge \widetilde{\mathcal{I}(\mathcal{L})} & [\text{def } \mathbf{U}] \\
= & P_1 \wedge \mathcal{I}(\mathcal{L}) \wedge \widetilde{P_1} \sqsubseteq P_2 \wedge \mathcal{I}(\mathcal{L}) \wedge \widetilde{P_2} & [\text{property of } \mathcal{I}(\mathcal{L})] \\
= & \mathbf{UC}(\mathcal{L}, P_1) \sqsubseteq \mathbf{UC}(\mathcal{L}, P_2) & [\text{def } \mathbf{UC}]
\end{array}
$$

Proof of Law 5.4

A special property of $\mathcal{I}(\mathcal{L})$ is that it maps each behaviour ϕ to its fog counterpart $\phi[\widetilde{x}, \widetilde{x'}/x, x']$. Hence:

$$
\begin{array}{cll}
\multicolumn{2}{l}{\mathbf{D}(\mathbf{UC}(\mathcal{L}, A))} & \\
= & \exists \widetilde{x}, \widetilde{x'} \bullet A \wedge \widetilde{A} \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \mathbf{D}, \mathbf{UC}] \\
= & A \wedge \exists \widetilde{x}, \widetilde{x'} \bullet \widetilde{A} \wedge \mathcal{I}(\mathcal{L}) & [\widetilde{x}, \widetilde{x'} \text{ free in } A] \\
= & A \wedge \exists \widetilde{x}, \widetilde{x'} \bullet \widetilde{A} & [\text{property of } \mathcal{I}(\mathcal{L})] \\
= & A & [\text{Lemma 4.6}]
\end{array}
$$

Proof of Lemma 5.6

$$
\begin{array}{cll}
\multicolumn{2}{l}{\mathbf{UC}(\mathcal{L}, A)} & \\
= & A \wedge \widetilde{A} \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \mathbf{UC}] \\
= & \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}(A) \wedge \widetilde{\mathbf{R1}} \circ \widetilde{\mathbf{R2}} \circ \widetilde{\mathbf{R3}}(\widetilde{A}) \wedge \mathcal{I}(\mathcal{L}) & [\text{property of } A] \\
= & \mathbf{R1} \circ \widetilde{\mathbf{R1}} \circ \mathbf{R2} \circ \widetilde{\mathbf{R2}} \circ \widehat{\mathbf{R3}}(A \wedge \widetilde{A}) \wedge \mathcal{I}(\mathcal{L}) & [\text{disjoint alphabets; def } \widehat{\mathbf{R3}}] \\
= & \widehat{\mathbf{R}}(\mathbf{U}(A)) \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \widehat{\mathbf{R}}, \mathbf{U}]
\end{array}
$$

Proof of Theorem 5.7

$$
\begin{array}{cll}
\multicolumn{2}{l}{\widehat{\mathbf{R}}(\mathbf{U}(A)) \wedge \mathcal{I}(\mathcal{L})} & \\
= & \widehat{\mathbf{R}}\left(A \wedge \widetilde{A}\right) \wedge \mathcal{I}(\mathcal{L}) & [\text{definition of } \mathbf{U}] \\
= & \widehat{\mathbf{R}}\left(\mathbf{R}\left(\neg A^f_f \vdash A^t_f\right) \wedge \left(\mathbf{R}\left(\neg A^f_f \vdash A^t_f\right)\right)[\widetilde{x}, \widetilde{x'}/x, x']\right) \wedge \mathcal{I}(\mathcal{L}) & [\text{Theorem 2.2}]
\end{array}
$$

$$= \quad \widehat{\mathbf{R}}\left(\left(\neg A_f^f \vdash A_f^t\right) \wedge \left(\neg A_f^f \vdash A_f^t\right)[\widetilde{x}, \widetilde{x}'/x, x']\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{idempotence of } \widehat{\mathbf{R}}]$$

$$= \quad \widehat{\mathbf{R}}\left(\left(ok \wedge \neg A_f^f \Rightarrow ok' \wedge A_f^t\right) \wedge \left(ok \wedge \neg \widetilde{A_f^f} \Rightarrow ok' \wedge \widetilde{A_f^t}\right)\right) \wedge \mathcal{I}(\mathcal{L})$$

$$[\mathcal{I}(\mathcal{L}) \text{ implies } ok = \widetilde{ok} \wedge ok' = \widetilde{ok'}]$$

$$= \quad \widehat{\mathbf{R}}\left(ok \wedge \neg \left(A_f^f \wedge \widetilde{A_f^f}\right) \Rightarrow ok' \wedge \left(\neg A_f^f \Rightarrow A_f^t\right) \wedge \left(\neg \widetilde{A_f^f} \Rightarrow \widetilde{A_f^t}\right)\right) \wedge \mathcal{I}(\mathcal{L})$$

$$[\text{prop calc}]$$

$$= \quad \widehat{\mathbf{R}}\left(ok \wedge \neg \left(A_f^f \wedge \widetilde{A_f^f}\right) \Rightarrow ok' \wedge A_f^t \wedge \widetilde{A_f^t}\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{property of } A_f^t]$$

$$= \quad \widehat{\mathbf{R}}\left(ok \wedge \neg \mathbf{U}\left(A_f^f\right) \Rightarrow ok' \wedge \mathbf{U}\left(A_f^t\right)\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \mathbf{U}]$$

$$= \quad \widehat{\mathbf{R}}\left(\neg \mathbf{U}\left(A_f^f\right) \vdash \mathbf{U}\left(A_f^t\right)\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \widehat{\mathbf{R}}]$$

Proof of Lemma 5.8

$$\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post))$$

$$= \quad \widehat{\mathbf{R}}\left(\neg \mathbf{U}\,(\mathbf{R}\,(Pre \vdash Post))_{ff}^{ff} \vdash \mathbf{U}\,(\mathbf{R}\,(Pre \vdash Post))_{ff}^{tt}\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{Theorem 5.7}]$$

$$= \quad \widehat{\mathbf{R}}\left(\neg \mathbf{U}\left(\mathbf{R}\,(Pre \vdash Post)_f^f\right) \vdash \mathbf{U}\left(\mathbf{R}\,(Pre \vdash Post)_f^t\right)\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{substitution}]$$

$$= \quad \widehat{\mathbf{R}}\left(\neg \mathbf{U}\left(\neg Pre_f^f\right) \vdash \mathbf{U}\left(Pre_f^t \Rightarrow Post_f^t\right)\right) \wedge \mathcal{I}(\mathcal{L})$$

$$[\text{property of reactive designs}]$$

$$= \quad \widehat{\mathbf{R}}\,(\neg \mathbf{U}\,(\neg Pre) \vdash \mathbf{U}\,(Pre \Rightarrow Post)) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \vdash, \mathbf{R3}; \text{assumption}]$$

$$= \quad \widehat{\mathbf{R}}\left(Pre \vee \widetilde{Pre} \vdash \mathbf{U}\,(Pre \Rightarrow Post)\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \mathbf{U}; \text{prop calc}]$$

Proof of Lemma 5.9

$$\mathbf{UC}\,(\mathcal{L}, w : [Pre, Post])$$

$$= \quad \mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post \wedge \neg wait' \wedge tr' = tr \wedge u' = u)) \quad [\text{def spec stmt}]$$

$$= \quad \widehat{\mathbf{R}}\left(Pre \vee \widetilde{Pre} \vdash \mathbf{U}\,(Pre \Rightarrow (Post \wedge \neg wait' \wedge tr' = tr \wedge u' = u))\right) \wedge \mathcal{I}(\mathcal{L})$$

$$[\text{Lemma 5.8}]$$

Proof of Theorem 5.14

This justification applies to the lifted operators defined in Section 5.3.

Condition 5.11

Condition 5.11 holds for sequential composition:

$$\begin{array}{lll}
& \mathbf{D}\left(\mathbf{UC}\,(\mathcal{L},A_1)\ \widehat{;}\ \mathbf{UC}\,(\mathcal{L},A_2)\right) & \\
= & \exists\,\widetilde{x},\widetilde{x'} \bullet \exists\,x_0,\widetilde{x_0} \bullet \left(\begin{array}{ll} & (A_1 \wedge \widetilde{A_1} \wedge \mathcal{I}(\mathcal{L}))[x_0,\widetilde{x_0}/x',\widetilde{x'}] \\ \wedge & (A_2 \wedge \widetilde{A_2} \wedge \mathcal{I}(\mathcal{L}))[x_0,\widetilde{x_0}/x,\widetilde{x}] \end{array}\right) & [\text{def } \mathbf{D},\ \mathbf{UC}] \\
= & \exists\,x_0 \bullet \left(\begin{array}{ll} & A_1[x_0/x'] \wedge A_2[x_0/x] \\ \wedge & \exists\,\widetilde{x},\widetilde{x_0},\widetilde{x'} \bullet \left(\begin{array}{ll} & (\widetilde{A_1} \wedge \mathcal{I}(\mathcal{L}))[x_0,\widetilde{x_0}/x',\widetilde{x'}] \\ \wedge & (\widetilde{A_2} \wedge \mathcal{I}(\mathcal{L}))[x_0,\widetilde{x_0}/x,\widetilde{x}] \end{array}\right) \end{array}\right) & \\
& & [\widetilde{x},\widetilde{x_0},\widetilde{x'} \text{ free in } A_1,A_2] \\
= & \exists\,x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x] & [\text{property of } \mathcal{I}(\mathcal{L})] \\
& A_1\,;A_2 & [\text{def } ;]
\end{array}$$

Condition 5.11 holds for internal choice:

$$\begin{array}{lll}
& \mathbf{D}\left(\mathbf{UC}\,(\mathcal{L},A_1)\ \widehat{\sqcap}\,\mathbf{UC}\,(\mathcal{L},A_2)\right) & \\
= & \mathbf{D}\,(\mathbf{UC}\,(\mathcal{L},A_1)) \sqcap \mathbf{D}\,(\mathbf{UC}\,(\mathcal{L},A_2)) & [\text{Law 4.8}] \\
= & A_1 \sqcap A_2 & [\text{Law 5.4}]
\end{array}$$

We appeal to Lemma 5.22 to treat the lifted prefixing operator in terms of lifted sequential composition:

$$\begin{array}{lll}
& \mathbf{D}\left(c.e\ \widehat{\rightarrow}\,\mathbf{UC}\,(\mathcal{L},A)\right) & \\
= & \mathbf{D}\,(\mathbf{UC}\,(\mathcal{L},c.e \rightarrow \mathit{Skip})\ ;\mathbf{UC}\,(\mathcal{L},A)) & [\text{Lemma 5.22}]
\end{array}$$

$$
\begin{array}{lll}
= & c.e \to Skip\,;A & \text{[as above]} \\
= & c.e \to A &
\end{array}
$$

so Condition 5.11 holds for prefixing.

For each lifted operator defined as a lifted reactive design, we have:

$$\mathbf{D}\left(\widehat{\mathbf{R}}\,(Pre \vdash Post)\right) \quad = \quad \mathbf{R}\,(\neg\,\mathbf{D}\,(\neg\,Pre) \vdash \mathbf{D}\,(Post))$$

By distributing **D** through the (lifted) *Pre* and *Post* terms, we derive the corresponding reactive design structure of the counterpart *Circus* operator, with each operand wrapped by **D**.

Taking the lifted guard as an example, we have:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{D}\left(g \mathbin{\widehat{\&}} \mathbf{UC}\,(\mathcal{L},A)\right)} \\
= & \mathbf{R}\left(\begin{array}{l} \neg\,\mathbf{D}\left(\neg\,(\mathbf{U}\,(g) \Rightarrow \neg\,\mathbf{UC}\,(\mathcal{L},A)^{ff}_{ff})\right) \\ \vdash\ \mathbf{D}\left(\begin{array}{l} \mathbf{U}\,(g) \wedge \mathbf{UC}\,(\mathcal{L},A)^{tt}_{ff} \\ \vee\,\mathbf{U}\,(\neg\,g \wedge tr' = tr \wedge wait') \end{array}\right) \end{array}\right) & \text{[property above]} \\
= & \mathbf{R}\left(\neg\,\mathbf{D}\left(\mathbf{U}\,(g) \wedge \mathbf{UC}\,(\mathcal{L},A)^{ff}_{ff}\right) \vdash \left(\begin{array}{l} \mathbf{D}\left(\mathbf{U}\,(g) \wedge \mathbf{UC}\,(\mathcal{L},A)^{tt}_{ff}\right) \\ \vee\,\mathbf{D}\,(\mathbf{U}\,(\neg\,g \wedge tr' = tr \wedge wait')) \end{array}\right)\right) & \\
 & & \text{[prop calc; Law 4.8]} \\
= & \mathbf{R}\left(\neg\,\left(g \wedge \mathbf{D}\left(\mathbf{UC}\,(\mathcal{L},A)^{ff}_{ff}\right)\right) \vdash \left(\begin{array}{l} g \wedge \mathbf{D}\left(\mathbf{UC}\,(\mathcal{L},A)^{tt}_{ff}\right) \\ \vee\,\neg\,g \wedge tr' = tr \wedge wait' \end{array}\right)\right) & \\
 & & [\widetilde{x},\widetilde{x'}\ \text{free}] \\
= & \mathbf{R}\left(\left(g \Rightarrow \neg\,A^f_f\right) \vdash \left((g \wedge A^t_f) \vee (\neg\,g \wedge tr' = tr \wedge wait')\right)\right) & \text{[Law 5.4]} \\
= & g \mathbin{\&} A & [\text{def } \&\,]
\end{array}
$$

Condition 5.12

For each lifted operator $\widehat{\oplus}$, we provide a lemma to show that:

$$\mathbf{UC}\left(\mathcal{L}, \bigoplus i \bullet A_i\right) \quad \sqsubseteq \quad \widehat{\bigoplus} i \bullet \mathbf{UC}\,(\mathcal{L}, A_i)$$

or define $\widehat{\oplus}$ in terms of other lifted operators where that lemma holds. Hence, Condition 5.12 is satisfied by each lifted operator.

Condition 5.13

It is given that:

- conjunction and disjunction are monotonic;
- negation is antimonotonic;
- implication is antimonotonic in the antecedent and monotonic in the consequent.

Moreover, for each lifted operator $\widehat{\oplus}$, it is the case that:

- each operand of $\widehat{\oplus}$ within its precondition is negated; and
- each operand of $\widehat{\oplus}$ within its postcondition is not negated.

It follows from the structure of each lifted operator that refining each operand of a lifted construct yields a refinement of the whole construct. Hence, Condition 5.13 is satisfied by each lifted operator.

Proof of Lemma 5.15

$$\begin{array}{ll}
 & \mathbf{UC}\,(\mathcal{L}, A_1 \,;A_2) \\
= & \left(\begin{array}{ll} & \exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x] \\ \wedge & (\exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x])\, [\widetilde{x}, \widetilde{x'}/x, x'] \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \mathbf{UC},\ ;] \\
= & \left(\begin{array}{ll} & \exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x] \\ \wedge & \exists\, \widetilde{x_0} \bullet A_1[\widetilde{x}, \widetilde{x_0}/x, x'] \wedge A_2[\widetilde{x_0}, \widetilde{x'}/x, x'] \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{pred calc}]
\end{array}$$

$$
\begin{array}{lll}
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{rl} & A_1[x_0/x'] \wedge A_1[\widetilde{x}, \widetilde{x_0}/x, x'] \\ \wedge & A_2[x_0/x] \wedge A_2[\widetilde{x_0}, \widetilde{x'}/x, x'] \end{array}\right) \wedge \mathcal{I}(\mathcal{L}) & \text{[pred calc]} \\
\Leftarrow & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{rl} & A_1[x_0/x'] \wedge A_1[\widetilde{x}, \widetilde{x_0}/x, x'] \wedge \mathcal{I}(\mathcal{L})[x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & A_2[x_0/x] \wedge A_2[\widetilde{x_0}, \widetilde{x'}/x, x'] \wedge \mathcal{I}(\mathcal{L})[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right) & \\
 & & \text{[property of } \mathcal{I}(\mathcal{L})] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \mathbf{UC}\,(\mathcal{L}, A_1)\,[x_0, \widetilde{x_0}/x', \widetilde{x'}] \wedge \mathbf{UC}\,(\mathcal{L}, A_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] & \text{[def } \mathbf{UC}]
\end{array}
$$

Proof of Law 5.17

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{U}\,(A_1 \,;A_2)} & \\
= & \left(\begin{array}{rl} & \exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x] \\ \wedge & (\exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x])[\widetilde{x}, \widetilde{x'}/x, x'] \end{array}\right) & \text{[def } \mathbf{U},\ ;] \\
= & \left(\begin{array}{rl} & \exists\, x_0 \bullet A_1[x_0/x'] \wedge A_2[x_0/x] \\ \wedge & \exists\, \widetilde{x_0} \bullet \widetilde{A_1}[\widetilde{x_0}/\widetilde{x'}] \wedge \widetilde{A_2}[\widetilde{x_0}/\widetilde{x}] \end{array}\right) & \text{[renaming]} \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{rl} & (A_1 \wedge \widetilde{A_1})[x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & (A_2 \wedge \widetilde{A_2})[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right) & \text{[disjoint alphabets]} \\
= & \mathbf{U}\,(A_1)\ \widehat{;}\ \mathbf{U}\,(A_2) & \text{[def } \mathbf{U},\ \widehat{;}]
\end{array}
$$

Proof of Lemma 5.18

Let A be a *Circus* action that terminates without changing the trace:

$$A \quad = \quad A \wedge (ok \wedge \neg\, wait \Rightarrow ok' \wedge \neg\, wait' \wedge tr' = tr)$$

Assuming this property holds, we have:

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{U}\,(A \wedge ok \wedge \neg\, wait)} & \\
= & \mathbf{U}\,(A) \wedge \mathbf{U}\,(ok \wedge \neg\, wait \wedge ok' \wedge \neg\, wait' \wedge tr' = tr) & \text{[assumption]} \\
= & \mathbf{U}\,(A) \wedge ok \wedge \neg\, wait \wedge ok' \wedge \neg\, wait' \wedge tr' = tr \wedge \mathcal{I}(\mathcal{L}) & \text{[def } \mathcal{I}(\mathcal{L})] \\
= & \mathbf{UC}\,(\mathcal{L}, A) \wedge ok \wedge \neg\, wait \wedge ok' \wedge \neg\, wait' \wedge tr' = tr & \text{[def } \mathbf{UC}]
\end{array}
$$

$$= \quad \mathbf{UC}(\mathcal{L}, A) \wedge ok \wedge \neg\, wait \qquad [\text{assumption}]$$

Now, consider the case where A_1 terminates immediately leaving the trace unchanged:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{UC}(\mathcal{L}, A_1 \,;A_2)} & \\
= & \mathbf{U}(A_1 \,;A_2) \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \mathbf{UC}] \\
= & \left(\mathbf{U}(A_1) \,\widehat{;}\, \mathbf{U}(A_2)\right) \wedge \mathcal{I}(\mathcal{L}) & [\text{Law 5.17}] \\
= & \left(\mathbf{UC}(\mathcal{L}, A_1) \,\widehat{;}\, \mathbf{U}(A_2)\right) \wedge \mathcal{I}(\mathcal{L}) & [A_1 \text{ terminates immediately}] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \mathbf{UC}(\mathcal{L}, A_1)\,[x_0, \widetilde{x_0}/x', \widetilde{x'}] \wedge \mathbf{U}(A_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \widehat{;}\,] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & \mathbf{UC}(\mathcal{L}, A_1)\,[x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & \mathbf{U}(A_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \\ \wedge & \mathcal{I}(\mathcal{L})[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right) & [A_1 \text{ terminates immediately } (*)] \\
= & \mathbf{UC}(\mathcal{L}, A_1) \,\widehat{;}\, (\mathbf{U}(A_2) \wedge \mathcal{I}(\mathcal{L})) & [\text{def } \widehat{;}\,] \\
= & \mathbf{UC}(\mathcal{L}, A_1) \,\widehat{;}\, \mathbf{UC}(\mathcal{L}, A_2) & [\text{def } \mathbf{UC}]
\end{array}$$

The step marked $(*)$ is justified because, since A_1 terminates immediately, it ensures $ok' = ok'$, $wait' = wait$ and $tr' = tr$, and the value of ref' is irrelevant upon termination of a *Circus* action.

The proof for the case where A_2 terminates immediately is similar.

Proof of Lemma 5.19

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{UC}(\mathcal{L}, A_1 \sqcap A_2)} & \\
= & (A_1 \vee A_2) \wedge (\widetilde{A_1} \vee \widetilde{A_2}) \wedge \mathcal{I}(\mathcal{L}) & [\text{def } \mathbf{UC}] \\
\Leftarrow & (A_1 \wedge \widetilde{A_1} \wedge \mathcal{I}(\mathcal{L})) \vee (A_2 \wedge \widetilde{A_2} \wedge \mathcal{I}(\mathcal{L})) & [\text{prop calc}] \\
= & \mathbf{UC}(\mathcal{L}, A_1) \vee \mathbf{UC}(\mathcal{L}, A_2) & [\text{def } \mathbf{UC}]
\end{array}$$

Proof of Lemma 5.22

$$
\begin{array}{ll}
& \mathbf{UC}\,(\mathcal{L}, c.e \to Skip) \\
= & \widehat{\mathbf{R}}\left(\mathbf{U}\,(c.e \to Skip)^{ff}_{ff} \vdash \mathbf{U}\,(c.e \to Skip)^{tt}_{ff}\right) \land \mathcal{I}(\mathcal{L}) \qquad \text{[Theorem 5.7]} \\
= & \widehat{\mathbf{R}}\left(\mathbf{true} \vdash \mathbf{U}\left(v' = v \land \left(\begin{array}{c} tr' = tr \land c.e \notin ref' \\ \lhd\ wait'\ \rhd \\ tr' = tr \frown \langle (c,e) \rangle \end{array}\right)\right)\right) \land \mathcal{I}(\mathcal{L}) \quad \text{[def} \to\text{]} \\
= & \widehat{\mathbf{R}}\left(\mathbf{true} \vdash \mathbf{U}\,(v' = v) \land \left(\begin{array}{c} \mathbf{U}\,(tr' = tr \land c.e \notin ref') \\ \lhd\ wait'\ \rhd \\ \mathbf{U}\,(tr' = tr \frown \langle (c,e) \rangle) \end{array}\right)\right) \land \mathcal{I}(\mathcal{L}) \\
& \qquad \text{[Law 4.7; } \mathcal{I}(\mathcal{L}) \text{ implies } wait = \widetilde{wait}\text{]}
\end{array}
$$

Proof of Lemma 5.24

$$
\begin{array}{ll}
& \mathbf{UC}\,(\mathcal{L}, g \,\&\, A) \\
= & \widehat{\mathbf{R}}\left(\neg\,\mathbf{U}\,(g \,\&\, A)^{ff}_{ff} \vdash \mathbf{U}\,(g \,\&\, A)^{tt}_{ff}\right) \land \mathcal{I}(\mathcal{L}) \qquad \text{[Theorem 5.7]} \\
= & \widehat{\mathbf{R}}\left(\neg\,\mathbf{U}\left(\neg\,(g \Rightarrow \neg\,A^f_f)\right) \vdash \mathbf{U}\left(A^t_f \lhd g \rhd tr' = tr \land wait'\right)\right) \land \mathcal{I}(\mathcal{L}) \\
& \qquad \text{[def } \& \text{]} \\
= & \widehat{\mathbf{R}}\left(\left(\mathbf{U}\,(g) \Rightarrow \neg\,\mathbf{U}\,(A)^{ff}_{ff}\right) \vdash \mathbf{U}\left(A^t_f \lhd g \rhd tr' = tr \land wait'\right)\right) \land \mathcal{I}(\mathcal{L}) \\
& \qquad \text{[def } \mathbf{U}\text{; prop calc]} \\
\Leftarrow & \widehat{\mathbf{R}}\left(\left(\mathbf{U}\,(g) \Rightarrow \neg\,\mathbf{U}\,(A)^{ff}_{ff}\right) \vdash \left(\begin{array}{l} \mathbf{U}\,(g) \land \mathbf{U}\,(A)^{tt}_{ff} \\ \lor\,\mathbf{U}\,(\neg\,g \land tr' = tr \land wait') \end{array}\right)\right) \land \mathcal{I}(\mathcal{L}) \\
& \qquad \text{[strengthen postcondition (Law 4.11)]}
\end{array}
$$

Proof of Lemma 5.26

$$
\begin{array}{ll}
& \mathbf{UC}\,(\mathcal{L}, A_1 \,\Box\, A_2) \\
= & \widehat{\mathbf{R}}\left(\neg\,\mathbf{U}\,(A_1 \,\Box\, A_2)^{ff}_{ff} \vdash \mathbf{U}\,(A_1 \,\Box\, A_2)^{tt}_{ff}\right) \land \mathcal{I}(\mathcal{L}) \qquad \text{[Theorem 5.7]}
\end{array}
$$

$$= \quad \widehat{\mathbf{R}}\left(\neg\,\mathbf{U}\left({A_1}_f^f \vee {A_2}_f^f\right) \vdash \mathbf{U}\left(\begin{array}{c} {A_1}_f^t \wedge {A_2}_f^t \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ {A_1}_f^t \vee {A_2}_f^t \end{array}\right)\right) \wedge \mathcal{I}(\mathcal{L}) \quad [\text{def } \Box]$$

$$\Leftarrow \quad \widehat{\mathbf{R}}\left(\neg\,\mathbf{U}\left({A_1}_f^f\right) \wedge \neg\,\mathbf{U}\left({A_2}_f^f\right) \vdash \mathbf{U}\left(\begin{array}{c} {A_1}_f^t \wedge {A_2}_f^t \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ {A_1}_f^t \vee {A_2}_f^t \end{array}\right)\right) \wedge \mathcal{I}(\mathcal{L})$$

[weaken precondition (contrapositive of Law 4.9)]

$$\Leftarrow \quad \widehat{\mathbf{R}}\left(\begin{array}{l} \neg\,\mathbf{U}\left({A_1}_f^f\right) \wedge \neg\,\mathbf{U}\left({A_2}_f^f\right) \\ \vdash \left(\begin{array}{ll} & \mathbf{U}\,(tr' = tr \wedge wait') \wedge \mathbf{U}\,(A_1)_{ff}^{tt} \wedge \mathbf{U}\,(A_2)_{ff}^{tt} \\ \vee & \mathbf{U}\,(\neg\,(tr' = tr \wedge wait')) \wedge \left(\mathbf{U}\,(A_1)_{ff}^{tt} \vee \mathbf{U}\,(A_2)_{ff}^{tt}\right) \end{array}\right) \end{array}\right)$$

$$\wedge\, \mathcal{I}(\mathcal{L}) \qquad [\text{strengthen postcondition (Law 4.11; Law 4.9)}]$$

Proof of Lemma 5.28

Let EQ denote $1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs$ in:

$$\mathbf{U}\left(\neg\,(A_1 \,\|[cs]\|\, A_2)_f^f\right)$$

$$= \quad \mathbf{U}\left(\begin{array}{ll} & \neg\,\exists\, 1.tr, 2.tr \bullet ({A_1}_f^f ; 1.tr' = tr) \wedge ({A_2}_f ; 2.tr' = tr) \wedge EQ \\ \wedge & \neg\,\exists\, 1.tr, 2.tr \bullet ({A_1}_f ; 1.tr' = tr) \wedge ({A_2}_f^f ; 2.tr' = tr) \wedge EQ \end{array}\right)$$

[def $\|$]

$$\Rightarrow \quad \left(\begin{array}{ll} & \neg\,\exists\, 1.tr, 2.tr \bullet \mathbf{U}\left(({A_1}_f^f ; 1.tr' = tr) \wedge ({A_2}_f ; 2.tr' = tr) \wedge EQ\right) \\ \wedge & \neg\,\exists\, 1.tr, 2.tr \bullet \mathbf{U}\left(({A_1}_f ; 1.tr' = tr) \wedge ({A_2}_f^f ; 2.tr' = tr) \wedge EQ\right) \end{array}\right)$$

[Law 4.7, Law 4.13]

$$= \quad \left(\begin{array}{ll} & \neg\,\exists\, 1.tr, 2.tr \bullet \left(\begin{array}{ll} & \mathbf{U}\,(A_1)_{ff}^{ff} \,; \mathbf{U}\,(1.tr' = tr) \\ \wedge & \mathbf{U}\,(A_2)_{ff} \,; \mathbf{U}\,(2.tr' = tr) \end{array}\right) \wedge \mathbf{U}\,(EQ) \\ \wedge & \neg\,\exists\, 1.tr, 2.tr \bullet \left(\begin{array}{ll} & \mathbf{U}\,(A_1)_{ff} \,; \mathbf{U}\,(1.tr' = tr) \\ \wedge & \mathbf{U}\,(A_2)_{ff}^{ff} \,; \mathbf{U}\,(2.tr' = tr) \end{array}\right) \wedge \mathbf{U}\,(EQ) \end{array}\right)$$

[Law 4.7, Law 5.17]

Proof of Lemma 5.30

Let HS denote $\left(\begin{array}{ll} & A[s, (cs \cup ref')/tr', ref'] \\ \wedge & (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs) \end{array}\right)$ in:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{UC}\,(\mathcal{L}, A \setminus cs)} & \\
= & \mathbf{UC}\,(\mathbf{R}\,(\exists\, s \bullet HS)\ ;Skip) & [\text{def } \setminus] \\
= & \mathbf{UC}\,(\mathbf{R}\,(\exists\, s \bullet HS))\ ;\mathbf{UC}\,(\mathcal{L}, Skip) & [\text{Lemma 5.18}] \\
= & \big(\widehat{\mathbf{R}}\,(\mathbf{U}\,(\exists\, s \bullet HS)) \wedge \mathcal{I}(\mathcal{L})\big)\ ;\mathbf{UC}\,(\mathcal{L}, Skip) & [\text{Lemma 5.6}] \\
= & \big(\widehat{\mathbf{R}}\,(\exists\, s, \widetilde{s} \bullet \mathbf{U}\,(HS)) \wedge \mathcal{I}(\mathcal{L})\big)\ ;\mathbf{UC}\,(\mathcal{L}, Skip) & [\text{property of } \mathbf{U}]
\end{array}$$

Proof of Theorem 5.40

$$\begin{array}{lll}
\multicolumn{2}{l}{\theta_1 \sqsubseteq \theta_2} & \\
= & (ok \wedge \neg\, wait \Rightarrow \theta_1) \sqsubseteq (ok \wedge \neg\, wait \Rightarrow \theta_2) & [\text{pred calc } (*)] \\
= & \mathsf{Conf}\,(\theta_1) \sqsubseteq \mathsf{Conf}\,(\theta_2) & [\text{def } \mathsf{Conf}] \\
= & \mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_1) \sqsubseteq \mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_2) & [\text{pred calc } (*)] \\
= & \langle\, \theta_1 \,\rangle \sqsubseteq \langle\, \theta_2 \,\rangle & [\text{def CA}]
\end{array}$$

It is assumed that obligations reference only the unprimed state variables $v, \widetilde{v}$. The steps marked $(*)$ are equivalences (rather than refinements), because they place no constraints on the unprimed state variables.

Proof of Law 5.41

We prove the result for two CAs. Since relational composition is associative, this result generalises to arbitrary sequences of CAs.

$$\begin{array}{lll}
\multicolumn{2}{l}{\langle\, \theta_1 \,\rangle\,;\langle\, \theta_2 \,\rangle} & \\
= & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_1))\ \widehat{;}\,(\mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_2)) & [\text{def CA}] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_1))[x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge \mathsf{Conf}\,(\theta_2))[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right) & [\text{def } \widehat{;}]
\end{array}$$

$$
\begin{array}{lll}
= & \mathsf{Conf}\,(\theta_1) \wedge \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & \mathbf{UC}\,(\mathcal{L}, \mathit{Skip})\,[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & (\mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_2))[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array} \right) & \\
& & [x', \widetilde{x'} \text{ free in } \mathsf{Conf}\,(\theta_1)] \\
= & \mathsf{Conf}\,(\theta_1) \wedge \Big(\mathbf{UC}\,(\mathcal{L}, \mathit{Skip})\, \widehat{;}\, (\mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_2)) \Big) & [\text{def } \widehat{;}\,] \\
= & \mathsf{Conf}\,(\theta_1) \wedge \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_2) & [\mathit{Skip} \text{ left unit}] \\
= & \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_1 \wedge \theta_2) & [\text{def } \mathsf{Conf}] \\
= & \langle \bigsqcup \{\theta_1, \theta_2\} \rangle & [\text{def CA}]
\end{array}
$$

Proof of Law 5.43

$$
\begin{array}{lll}
\multicolumn{3}{l}{\langle \theta_1 \rangle \sqcap \cdots \sqcap \langle \theta_n \rangle} \\
= & (\mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_1)) \vee \cdots \vee (\mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_n)) & [\text{def CA}, \sqcap] \\
= & \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge (\mathsf{Conf}\,(\theta_1) \vee \cdots \vee \mathsf{Conf}\,(\theta_n)) & [\text{prop calc}] \\
= & \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\theta_1 \vee \cdots \vee \theta_n) & [\text{def } \mathsf{Conf}] \\
= & \langle \bigsqcap \{\theta_1, \ldots, \theta_n\} \rangle & [\text{def CA}]
\end{array}
$$

Proof of Law 5.44

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{UC}\,(\mathcal{L}, [\,C\,])} \\
= & \widehat{\mathbf{R}}\,(\mathbf{true} \vdash \mathbf{U}\,(C \wedge \neg\, \mathit{wait}' \wedge \mathit{tr}' = \mathit{tr} \wedge v' = v)) \wedge \mathcal{I}(\mathcal{L}) & [\text{Lemma 5.9}] \\
= & \widehat{\mathbf{R}}\,(\mathbf{true} \vdash \mathbf{U}\,(\neg\, \mathit{wait}' \wedge \mathit{tr}' = \mathit{tr} \wedge v' = v)) \wedge \mathcal{I}(\mathcal{L}) \wedge \mathsf{Conf}\,(\mathbf{U}\,(C)) & \\
& & [\text{prop calc; def } \mathsf{Conf}] \\
= & \mathbf{UC}\,(\mathcal{L}, \mathit{Skip}) \wedge \mathsf{Conf}\,(\mathbf{U}\,(C)) & [\text{def } \mathit{Skip}] \\
= & \langle \mathbf{U}\,(C) \rangle & [\text{def CA}]
\end{array}
$$

Proof of Law 5.45

$\mathbf{D}\,(\langle \theta \rangle)$

$$
\begin{array}{lll}
= & \mathbf{D}\,(\mathbf{UC}\,(\mathcal{L}, Skip) \wedge (ok \wedge \neg\, wait \Rightarrow \theta)) & [\text{Law 5.44}] \\
= & \mathbf{D}\left(\widehat{\mathbf{R}}\,(\mathbf{true} \vdash \theta \wedge \mathbf{U}\,(\neg\, wait' \wedge tr' = tr \wedge v' = v)) \wedge \mathcal{I}(\mathcal{L})\right) & \\
 & & [\text{Lemma 5.8, def } Skip] \\
= & \mathbf{R}\,(\mathbf{true} \vdash \mathbf{D}\,(\theta) \wedge \neg\, wait' \wedge tr' = tr \wedge v' = v) & [\text{property of } \mathbf{D}] \\
= & [\, \mathbf{D}\,(\theta)\,] & [\text{def coercion}]
\end{array}
$$

Proof of Law 5.46

$$
\begin{array}{lll}
 & \langle\, \theta \,\rangle\,;\,\langle\, A \,\rangle & \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge (ok \wedge \neg\, wait \Rightarrow \theta))[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & \mathbf{UC}\,(\mathcal{L}, A)\,[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array}\right) & \\
 & & [\text{def } \widehat{;}, \text{CA}] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta'))[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & \mathbf{UC}\,(\mathcal{L}, A)\,[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array}\right) & \\
 & & [\text{property of } Skip] \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & \mathbf{UC}\,(\mathcal{L}, Skip)\,[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & (\mathbf{UC}\,(\mathcal{L}, A) \wedge (ok \wedge \neg\, wait \Rightarrow \theta))[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array}\right) & \\
 & & [\text{renaming}] \\
= & \mathbf{UC}\,(\mathcal{L}, Skip)\,\widehat{;}\,(\mathbf{UC}\,(\mathcal{L}, A) \wedge (ok \wedge \neg\, wait \Rightarrow \theta)) & [\text{def } \widehat{;}] \\
= & (\mathbf{UC}\,(\mathcal{L}, A) \wedge (ok \wedge \neg\, wait \Rightarrow \theta)) & [Skip \text{ left unit}] \\
= & \langle\, A \,\rangle \wedge \mathsf{Conf}\,(\theta) & [\text{def } \mathsf{Conf}]
\end{array}
$$

Proof of Law 5.47

$$
\begin{array}{lll}
 & \langle\, A \,\rangle\,;\langle\, \theta \,\rangle & \\
= & \exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & \mathbf{UC}\,(\mathcal{L}, A)\,[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & (\mathbf{UC}\,(\mathcal{L}, Skip) \wedge (ok \wedge \neg\, wait \Rightarrow \theta))[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array}\right) & \\
 & & [\text{def } \widehat{;}, \text{CA}]
\end{array}
$$

$$= \quad \exists x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & (\textsf{\textbf{UC}}\,(\mathcal{L}, A) \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta'))[x_0, \widetilde{x_0} / x', \widetilde{x'}] \\ \wedge & \textsf{\textbf{UC}}\,(\mathcal{L}, Skip)\,[x_0, \widetilde{x_0} / x, \widetilde{x}] \end{array} \right)$$

[renaming]

$$= \quad (\textsf{\textbf{UC}}\,(\mathcal{L}, A) \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta'))\ \widehat{;}\ \textsf{\textbf{UC}}\,(\mathcal{L}, Skip) \qquad [\text{def } \widehat{;}]$$

$$= \quad \textsf{\textbf{UC}}\,(\mathcal{L}, A) \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta') \qquad [\textit{Skip} \text{ right unit}]$$

$$= \quad \langle\, A \,\rangle \wedge \textsf{Conf}'\,(\theta) \qquad [\text{def } \textsf{Conf}']$$

Proof of Law 6.4

Let $C1 = c_1 \rightarrow A_1$ and $C2 = c_2 \rightarrow A_2$ in:

$$\langle\, L\,:\, C1 \sqcap C2 \,\rangle$$

$$= \quad \textsf{\textbf{UC}}\,(\mathcal{L}, C1 \vee C2) \qquad [\text{def } \langle\,\cdot\,\rangle, \sqcap]$$

$$= \quad (C1 \vee C2) \wedge (\widetilde{C1} \wedge \widetilde{C2}) \wedge \mathcal{I}(\mathcal{L}) \qquad [\text{def } \textsf{\textbf{UC}}]$$

$$= \quad ((C1 \wedge \widetilde{C1}) \vee (C2 \wedge \widetilde{C2})) \wedge \mathcal{I}(\mathcal{L}) \qquad [\text{property of } \mathcal{I}(\mathcal{L})]$$

$$= \quad \textsf{\textbf{UC}}\,(\mathcal{L}, C1) \vee \textsf{\textbf{UC}}\,(\mathcal{L}, C2) \qquad [\text{def } \textsf{\textbf{UC}}]$$

$$= \quad \langle\, L\,:\, C1 \,\rangle \sqcap \langle\, L\,:\, C2 \,\rangle \qquad [\text{def } \langle\,\cdot\,\rangle, \sqcap]$$

Proof of Theorem 6.12

$$\textsf{\textbf{bwQ}}\,(\textsf{\textbf{UC}}\,(\mathcal{L}, \textsf{\textbf{R}}\,(Pre \vdash Post))\,, \theta)$$

$$= \quad \forall\, x' \bullet \left(\begin{array}{l} \textsf{\textbf{D}}\,(\textsf{\textbf{UC}}\,(\mathcal{L}, \textsf{\textbf{R}}\,(Pre \vdash Post)))! \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \textsf{\textbf{UC}}\,(\mathcal{L}, \textsf{\textbf{R}}\,(Pre \vdash Post))! \wedge \textsf{Conf}'\,(\theta) \end{array} \right)$$

$$= \quad \forall\, x' \bullet \left(\begin{array}{l} \textsf{\textbf{R}}\,(Pre \vdash Post)! \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \textsf{\textbf{UC}}\,(\mathcal{L}, \textsf{\textbf{R}}\,(Pre \vdash Post))! \wedge \textsf{Conf}'\,(\theta) \end{array} \right) \qquad [\text{Law 5.4}]$$

$$= \quad \forall\, x' \bullet \left(\begin{array}{l} \textsf{\textbf{R}}\,(Pre \vdash Post)! \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \left(\begin{array}{ll} & \widehat{\textsf{\textbf{R}}}\left(Pre \vee \widetilde{Post} \vdash \textsf{\textbf{U}}\,(Pre \Rightarrow Post)\right)! \\ \wedge & \mathcal{I}(\mathcal{L})! \wedge \textsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right)$$

[Lemma 5.8]

$$
= \quad \forall x' \bullet \left(\begin{array}{l} (Pre! \Rightarrow ok' \land Post!) \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet \left(\begin{array}{ll} & ((Pre! \lor \widetilde{Pre}!) \Rightarrow ok' \land \mathbf{U}\,(Pre! \Rightarrow Post!)) \\ \land & tr \leq tr' \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right)
$$

[!-apply]

$$
= \quad \begin{array}{ll} & \forall x' \bullet Pre! \Rightarrow \left(\begin{array}{l} ok' \land Post! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet \left(\begin{array}{ll} & \widetilde{Pre}! \Rightarrow \widetilde{Post!} \\ \land & \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right) \\ \land & \forall x' \bullet \neg\, Pre! \Rightarrow \\ & \quad \left(tr \leq tr' \Rightarrow \exists \widetilde{x'} \bullet \left(\begin{array}{ll} & \widetilde{Pre}! \Rightarrow ok' \land \widetilde{Post}! \\ \land & \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \right) \end{array}
$$

[case split]

$$
= \quad \begin{array}{ll} & \forall x' \bullet \left(\begin{array}{l} Pre! \land ok' \land Post! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet (\widetilde{Pre}! \Rightarrow \widetilde{Post!}) \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \\ \land & \forall x' \bullet \left(\begin{array}{l} \neg\, Pre! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet \left(\begin{array}{ll} & \widetilde{Pre}! \Rightarrow ok' \land \widetilde{Post}! \\ \land & \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right) \end{array}
$$

[prop calc]

$$
= \quad \begin{array}{ll} & \forall x' \bullet \left(\begin{array}{l} Pre! \land ok' \land Post! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet (\widetilde{Pre}! \Rightarrow \widetilde{Post!}) \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \\ \land & \forall x' \bullet \left(\begin{array}{l} \neg\, Pre! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet \neg\, \widetilde{Pre}! \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array}
$$

[ok' free in antecedent]

$$
= \quad \begin{array}{ll} & \forall x' \bullet \left(\begin{array}{l} Pre! \land ok' \land Post! \land tr \leq tr' \Rightarrow \\ \quad \exists \widetilde{x'} \bullet (\widetilde{Pre}! \Rightarrow \widetilde{Post!}) \land \widetilde{tr} \leq \widetilde{tr'} \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'\,(\theta) \end{array} \right) \\ \land & \forall x' \bullet \left(\neg\, Pre! \Rightarrow \exists \widetilde{x'} \bullet \neg\, \widetilde{Pre}! \land \theta' \right) \end{array}
$$

[x' free in $Pre!$; $ok', wait'$ free in antecendent]

The second conjunct is simplified by the proviso $\forall x \bullet \exists \widetilde{x} \bullet \theta$, which is equivalent to $\forall x' \bullet \exists \widetilde{x'} \bullet \theta'$:

$$
\begin{array}{ll} & \forall x' \bullet \left(\neg\, Pre! \Rightarrow \exists \widetilde{x'} \bullet \neg\, \widetilde{Pre}! \land \theta' \right) \\ = & \forall x' \bullet \left(\neg\, Pre! \Rightarrow \exists \widetilde{x'} \bullet \neg\, \widetilde{Pre}! \right) \qquad \text{[proviso]} \end{array}
$$

$$= \quad \neg\, Pre! \Rightarrow \neg\, \widetilde{Pre}! \qquad [Pre \text{ a condition}]$$
$$= \quad \widetilde{Pre}! \Rightarrow Pre! \qquad [\text{contrapositive}]$$

Proof of Theorem 6.14

$$\begin{aligned}
& \mathbf{bwR}\,(\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post))) \\
= \quad & \exists\, x', \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post))! && [\text{def } \mathbf{bwR}] \\
= \quad & \exists\, x', \widetilde{x'} \bullet \widehat{\mathbf{R}}\left(Pre \vee \widetilde{Pre} \vdash \mathbf{U}\,(Pre \Rightarrow Post)\right)! \wedge \mathcal{I}(\mathcal{L})! && [\text{Lemma 5.8}] \\
= \quad & \exists\, x', \widetilde{x'} \bullet \mathbf{U}\,(tr \leq tr') \wedge ((Pre! \vee \widetilde{Pre}!) \Rightarrow ok' \wedge \mathbf{U}\,(Pre! \Rightarrow Post!)) \wedge \mathcal{I}(\mathcal{L})! \\
& && [\text{!-apply}]
\end{aligned}$$

Proof of Lemma 6.16

To prove **bw** (B, θ) is monotonic in θ, it suffices to prove that **bwQ** (B, θ) is monotonic in θ. Let $\theta_1 \sqsubseteq \theta_2$. Then:

$$\begin{aligned}
& \mathbf{bwQ}\,(B, \theta_2) \\
= \quad & \forall\, x' \bullet \mathbf{D}\,(B!) \Rightarrow \exists\, \widetilde{x'} \bullet B! \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta_2') && [\text{def } \mathbf{bwQ}] \\
\Rightarrow \quad & \forall\, x' \bullet \mathbf{D}\,(B!) \Rightarrow \exists\, \widetilde{x'} \bullet B! \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta_1') && [\theta_1 \sqsubseteq \theta_2] \\
= \quad & \mathbf{bwQ}\,(B, \theta_1) && [\text{def } \mathbf{bwQ}]
\end{aligned}$$

Proof of Law 6.18

First, we calculate **bwR** $(\langle\, \theta_1 \,\rangle)$:

$$\begin{aligned}
& \mathbf{bwR}\,(\langle\, \theta_1 \,\rangle) \\
= \quad & \exists\, x', \widetilde{x'} \bullet \left(\mathbf{UC}\,(\mathcal{L}, Skip) \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta_1')\right)! && [\text{def } \mathbf{bwR}, \text{def CA}] \\
= \quad & \exists\, x', \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, Skip)! \wedge \theta_1' && [\text{def !}] \\
= \quad & \theta_1 && [\text{def } Skip]
\end{aligned}$$

Second, we calculate $\mathbf{bwQ}\,(\langle\,\theta_1\,\rangle, \theta_2)$:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{bwQ}\,(\langle\,\theta_1\,\rangle, \theta_2)} \\
= & \forall\, x' \bullet \mathbf{D}\,(\langle\,\theta_1\,\rangle!) \Rightarrow \exists\, \widetilde{x'} \bullet \langle\,\theta_1\,\rangle! \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta_2') & [\text{def } \mathbf{bw}] \\
= & \forall\, x' \bullet \mathbf{D}\,(\langle\,\theta_1\,\rangle!) \Rightarrow \exists\, \widetilde{x'} \bullet \langle\,\theta_1\,\rangle! \wedge \theta_2 & [\text{property of } Skip] \\
= & \forall\, x' \bullet \mathbf{D}\,(\theta_1) \Rightarrow \exists\, \widetilde{x'} \bullet \theta_1 \wedge \theta_2 & [\text{as above}] \\
= & \mathbf{D}\,(\theta_1) \Rightarrow \theta_1 \wedge \theta_2 & [x', \widetilde{x'} \text{ free in } \theta_1, \theta_2]
\end{array}$$

The result follows by conjoining the **bwR** and **bwQ** terms:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{bwR}\,(\langle\,\theta_1\,\rangle) \wedge \mathbf{bwQ}\,(\langle\,\theta_1\,\rangle, \theta_2)} \\
= & \theta_1 \wedge (\mathbf{D}\,(\theta_1) \Rightarrow \theta_1 \wedge \theta_2) & [\text{as above}] \\
= & \theta_1 \wedge \theta_2 & [\mathbf{D}\,(\theta_1) \sqsubseteq \theta_1]
\end{array}$$

Proof of Lemma 6.19

By the definition of **bwR**, we have:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{bwR}\,(B)} \\
= & \exists\, x', \widetilde{x'} \bullet B! & [\text{def } \mathbf{bwR}] \\
= & \exists\, x', \widetilde{x'} \bullet (B \wedge (C \vee \neg\, C))! & [\text{excluded middle}] \\
= & \exists\, x', \widetilde{x'} \bullet (B \wedge C)! \vee \exists\, x', \widetilde{x'} \bullet (B \wedge \neg\, C)! & [\text{prop calc}] \\
= & \mathbf{bwR}\,(B \wedge C) \vee \mathbf{bwR}\,(B \wedge \neg\, C) & [\text{def } \mathbf{bwR}] \\
\Leftarrow & \mathbf{bwR}\,(B \wedge C) \wedge \mathbf{bwR}\,(B \wedge \neg\, C) & [\text{prop calc}]
\end{array}$$

The proof for **bwQ** has a similar structure:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{bwQ}\,(B, \theta)} \\
= & \forall\, x' \bullet \Big(\mathbf{D}\,(B!) \Rightarrow \exists\, \widetilde{x'} \bullet B! \wedge \mathsf{Conf}'\,(\theta)\Big) & [\text{def } \mathbf{bwQ}] \\
= & \left(\begin{array}{l} \forall\, x' \bullet \mathbf{D}\,((B \wedge (C \vee \neg\, C))!) \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet (B \wedge (C \vee \neg\, C))! \wedge \mathsf{Conf}'\,(\theta) \end{array}\right) & [\text{excluded middle}]
\end{array}$$

$$\begin{array}{ll} \Leftarrow & \left(\begin{array}{ll} & \forall x' \bullet \Big(\mathbf{D}\left((B \wedge C)!\right) \Rightarrow \exists \widetilde{x'} \bullet (B \wedge C)! \wedge \mathsf{Conf}'(\theta)\Big) \\ \wedge & \forall x' \bullet \Big(\mathbf{D}\left((B \wedge \neg C)!\right) \Rightarrow \exists \widetilde{x'} \bullet (B \wedge \neg C)! \wedge \mathsf{Conf}'(\theta)\Big) \end{array}\right) \\ & \hfill \text{[pred calc]} \\ = & \mathbf{bwQ}(B \wedge C, \theta) \wedge \mathbf{bwQ}(B \wedge \neg C, \theta) \hfill \text{[def } \mathbf{bwQ}\text{]} \end{array}$$

The result follows by conjoining the **bwR** and **bwQ** terms together.

Proof of Theorem 6.20

The proof for **bwQ** depends on side-conditions 3 and 4:

$$\begin{array}{lll} & \mathbf{bwQ}(B, \theta) & \\ = & \forall x' \bullet \Big(\mathbf{D}(B!) \Rightarrow \exists \widetilde{x'} \bullet B! \wedge \mathsf{Conf}'(\theta)\Big) & \text{[def } \mathbf{bwQ}\text{]} \\ = & \left(\begin{array}{ll} & \forall x' \bullet \mathbf{D}(B!) \wedge C \Rightarrow \exists \widetilde{x'} \bullet B! \wedge \mathsf{Conf}'(\theta) \\ \wedge & \forall x' \bullet \mathbf{D}(B!) \wedge \neg C \Rightarrow \exists \widetilde{x'} \bullet B! \wedge \mathsf{Conf}'(\theta) \end{array}\right) & \text{[case split]} \\ = & \left(\begin{array}{ll} & \forall x' \bullet \mathbf{D}(B!) \wedge C \Rightarrow \exists \widetilde{x'} \bullet B! \wedge \mathbf{U}(C) \wedge \mathsf{Conf}'(\theta) \\ \wedge & \forall x' \bullet \mathbf{D}(B!) \wedge \neg C \Rightarrow \exists \widetilde{x'} \bullet B! \wedge \mathbf{U}(\neg C) \wedge \mathsf{Conf}'(\theta) \end{array}\right) & \text{[SC 4]} \\ = & \left(\begin{array}{ll} & \forall x' \bullet \mathbf{D}\left((B \wedge \mathbf{U}(C))!\right) \Rightarrow \exists \widetilde{x'} \bullet (B \wedge \mathbf{U}(C))! \wedge \mathsf{Conf}'(\theta) \\ \wedge & \forall x' \bullet \mathbf{D}\left((B \wedge \mathbf{U}(\neg C))!\right) \Rightarrow \exists \widetilde{x'} \bullet (B \wedge \mathbf{U}(\neg C))! \wedge \mathsf{Conf}'(\theta) \end{array}\right) & \\ & & \text{[SC 4]} \\ = & \mathbf{bwQ}(B \wedge \mathbf{U}(C), \theta) \wedge \mathbf{bwQ}(B \wedge \mathbf{U}(\neg C), \theta) & \text{[def } \mathbf{bwQ}\text{]} \\ = & \mathbf{bwQ}(B \wedge C, \theta) \wedge \mathbf{bwQ}(B \wedge \neg C, \theta) & \text{[SC 4]} \end{array}$$

Hence:

$$\begin{array}{lll} & \mathbf{bw}(B, \theta) & \\ = & \mathbf{bwR}(B) \wedge \mathbf{bwQ}(B, \theta) & \text{[def } \mathbf{bw}\text{]} \\ = & \mathbf{bwR}(B \wedge C) \wedge \mathbf{bwR}(B \wedge \neg C) \wedge \mathbf{bwQ}(B, \theta) & \text{[property of } \mathbf{bwR}\ (*)\text{]} \\ = & \mathbf{bwR}(B \wedge C) \wedge \mathbf{bwR}(B \wedge \neg C) \wedge \mathbf{bwQ}(B \wedge C, \theta) \wedge \mathbf{bwQ}(B \wedge \neg C, \theta) & \\ & & \text{[as above]} \\ = & \mathbf{bw}(B \wedge C, \theta) \wedge \mathbf{bw}(B \wedge \neg C, \theta) & \text{[def } \mathbf{bw}\text{]} \end{array}$$

The property of **bwR** in question is that side-conditions 1 and 2 each imply $\textsf{\textbf{bwR}}\,(B) = \textbf{true}$.

Proof of Lemma 6.21

It suffices to establish *HT* respects side-condition 4 of Theorem 6.20:

$$
\begin{array}{lll}
\multicolumn{3}{l}{HT \wedge \mathcal{I}(\mathcal{L})} \\
= & (tr' - tr) \upharpoonright \mathcal{W} \in S \wedge (tr' - tr) \upharpoonright \mathcal{L} = (\widetilde{tr}' - \widetilde{tr}) \upharpoonright \mathcal{L} \wedge \mathcal{I}(\mathcal{L}) & \\
& & [\text{def } HT, \text{def } \mathcal{I}(\mathcal{L})] \\
= & (tr' - tr) \upharpoonright \mathcal{W} \in S \wedge (tr' - tr) \upharpoonright \mathcal{W} = (\widetilde{tr}' - \widetilde{tr}) \upharpoonright \mathcal{W} \wedge \mathcal{I}(\mathcal{L}) & [\mathcal{W} \subseteq \mathcal{L}] \\
= & (tr' - tr) \upharpoonright \mathcal{W} \in S \wedge (\widetilde{tr}' - \widetilde{tr}) \upharpoonright \mathcal{W} \in S \wedge \mathcal{I}(\mathcal{L}) & [\text{substitution}] \\
= & \textsf{\textbf{U}}\,(HT) \wedge \mathcal{I}(\mathcal{L}) & [\text{def } HT, \text{def } \textsf{\textbf{U}}]
\end{array}
$$

Proofs for Lemma 6.22 and Lemma 6.23 are similar.

Proof of Law 6.24

Skip is not miraculous, so $\textsf{\textbf{bw}}\,(\langle\, Skip\,\rangle\,,\theta) = \textsf{\textbf{bwQ}}\,(\langle\, Skip\,\rangle\,,\theta)$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsf{\textbf{bwQ}}\,(\langle\, Skip\,\rangle\,,\theta)} \\
= & \textsf{\textbf{bwQ}}\,(\textsf{\textbf{UC}}\,(\mathcal{L}, \textsf{\textbf{R}}\,(\textbf{true} \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v))\,,\theta) & [\text{def } Skip] \\
= & \forall\, x' \bullet \left(\begin{array}{l} ok' \wedge tr' = tr \wedge \neg\, wait' \wedge v' = v \\ \quad \Rightarrow \exists\, \widetilde{x}' \bullet \widetilde{tr}' = \widetilde{tr} \wedge \neg\, \widetilde{wait'} \wedge \widetilde{v}' = \widetilde{v} \wedge \mathcal{I}(\mathcal{L})! \wedge \textsf{Conf}'\,(\theta) \end{array} \right) & \\
& & [\text{Theorem 6.12}] \\
= & \forall\, v' \bullet (v' = v \Rightarrow \exists\, \widetilde{v}' \bullet \widetilde{v}' = \widetilde{v} \wedge \theta') & [\text{pred calc}] \\
= & \theta'[v, \widetilde{v} / v', \widetilde{v}'] & [\text{one point rule}] \\
= & \theta & [\text{renaming}]
\end{array}
$$

Proof of Law 6.25

Stop is not miraculous, so $\mathsf{bw}\,(\langle\, Stop\,\rangle\,,\theta) = \mathsf{bwQ}\,(\langle\, Stop\,\rangle\,,\theta)$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{bwQ}\,(\langle\, Stop\,\rangle\,,\theta)} \\
= & \mathsf{bwQ}\,(\mathsf{UC}\,(\mathcal{L}, \mathsf{R}\,(\mathbf{true} \vdash tr' = tr \wedge wait'))\,,\theta) \qquad \text{[def } Stop\text{]} \\
= & \forall\, x' \bullet \left(\begin{array}{l} ok' \wedge tr' = tr \wedge wait' \\ \quad \Rightarrow \exists\, \widetilde{x'} \bullet \widetilde{tr'} = \widetilde{tr} \wedge \widetilde{wait'} \wedge \mathcal{I}(\mathcal{L})! \wedge \mathsf{Conf}'\,(\theta) \end{array} \right) \\
& \qquad \text{[Theorem 6.12]} \\
= & \forall\, x' \bullet \left(\begin{array}{l} ok' \wedge tr' = tr \wedge wait' \\ \quad \Rightarrow \exists\, \widetilde{x'} \bullet \widetilde{tr'} = \widetilde{tr} \wedge \widetilde{wait'} \wedge \mathcal{I}(\mathcal{L})! \end{array} \right) \qquad \text{[def } \mathsf{Conf}'\text{]} \\
= & \mathbf{true} \qquad \text{[pred calc]}
\end{array}
$$

Proof of Law 6.26

Chaos is not miraculous, so $\mathsf{bw}\,(\langle\, Chaos\,\rangle\,,\theta) = \mathsf{bwQ}\,(\langle\, Chaos\,\rangle\,,\theta)$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{bwQ}\,(\langle\, Chaos\,\rangle\,,\theta)} \\
= & \mathsf{bwQ}\,(\mathsf{UC}\,(\mathcal{L}, \mathsf{R}\,(\mathbf{false} \vdash \mathbf{true}))\,,\theta) \qquad \text{[def } Chaos\text{]} \\
= & \forall\, x' \bullet (\mathbf{false}! \wedge ok' \wedge \mathbf{true}! \wedge tr \leq tr' \Rightarrow \ldots) \wedge \widetilde{\mathbf{false}}! \Rightarrow \mathbf{false}! \\
& \qquad \text{[Theorem 6.12]} \\
= & \mathbf{true} \qquad \text{[pred calc]}
\end{array}
$$

Proof of Law 6.27

The scope constructs are not miraculous, so we have $\mathsf{bw}\,(\langle\, \mathbf{var}\; a : T\,\rangle\,,\theta) = \mathsf{bwQ}\,(\langle\, \mathbf{var}\; a : T\,\rangle\,,\theta)$ and $\mathsf{bw}\,(\langle\, \mathbf{end}\; a : T\,\rangle\,,\theta) = \mathsf{bwQ}\,(\langle\, \mathbf{end}\; a : T\,\rangle\,,\theta)$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{bw}\,(\langle\, \mathbf{var}\; a : T\,\rangle\,,\theta)} \\
= & \mathsf{bwQ}\,(\langle\, \mathbf{var}\; a : T\,\rangle\,,\theta) \qquad \text{[}\mathbf{var}\text{ non-miraculous]} \\
= & \forall\, x' \bullet \mathsf{D}\,(\mathsf{UC}\,(\mathcal{L}, \mathbf{var}\; a : T))! \Rightarrow \exists\, \widetilde{x'} \bullet \mathsf{UC}\,(\mathcal{L}, \mathbf{var}\; a : T)! \wedge \mathsf{Conf}'\,(\theta)
\end{array}
$$

[def **bwQ**]

$$= \quad \forall x' \bullet (\exists a : T \bullet \mathit{II})! \Rightarrow \exists \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, \exists a : T \bullet \mathit{II})! \wedge (ok' \wedge \neg\, wait' \Rightarrow \theta')$$

[def **var**, Conf′]

$$= \quad \forall x' \bullet (\exists a : T \bullet \mathit{II})! \Rightarrow \exists \widetilde{x'} \bullet \mathbf{UC}\,(\mathcal{L}, \exists a : T \bullet \mathit{II})! \wedge \theta' \qquad [\text{property of } \mathit{II}]$$

$$= \quad \forall v' \bullet (\exists a : T \bullet \mathit{II}!) \Rightarrow \exists \widetilde{v'} \bullet (\exists \widetilde{a} : T \bullet \widetilde{\mathit{II}}!) \wedge \theta' \qquad [w', \widetilde{w'} \text{ free in } \theta']$$

$$= \quad \forall v' \bullet (\exists a : T \bullet \mathit{II}!) \wedge a' \in T \Rightarrow \exists \widetilde{v'} \bullet (\exists \widetilde{a} : T \bullet \widetilde{\mathit{II}}!) \wedge \widetilde{a'} \in T \wedge \theta'$$

[property of II]

$$= \quad \forall v' \bullet (\exists a : T \bullet \mathit{II}!) \wedge a' \in T \Rightarrow (\exists \widetilde{a'} : T \bullet \theta')[\widetilde{x}/\widetilde{x'}] \qquad [\text{one point rule}]$$

$$= \quad (\forall a' : T \bullet (\exists \widetilde{a'} : T \bullet \theta')[\widetilde{x}/\widetilde{x'}])[x/x'] \qquad [\text{one point rule}]$$

$$= \quad \forall a : T \bullet \exists \widetilde{a} : T \bullet \theta \qquad [\text{renaming}]$$

The proof that **bw** $(\langle\, \mathbf{end}\ a : T \,\rangle, \theta) = \forall a \bullet \exists \widetilde{a} : T \bullet \theta$ is similar. Hence:

bw $(\mathbf{var}\ a : T \bullet B, \theta)$

$$= \quad \mathbf{bw}\,(\langle\, \mathbf{var}\ a : T \,\rangle \,; B \,; \langle\, \mathbf{end}\ a : T \,\rangle, \theta) \qquad [\text{def } \mathbf{var}]$$

$$= \quad \mathbf{bw}\,(\langle\, \mathbf{var}\ a : T \,\rangle, \mathbf{bw}\,(B, \mathbf{bw}\,(\langle\, \mathbf{end}\ a : T \,\rangle, \theta))) \qquad [\text{Law 6.32, twice}]$$

$$= \quad \forall a : T \bullet \exists \widetilde{a} : T \bullet \mathbf{bw}\,(B, \forall a : T \bullet \exists \widetilde{a} : T \bullet \theta)$$

Proof of Law 6.28

Let X abbreviate $\neg\, wait' \wedge tr' = tr \wedge u' = u$ in the following:

bwR $(\mathbf{UC}\,(\mathcal{L}, w : [Pre, Post]))$

$$= \quad \mathbf{bwR}\,(\mathbf{UC}\,(\mathcal{L}, \mathbf{R}\,(Pre \vdash Post \wedge X))) \qquad [\text{def } \mathbf{bwR}, \text{ spec stmt}]$$

$$= \quad \exists x', \widetilde{x'} \bullet \left(\begin{array}{ll} & \mathbf{U}\,(tr \le tr') \\ \wedge & ((Pre! \vee \widetilde{Pre}!) \Rightarrow ok' \wedge \mathbf{U}\,(Pre! \Rightarrow Post! \wedge X)) \wedge \mathcal{I}(\mathcal{L})! \end{array} \right)$$

[Theorem 6.14]

$$= \quad \exists v', \widetilde{v'} \bullet (Pre \vee \widetilde{Pre}) \Rightarrow \mathbf{U}\,(Pre \Rightarrow Post \wedge u' = u) \qquad [Pre, Post \text{ over } v, v']$$

$$= \quad \exists v', \widetilde{v'} \bullet \mathbf{U}\,(Pre \Rightarrow Post \wedge u' = u) \qquad [\text{prop calc}]$$

$$= \quad \mathbf{U}\,(Pre \Rightarrow \exists v' \bullet Post \wedge u' = u) \qquad [Pre \text{ a condition}]$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, w : [Pre, Post])\,, \theta)} \\
= & \textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, \textsf{R}\,(Pre \vdash Post \wedge X))\,, \theta) \hfill [\text{def } \textsf{bwQ}\text{, spec stmt}] \\
= & \forall\, x' \bullet \left(\begin{array}{l} Pre! \wedge ok' \wedge Post! \wedge X \\ \quad \Rightarrow \exists\, \widetilde{x'} \bullet \left(\begin{array}{ll} & \widetilde{Pre}! \Rightarrow (\widetilde{Post!} \wedge \widetilde{X}) \\ \wedge & \widetilde{tr} \leq \widetilde{tr'} \wedge \mathcal{I}(\mathcal{L})! \wedge \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right) \\
& \wedge \quad \widetilde{Pre}! \Rightarrow Pre! \\
& \hfill [\text{Theorem 6.12}] \\
= & \forall\, v' \bullet \left(\begin{array}{l} (Pre \wedge Post \wedge u' = u) \Rightarrow \\ \quad \exists\, \widetilde{v'} \bullet (\widetilde{Pre} \Rightarrow \widetilde{Post} \wedge \widetilde{u'} = \widetilde{u}) \wedge \theta' \end{array} \right) \wedge (\widetilde{Pre} \Rightarrow Pre) \\
& \hfill [Pre, Post, \theta \text{ over state variables}]
\end{array}
$$

Proof of Law 6.29

Assumptions are not miraculous, so $\textsf{bw}\,(\langle\, \{\, C \,\}\,\rangle\,, \theta) = \textsf{bwQ}\,(\langle\, \{\, C \,\}\,\rangle\,, \theta)$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, \{\, C \,\})\,, \theta)} \\
= & \textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, : [C, \mathbf{true}])\,, \theta) & [\text{def assumption}] \\
= & \forall\, v' \bullet \Big((C \wedge v' = v) \Rightarrow \exists\, \widetilde{v'} \bullet (\widetilde{C} \Rightarrow \widetilde{v'} = \widetilde{v}) \wedge \theta'\Big) \wedge (\widetilde{C} \Rightarrow C) & [\text{Law 6.28}] \\
= & (C \Rightarrow \exists\, \widetilde{v'} \bullet (\widetilde{C} \Rightarrow \widetilde{v'} = \widetilde{v}) \wedge \theta'[v/v']) \wedge (\widetilde{C} \Rightarrow C) & [\text{one point rule}]
\end{array}
$$

Proof of Law 6.31

Assignment is not miraculous, so $\textsf{bw}\,(\langle\, a := E \,\rangle\,, \theta) = \textsf{bwQ}\,(\langle\, a := E \,\rangle\,, \theta)$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, \{\, C \,\})\,, \theta)} \\
= & \textsf{bwQ}\,(\textsf{UC}\,(\mathcal{L}, a : [\mathbf{true}, a' = E])\,, \theta) & [\text{def assignment}] \\
= & \forall\, v' \bullet \Big((a' = E \wedge u' = u) \Rightarrow \exists\, \widetilde{v'} \bullet (\widetilde{a'} = \widetilde{E} \wedge \widetilde{u'} = \widetilde{u}) \wedge \theta'\Big) & [\text{Law 6.28}] \\
= & \theta'[E, \widetilde{E}/a', \widetilde{a'}][u/u'] & [\text{one point rule}]
\end{array}
$$

$$= \quad \theta[E, \widetilde{E}/a, \widetilde{a}] \qquad \text{[renaming]}$$

Proof of Law 6.32

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{bwR}\left(B_1 \mathbin{\widehat{;}} B_2\right)} \\
= & \exists\, x', \widetilde{x'} \bullet \left(B_1 \mathbin{\widehat{;}} B_2\right)! \qquad \text{[def } \mathbf{bwR}\text{]} \\
= & \exists\, x', \widetilde{x'} \bullet B_1! \mathbin{\widehat{;}} \mathsf{Conf}\,(B_2!) \qquad \text{[property of sequential composition]} \\
= & \exists\, x', \widetilde{x'} \bullet \exists\, x_0, \widetilde{x_0} \bullet B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \wedge \mathsf{Conf}\,(B_2!)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \qquad \text{[def } \widehat{;}\text{]} \\
= & \exists\, x_0, \widetilde{x_0} \bullet B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \wedge \mathsf{Conf}\left(\exists\, x', \widetilde{x'} \bullet B_2!\right)[x_0, \widetilde{x_0}/x, \widetilde{x}] \\
 & \qquad [x', \widetilde{x'} \text{ free in } B_1[x_0, \widetilde{x_0}/x', \widetilde{x'}]] \\
= & \exists\, x', \widetilde{x'} \bullet B_1! \wedge \mathsf{Conf}\left(\exists\, x', \widetilde{x'} \bullet B_2!\right)[x', \widetilde{x'}/x, \widetilde{x}] \qquad \text{[renaming]} \\
= & \mathbf{bwR}\left(B_1 \wedge \mathsf{Conf}\,(\mathbf{bwR}\,(B_2))\,[x', \widetilde{x'}/x, \widetilde{x}]\right) \qquad \text{[def } \mathbf{bwR}\text{]} \\
= & \mathbf{bwR}\left(B_1 \wedge \mathsf{Conf}'\left(\mathbf{bwR}\left(B_2[x', \widetilde{x'}/x, \widetilde{x}]\right)\right)\right) \qquad \text{[renaming]}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{bwQ}\left(B_1 \mathbin{\widehat{;}} B_2, \theta\right)} \\
= & \forall\, x' \bullet \left(\mathbf{D}\left(B_1 \mathbin{\widehat{;}} B_2\right)! \Rightarrow \exists\, \widetilde{x'} \bullet (B_1 \mathbin{\widehat{;}} B_2)! \wedge \mathsf{Conf}'\,(\theta)\right) \qquad \text{[def } \mathbf{bwQ}\text{]} \\
= & \forall\, x' \bullet \left((\mathbf{D}\,(B_1)\ ;\mathbf{D}\,(B_2))! \Rightarrow \exists\, \widetilde{x'} \bullet (B_1 \mathbin{\widehat{;}} B_2)! \wedge \mathsf{Conf}'\,(\theta)\right) \qquad \text{[Condition 5.11]} \\
= & \forall\, x' \bullet \left(\begin{array}{l} (\exists\, x_0 \bullet (\mathbf{D}\,(B_1)\,[x_0/x'] \wedge \mathbf{D}\,(B_2)\,[x_0/x])!) \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \left(\exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & B_1[x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & B_2[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right)! \wedge \mathsf{Conf}'\,(\theta)\right) \end{array}\right) \\
 & \qquad \text{[def } ;, \widehat{;}\text{]} \\
= & \forall\, x' \bullet \left(\begin{array}{l} (\exists\, x_0 \bullet \mathbf{D}\,(B_1)![x_0/x'] \wedge \mathsf{Conf}\,(\mathbf{D}\,(B_2))\,[x_0/x]) \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \left(\exists\, x_0, \widetilde{x_0} \bullet \left(\begin{array}{ll} & B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge & \mathsf{Conf}\,(B_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array}\right) \wedge \mathsf{Conf}'\,(\theta)\right) \end{array}\right)
\end{array}
$$

[property of sequential composition]

$$= \quad \forall\, x', x_0 \bullet \left(\begin{array}{l} \mathbf{D}\,(B_1)![x_0/x'] \Rightarrow \\ \left(\begin{array}{l} \mathsf{Conf}\,(\mathbf{D}\,(B_2))\,[x_0/x] \Rightarrow \\ \left(\begin{array}{l} \exists\, \widetilde{x_0} \bullet B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \mathsf{Conf}\,(B_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \wedge \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right) \end{array} \right)$$

[pred calc]

$$= \quad \forall\, x_0 \bullet \left(\begin{array}{l} \mathbf{D}\,(B_1)![x_0/x'] \Rightarrow \\ \left(\begin{array}{l} \exists\, \widetilde{x_0} \bullet B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \\ \wedge\, \forall\, x' \bullet \left(\begin{array}{l} \mathsf{Conf}\,(\mathbf{D}\,(B_2))\,[x_0/x] \Rightarrow \\ \quad \exists\, \widetilde{x'} \bullet \mathsf{Conf}\,(B_2)\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \wedge \mathsf{Conf}'\,(\theta) \end{array} \right) \end{array} \right) \end{array} \right)$$

[pred calc]

$$= \quad \forall\, x_0 \bullet \left(\begin{array}{l} \mathbf{D}\,(B_1)![x_0/x'] \Rightarrow \\ \quad \exists\, \widetilde{x_0} \bullet B_1![x_0, \widetilde{x_0}/x', \widetilde{x'}] \wedge \mathsf{Conf}\,(\mathbf{bwQ}\,(B_2, \theta))\,[x_0, \widetilde{x_0}/x, \widetilde{x}] \end{array} \right)$$

[def **bwQ**]

$$= \quad \forall\, x' \bullet \Big(\mathbf{D}\,(B_1)! \Rightarrow \exists\, \widetilde{x'} \bullet B_1! \wedge \mathsf{Conf}'\,(\mathbf{bwQ}\,(B_2, \theta)) \Big) \qquad \text{[renaming]}$$

$$= \quad \mathbf{bwQ}\,(B_1, \mathbf{bwQ}\,(B_2, \theta)) \qquad \text{[def } \mathbf{bwQ}\text{]}$$

Proof of Law 6.33

$$\mathbf{bwR}\left(g \mathbin{\widehat{\&}} B \right)$$

$$= \quad \left(\begin{array}{ll} & \mathbf{U}\,(g) \wedge \mathbf{bwR}\left(g \mathbin{\widehat{\&}} B \right) \\ \vee & \mathbf{U}\,(\neg\, g) \wedge \mathbf{bwR}\left(g \mathbin{\widehat{\&}} B \right) \\ \vee & (g \neq \widetilde{g}) \wedge \mathbf{bwR}\left(g \mathbin{\widehat{\&}} B \right) \end{array} \right) \qquad \text{[case split]}$$

$$= \quad \left(\begin{array}{ll} & \mathbf{U}\,(g) \wedge \mathbf{bwR}\,(B) \\ \vee & \mathbf{U}\,(\neg\, g) \wedge \mathbf{bwR}\,(\widehat{\mathbf{R}}\,(\mathbf{true} \vdash \mathbf{U}\,(tr' = tr \wedge wait'))) \\ \vee & (g \neq \widetilde{g}) \wedge \mathbf{bwR}\,(\widehat{\mathbf{R}}\,(\mathbf{true} \vdash \mathbf{false})) \end{array} \right) \qquad \text{[def } \widehat{\&}\text{]}$$

$$= \quad \left(\begin{array}{ll} & \mathbf{U}\,(g) \wedge \mathbf{bwR}\,(B) \\ \vee & \mathbf{U}\,(\neg\, g) \wedge \mathbf{true} \\ \vee & (g \neq \widetilde{g}) \wedge \mathbf{false} \end{array} \right) \qquad \text{[def } \mathbf{bwR}\text{]}$$

$$= \quad (\mathbf{U}\,(g) \wedge \mathbf{bwR}\,(B)) \vee \mathbf{U}\,(\neg\, g)$$

This result simplifies the calculation of **bwQ**:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{bw}\left(g \mathbin{\widehat{\&}} B,\theta\right)} & \\
= & \left(\left(\mathbf{U}(g) \land \mathbf{bwR}(B)\right) \lor \mathbf{U}(\neg g)\right) \land \mathbf{bwQ}\left(g \mathbin{\widehat{\&}} B,\theta\right) & [\text{def } \mathbf{bw}] \\
= & \left(\begin{array}{ll} & \mathbf{U}(g) \land \mathbf{bwR}(B) \land \mathbf{bwQ}\left(g \mathbin{\widehat{\&}} B,\theta\right) \\ \lor & \mathbf{U}(\neg g) \land \mathbf{bwQ}\left(g \mathbin{\widehat{\&}} B,\theta\right) \end{array}\right) & [\text{distributivity}] \\
= & \left(\begin{array}{ll} & \mathbf{U}(g) \land \mathbf{bwR}(B) \land \mathbf{bwQ}(B,\theta) \\ \lor & \mathbf{U}(\neg g) \land \mathbf{bwQ}\left(\widehat{\mathbf{R}}\left(\mathbf{true} \vdash \mathbf{U}\left(tr' = tr \land wait'\right)\right),\theta\right) \end{array}\right) & [\text{def } \widehat{\&}] \\
= & \left(\begin{array}{ll} & \mathbf{U}(g) \land \mathbf{bwR}(B) \land \mathbf{bwQ}(B,\theta) \\ \lor & \mathbf{U}(\neg g) \land \mathbf{true} \end{array}\right) & [\text{property of } \mathbf{bwQ}] \\
= & \left(\mathbf{U}(g) \land \mathbf{bw}(B,\theta)\right) \lor \mathbf{U}(\neg g) & [\text{def } \mathbf{bw}]
\end{array}$$

Proof of Law 6.34

The prefixing operator is not miraculous, so $\mathbf{bw}(\langle\, c.E \to Skip \,\rangle,\theta) = \mathbf{bwQ}(\langle\, c.E \to Skip \,\rangle,\theta)$. Let X abbreviate:

$$\left(tr' = tr \land (c,E) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (c,E) \rangle\right) \land v' = v$$

in the following:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathbf{bwQ}\left(\mathbf{UC}\left(\mathcal{L}, c.E \to Skip\right),\theta\right)} & \\
= & \mathbf{bwQ}\left(\mathbf{UC}\left(\mathcal{L}, \mathbf{R}\left(\mathbf{true} \vdash X\right)\right),\theta\right) & [\text{def prefix}] \\
= & \forall x' \bullet \left(ok' \land X! \Rightarrow \exists \widetilde{x'} \bullet \widetilde{X}! \land \mathcal{I}(\mathcal{L})! \land \mathsf{Conf}'(\theta)\right) & [\text{Theorem 6.12}] \\
= & \begin{array}{ll} & \forall x' \bullet \left(ok' \land wait' \land X! \Rightarrow \exists \widetilde{x'} \bullet \widetilde{X}! \land \mathcal{I}(\mathcal{L})!\right) \\ \land & c \in \mathcal{L} \Rightarrow \forall x' \bullet \left(ok' \land \neg\, wait' \land X! \Rightarrow \exists \widetilde{x'} \bullet \widetilde{X}! \land \mathcal{I}(\mathcal{L})! \land \theta'\right) \\ \land & c \notin \mathcal{L} \Rightarrow \forall x' \bullet \left(ok' \land \neg\, wait' \land X! \Rightarrow \exists \widetilde{x'} \bullet \widetilde{X}! \land \mathcal{I}(\mathcal{L})! \land \theta'\right) \end{array} & \\
& & [\text{case split}]
\end{array}$$

$$
\begin{array}{ll}
= & \begin{array}{l} c \in \mathcal{L} \Rightarrow \forall x' \bullet \left(\begin{array}{c} tr' = tr \frown \langle (c,E) \rangle \wedge v' = v \Rightarrow \\ \exists \widetilde{x'} \bullet \left(\begin{array}{l} \widetilde{tr'} = \widetilde{tr} \frown \langle (c,\widetilde{E}) \rangle \\ \wedge \;\; \langle (c,E) \rangle = \langle (c,\widetilde{E}) \rangle \\ \wedge \;\; \widetilde{v'} = \widetilde{v} \wedge \theta' \end{array} \right) \end{array} \right) \\ \wedge \quad c \notin \mathcal{L} \Rightarrow \forall x' \bullet \left(v' = v \Rightarrow \exists \widetilde{x'} \bullet \widetilde{v'} = \widetilde{v} \wedge \theta' \right) \end{array}
\end{array}
$$

[simplify]

$$= \quad \left(c \in \mathcal{L} \Rightarrow \theta'[v,\widetilde{v}/v',\widetilde{v'}] \wedge E = \widetilde{E} \right) \wedge \left(c \notin \mathcal{L} \Rightarrow \theta'[v,\widetilde{v}/v',\widetilde{v'}] \right)$$

[one point rule]

$$= \quad \left(c \in \mathcal{L} \Rightarrow \theta \wedge E = \widetilde{E} \right) \wedge (c \notin \mathcal{L} \Rightarrow \theta) \qquad \text{[renaming]}$$

Proof of Law 6.37

$$\textbf{bwR}\left(B_1 \,\widehat{\sqcap}\, B_2 \right)$$

$$
\begin{array}{lll}
= & \exists x', \widetilde{x'} \bullet B_1! \vee B_2! & \text{[def } \textbf{bwR}\text{]} \\
\Leftarrow & \exists x', \widetilde{x'} \bullet B_1! \wedge \exists x', \widetilde{x'} \bullet B_2! & \text{[pred calc]} \\
= & \textbf{bwR}\,(B_1) \wedge \textbf{bwR}\,(B_2) & \text{[def } \textbf{bwR}\text{]}
\end{array}
$$

$$\textbf{bwQ}\left(B_1 \,\widehat{\sqcap}\, B_2, \theta \right)$$

$$
\begin{array}{lll}
= & \forall x' \bullet \left(\mathbf{D}\,((B_1 \vee B_2)!) \Rightarrow \exists \widetilde{x'} \bullet (B_1 \vee B_2)! \wedge \mathsf{Conf}'\,(\theta) \right) & \text{[def } \textbf{bwQ}\text{]} \\
= & \forall x' \bullet \left(\mathbf{D}\,(B_1)! \vee \mathbf{D}\,(B_2)! \Rightarrow \exists \widetilde{x'} \bullet (B_1! \vee B_2!) \wedge \mathsf{Conf}'\,(\theta) \right) & \text{[Law 4.8]} \\
\Leftarrow & \left(\begin{array}{l} \quad\; \forall x' \bullet \left(\mathbf{D}\,(B_1)! \Rightarrow \exists \widetilde{x'} \bullet B_1! \wedge \mathsf{Conf}'\,(\theta) \right) \\ \wedge \;\; \forall x' \bullet \left(\mathbf{D}\,(B_2)! \Rightarrow \exists \widetilde{x'} \bullet B_2! \wedge \mathsf{Conf}'\,(\theta) \right) \end{array} \right) & \text{[pred calc]} \\
= & \textbf{bwQ}\,(B_1, \theta) \wedge \textbf{bwQ}\,(B_2, \theta) & \text{[def } \textbf{bwQ}\text{]}
\end{array}
$$

A final refinement step glues the pieces together.

Proof of Law 6.38

We prove the result for an external choice between two alternatives. It is straightforward to generalise this proof to arbitary numbers of alternatives.

First, we apply the approach described in Subsection 6.4.1 to derive:

$$
\begin{aligned}
&\mathbf{bw}\left(B_1 \mathbin{\widehat{\Box}} B_2, \theta\right) \\
\Leftarrow\;& \left(\begin{array}{ll} & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\Box}} B_2\right) \wedge \mathbf{U}\left(tr' = tr \wedge wait'\right), \theta\right) \\ \wedge & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\Box}} B_2\right) \wedge \neg\, \mathbf{U}\left(tr' = tr \wedge wait'\right), \theta\right) \end{array}\right) && \text{[Lemma 6.19]} \\
\Leftarrow\;& \left(\begin{array}{ll} & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\Box}} B_2\right) \wedge \mathbf{U}\left(tr' = tr \wedge wait'\right), \theta\right) \\ \wedge & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\Box}} B_2\right) \wedge \mathbf{U}\left(\neg\left(tr' = tr \wedge wait'\right)\right), \theta\right) \end{array}\right) && \text{[Law 4.13]} \\
=\;& \left(\begin{array}{ll} & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\Box}} B_2\right) \wedge \mathbf{U}\left(tr' = tr \wedge wait'\right), \theta\right) \\ \wedge & \mathbf{bw}\left(\left(B_1 \mathbin{\widehat{\sqcap}} B_2\right) \wedge \mathbf{U}\left(\neg\left(tr' = tr \wedge wait'\right)\right), \theta\right) \end{array}\right) \\
& && \text{[property of } \widehat{\Box}\text{]}
\end{aligned}
$$

The last step is a consequence of Definition 5.27: when $\neg\left(tr' = tr \wedge wait'\right)$ holds, (lifted) external choice is equivalent to the (lifted) reactive design form of (lifted) internal choice.

Assuming at least one of the guards holds in each state, we sketch how this result can be specialised to an external choice of guarded prefixes:

$$
\begin{aligned}
&\mathbf{bw}\left(g_1 \mathbin{\widehat{\&}} c_1 \mathbin{\widehat{\rightarrow}} B_1 \mathbin{\widehat{\Box}} g_2 \mathbin{\widehat{\&}} c_2 \mathbin{\widehat{\rightarrow}} B_2, \theta\right) \\
\Leftarrow\;& \left(\begin{array}{ll} & \mathbf{bw}\left(\left(\begin{array}{ll} & g_1 \mathbin{\widehat{\&}} c_1 \mathbin{\widehat{\rightarrow}} B_1 \mathbin{\widehat{\Box}} g_2 \mathbin{\widehat{\&}} c_2 \mathbin{\widehat{\rightarrow}} B_2 \\ \wedge & \mathbf{U}\left(tr' = tr \wedge wait'\right) \end{array}\right), \theta\right) \\ \wedge & \mathbf{bw}\left(\left(\begin{array}{ll} & g_1 \mathbin{\widehat{\&}} c_1 \mathbin{\widehat{\rightarrow}} B_1 \mathbin{\widehat{\sqcap}} g_2 \mathbin{\widehat{\&}} c_2 \mathbin{\widehat{\rightarrow}} B_2 \\ \wedge & \mathbf{U}\left(\neg\left(tr' = tr \wedge wait'\right)\right) \end{array}\right), \theta\right) \end{array}\right) \\
& && \text{[as above]} \\
=\;& \left(\begin{array}{ll} & \mathbf{bw}\left(\left(g_1 \mathbin{\widehat{\&}} Stop \mathbin{\widehat{\Box}} g_2 \mathbin{\widehat{\&}} Stop\right) \wedge \mathbf{U}\left(tr' = tr \wedge wait'\right), \theta\right) \\ \wedge & \mathbf{bw}\left(\left(g_1 \mathbin{\widehat{\&}} B_1 \mathbin{\widehat{\sqcap}} g_2 \mathbin{\widehat{\&}} B_2\right) \wedge \mathbf{U}\left(\neg\left(tr' = tr \wedge wait'\right)\right), \theta\right) \end{array}\right)
\end{aligned}
$$

[property of **bw**]

$$= \left(\begin{array}{ll} & \mathbf{bw}\left(g_1 \mathbin{\widehat{\&}} Stop \mathbin{\widehat{\sqcap}} g_2 \mathbin{\widehat{\&}} Stop, \theta\right) \\ \wedge & \mathbf{bw}\left(g_1 \mathbin{\widehat{\&}} B_1 \mathbin{\widehat{\sqcap}} g_2 \mathbin{\widehat{\&}} B_2, \theta\right) \end{array}\right) \qquad \text{[assumption]}$$

$$\Leftarrow \left(\begin{array}{ll} & \mathbf{bw}\left(g_1 \mathbin{\widehat{\&}} Stop, \theta\right) \wedge \mathbf{bw}\left(g_2 \mathbin{\widehat{\&}} Stop, \theta\right) \\ \wedge & \mathbf{bw}\left(g_1 \mathbin{\widehat{\&}} B_1, \theta\right) \wedge \mathbf{bw}\left(g_2 \mathbin{\widehat{\&}} B_2, \theta\right) \end{array}\right) \qquad \text{[Law 6.37]}$$

$$= \left(\begin{array}{ll} & (\mathbf{U}(g_1) \wedge \mathbf{true} \vee \mathbf{U}(\neg g_1)) \\ \wedge & (\mathbf{U}(g_2) \wedge \mathbf{true} \vee \mathbf{U}(\neg g_2)) \\ \wedge & (\mathbf{U}(g_1) \wedge \mathbf{bw}(B_1, \theta) \vee \mathbf{U}(\neg g_1)) \\ \wedge & (\mathbf{U}(g_2) \wedge \mathbf{bw}(B_2, \theta) \vee \mathbf{U}(\neg g_2)) \end{array}\right) \qquad \text{[Law 6.25, Law 6.33]}$$

$$= \left(\begin{array}{ll} & (\mathbf{U}(g_1) \wedge \mathbf{bw}(B_1, \theta) \vee \mathbf{U}(\neg g_1)) \\ \wedge & (\mathbf{U}(g_2) \wedge \mathbf{bw}(B_2, \theta) \vee \mathbf{U}(\neg g_2)) \end{array}\right) \qquad \text{[prop calc]}$$

List of References

Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, first edition, June 2010. ISBN 0-521-89556-1.

Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11787006_10.

James M. Anderson. Why we need a new definition of information security. *Computers & Security*, 22(4):308–313, May 2003. doi: 10.1016/S0167-4048(03)00407-3.

Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, first edition, 2001.

R. J. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, May 1996. ISSN 0934-5043. doi: 10.1007/BF01214918.

Michael J. Banks and Jeremy L. Jacob. Unifying theories of confidentiality. In Shengchao Qin, editor, *3rd International Symposium on Unifying Theories of Programming (UTP 2010)*, volume 6445 of *Lecture Notes in Computer Science*, pages 120–136. Springer Berlin / Heidelberg, 2010a. doi: 10.1007/978-3-642-16690-7_5.

Michael J. Banks and Jeremy L. Jacob. On modelling user observations in the UTP. In Shengchao Qin, editor, *3rd International Symposium on Unifying Theories of Programming (UTP 2010)*, volume 6445 of *Lecture Notes in Computer Science*, pages 101–119. Springer Berlin / Heidelberg, 2010b. doi: 10.1007/978-3-642-16690-7_4.

Michael J. Banks and Jeremy L. Jacob. Specifying confidentiality in Circus. In *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin / Heidelberg, 2011a. doi: 10.1007/978-3-642-21437-0_18.

Michael J. Banks and Jeremy L. Jacob. Confidentiality annotations for Circus. In Christopher M. Poskitt, editor, *Proceedings of the Fourth York Doctoral Symposium on Computer Science (YDS 2011)*, pages 15–24. Department of Computer Science, University of York, UK, October 2011b.

David E. Bell and Leonard J. La Padula. Secure computer systems: Volume I – mathematical foundations, Volume II – A mathematical model, Volume III – A refinement of the mathematical model. Technical Report MTR-2547, The MITRE Corporation, Bedford, Massachusetts, 1973.

David E. Bell and Leonard J. La Padula. Secure computer system: Unified exposition and MULTICS interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, 1976.

Riccardo Bresciani and Andrew Butterfield. Towards a UTP-style framework to deal with probabilities. Technical report, Foundations and Methods Group, Trinity College Dublin, Dublin, Ireland, August 2011.

Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, February 1989.

Andrew Butterfield. Saoíthin: A theorem prover for UTP. In *3rd International Symposium on Unifying Theories of Programming (UTP 2010)*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.

Ulf Carlsen. Cryptographic protocol flaws: know your enemy. In *7th IEEE Computer Security Foundations Workshop (CSFW '94)*, pages 192–200. IEEE Computer Society Press, 1994. doi: 10.1109/CSFW.1994.315934.

A. L. C. Cavalcanti and J. C. P. Woodcock. Predicate transformers in the semantics of Circus. *IEE Proceedings - Software*, 150(2):85–94, April 2003. ISSN 1462-5970. doi: 10.1049/ip-sen:20030131.

Ana Cavalcanti and Jim Woodcock. ZRC – A refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267–289–289, March 1998. ISSN 0934-5043. doi: 10.1007/s001650050016.

Ana Cavalcanti and Jim Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11889229_6.

Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A refinement strategy for Circus. *Formal Aspects of Computing*, 15(2-3):146–181, November 2003. doi: 10.1007/s00165-003-0006-5.

Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and Systems Modeling*, 4(3):277–296–296, July 2005. ISSN 1619-1366. doi: 10.1007/s10270-005-0085-2.

K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-201-05866-9.

Tom Chothia and Apratim Guha. A statistical test for information leaks using continuous mutual information. In *2011 IEEE 24th Computer Security Foundations Symposium (CSF)*, pages 177–190, Los Alamitos, CA, USA, June 2011. IEEE. ISBN 9781-612-8464-4-6. doi: 10.1109/CSF.2011.19.

David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987. doi: 10.1109/SP.1987.10001.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, July 1992. ISSN 07431066. doi: 10.1016/0743-1066(92)90030-7.

Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA, 1982. ISBN 0-201-10150-5.

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, October 1976. ISBN 0-13-215871-X.

Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and Monographs in Computer Science. Springer-Verlag New York, Inc., New York, NY, USA, 1990. ISBN 0-387-96957-8.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983. doi: 10.1109/TIT.1983.1056650.

Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/Circus: a process specification and verification environment. Technical Report 1547, LRI, Université Paris-Sud XI, November 2011.

Riccardo Focardi and Roberto Gorrieri. A taxonomy of security properties for process algebras. *Journal of Computer Security*, 3(1):5–34, 1995.

Simon N. Foley. Believing the integrity of a system (invited talk). *Electronic Notes in Theoretical Computer Science*, 125(1):3–12, 2005. doi: 10.1016/j.entcs.2004.09.037.

Paweł Gancarski and Andrew Butterfield. The denotational semantics of slotted-Circus. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 451–466, Berlin, Heidelberg, 2009. Springer Berlin / Heidelberg. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_29.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984. doi: 10.1109/SP.1984.10019.

David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, 1981. ISBN 978-0-387-96480-5.

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined (resumé). In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP'86: Proceedings of the European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer Berlin / Heidelberg, 1986. doi: 10.1007/3-540-16442-1_14.

Eric C. R. Hehner. Predicative programming part I. *Communications of the ACM*, 27(2):134–143, 1984a. ISSN 0001-0782. doi: 10.1145/69610.357988.

Eric C. R. Hehner. Predicative programming part II. *Communications of the ACM*, 27(2):144–151, 1984b. ISSN 0001-0782. doi: 10.1145/69610.357990.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.

C. A. R. Hoare. Programming: Sorcery or science? *IEEE Software*, 1(2): 5–16, April 1984a. ISSN 0740-7459. doi: 10.1109/MS.1984.234042.

C. A. R. Hoare. Programs are predicates. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312 (1522), 1984b. ISSN 00804614. doi: 10.2307/37446.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Inc., 1985a.

C. A. R. Hoare. A couple of novelties in the propositional calculus. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 31 (9-12):173–178, 1985b. ISSN 1521-3870. doi: 10.1002/malq.19850310905.

C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall Inc., 1998.

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. ISSN 0956-7968. doi: 10.1017/S0956796899003500.

Jeremy L. Jacob. *On Shared Systems*. PhD thesis, Oxford University, 1987.

Jeremy L. Jacob. Security specifications. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 14–23, 1988. doi: 10.1109/SECPRI.1988.8094.

Jeremy L. Jacob. On the derivation of secure components. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 242–247. IEEE Computer Society, 1989a. doi: 10.1109/SECPRI.1989.36298.

Jeremy L. Jacob. Refinement of shared systems. In John A. McDermid, editor, *The Theory and Practice of Refinement: Approaches to the Develop-*

ment of Large-Scale Software Systems, pages 27–36. Butterworths, 1989b. ISBN 0-408-03981-7.

Jeremy L. Jacob. Separability and the detection of hidden channels. *Information Processing Letters*, 34(1):27–29, 1990. doi: 10.1016/0020-0190(90)90225-M.

Jeremy L. Jacob. A uniform presentation of confidentiality properties. *IEEE Transactions on Software Engineering*, 17(11):1186–1194, 1991. doi: 10.1109/32.106973.

Jeremy L. Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

Dale M. Johnson and F. Javier Thayer. Security and the composition of machines. In *First IEEE Computer Security Foundations Workshop - CSFW'88*, pages 72–89. IEEE Computer Society, May 1988.

Cliff B. Jones. *Systematic software development using VDM*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition, 1990. ISBN 0-13-880733-7.

Nigel A. Jones. Building in... information security, privacy and assurance - A high-level roadmap. Technical report, Cyber Security Knowledge Transfer Network (CSKTN), 2009.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag Berlin / Heidelberg, 1999. doi: 10.1007/3-540-48405-1_25.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973. doi: 10.1145/362375.362389.

Heiko Mantel. Unwinding possibilistic security properties. In *Computer Security – ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 238–254. Springer Berlin / Heidelberg, 2000a. doi: 10.1007/10722599_15.

Heiko Mantel. Possibilistic definitions of security — an assembly kit. In *13th IEEE Computer Security Foundations Workshop (CSFW '00)*, pages 185–199, 2000b. doi: 10.1109/CSFW.2000.856936.

Heiko Mantel. Preserving information flow properties under refinement. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 78–91. IEEE Computer Society Press, 2001. doi: 10.1109/SECPRI.2001.924289.

Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität Saarbrücken, July 2003.

Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166, Los Alamitos, CA, USA, 1987. IEEE Computer Society. doi: 10.1109/SP.1987.10009.

Annabelle McIver. The secret art of computer programming. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, chapter 3, pages 61–78–78. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03465-7. doi: 10.1007/978-3-642-03466-4_3.

Annabelle McIver and Carroll Morgan. Compositional refinement in agent-based security protocols. *Formal Aspects of Computing*, pages 1–27, November 2010. ISSN 0934-5043. doi: 10.1007/s00165-010-0164-1.

John McLean. Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 123–131. IEEE Computer Society Press, 1987. doi: 10.1109/SP.1987.10020.

John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, pages 79–93, 1994a. doi: 10.1109/RISP.1994.296590.

John McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 1136–1145. John Wiley & Sons, Inc., 1994b.

Carroll Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall Inc., Hertfordshire, UK, second edition, 1994.

Carroll Morgan. The shadow knows: Refinement and security in sequential programs. *Science of Computer Programming*, 74(8):629–653, June 2009. ISSN 0167-6423. doi: 10.1016/j.scico.2007.09.003.

Carroll Morgan. Compositional noninterference from first principles. *Formal Aspects of Computing*, 24(1):3–26, January 2012. ISSN 1433-299X. doi: 10.1007/s00165-010-0167-y.

Chunyan Mu. Quantitative information flow for security: a survey. Technical Report TR-08-06, Department of Computer Science, King's College London, September 2008.

Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989. ISSN 0164-0925. doi: 10.1145/69558.69559.

NIST. Minimum security requirements for federal information and information systems. Technical Report 200, National Institute of Standards and Technology, Gaithersburg, MD, 2006.

Colin O'Halloran. A calculus of information flow. In *ESORICS 90 – First European Symposium on Research in Computer Security*, pages 147–159. AFCET, 1990.

M. V. M. Oliveira, A. C. Gurgel, and C. G. Castro. CRefine: Support for the Circus refinement calculus. In *Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290. IEEE, November 2008. ISBN 978-0-7695-3437-4. doi: 10.1109/SEFM.2008.9.

Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21(1):3–32, February 2009. doi: 10.1007/s00165-007-0052-5.

Marcel V. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2005.

A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994. ISBN 0-13-294844-3.

A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society, 1995. doi: 10.1109/SECPRI.1995.398927.

A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0-13-674409-5.

A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-Interference through determinism. In *ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 33–53. Springer Berlin / Heidelberg, 1994. doi: 10.1007/3-540-58618-0_55.

John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles (SOSP '81)*, pages 12–21. ACM, December 1981. doi: 10.1145/800216.806586.

Peter Ryan. Mathematical models of computer security. In *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 1–62. Springer Berlin / Heidelberg, 2001. doi: 10.1007/3-540-45608-2_1.

Peter Ryan and Steve Schneider. An attack on a recursive authentication protocol. A cautionary tale. *Information Processing Letters*, 65(1):7–10, 1998. ISSN 0020-0190. doi: 10.1016/S0020-0190(97)00180-4.

Peter Ryan and Steve A. Schneider. Process algebra and non-interference. In *12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 214–227. IEEE Computer Society, 1999. doi: 10.1109/CSFW.1999.779775.

Peter Ryan, John McLean, Jon Millen, and Vergil Gligor. Non-interference: who needs it? In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 237–238. IEEE Computer Society, 2001. doi: 10.1109/CSFW.2001.930149.

Hossein Saiedian. An invitation to formal methods. *Computer*, 29(4): 16–17, April 1996. ISSN 0018-9162. doi: 10.1109/MC.1996.488298.

Thomas Santen. A formal framework for confidentiality-preserving refinement. In *Computer Security – ESORICS 2006*, volume 4189 of *Lecture Notes in Computer Science*, pages 225–242. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11863908_15.

Thomas Santen. Preservation of probabilistic information flow under refinement. *Information and Computation*, 206(2-4):213–249, February 2008. doi: 10.1016/j.ic.2007.07.008.

Thomas Santen, Maritta Heisel, and Andreas Pfitzmann. Confidentiality-Preserving refinement is compositional – sometimes. In Dieter Gollmann, Günther Karjoth, and Michael Waidner, editors, *Computer Security – ESORICS 2002*, volume 2502 of *Lecture Notes in Computer Science*, pages 194–211. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-44345-2. doi: 10.1007/3-540-45853-0_12.

Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach (Worldwide Series in Computer Science)*. John Wiley & Sons, September 1999. ISBN 0-471-62373-3.

Steve Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave Macmillan, October 2001. ISBN 0-333-79284-X.

Steve Schneider and Helen Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, December 2005. ISSN 0934-5043. doi: 10.1007/s00165-005-0076-7.

Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 2004.

Dana Scott and Christopher Strachey. Toward A mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., April 1971. Polytechnic Press.

Fredrik Seehusen and Ketil Stølen. Maintaining information flow security under refinement and transformation. In *Formal Aspects in Security and Trust*, volume 4691 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-75227-1_10.

Fredrik Seehusen and Ketil Stølen. Information flow security, abstraction and composition. *IET Information Security*, 3(1):9–33, 2009. doi: 10.1049/iet-ifs:20080069.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, July 1948.

Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

J. M. Spivey. *The Z Notation: a reference manual*. Prentice Hall, second edition, 1992.

Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42412.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

Pavol Černý. *Software Model Checking for Confidentiality*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2009.

Kun Wei, Jim Woodcock, and Alan Burns. Formalising the timebands model in timed Circus. Technical report, Department of Computer Science, University of York, York, UK, August 2010.

Jeanette M. Wing. A symbiotic relationship between formal methods and security. In *Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*, pages 26–38. IEEE Computer Society, 1998. doi: 10.1109/CSDA.1998.798355.

Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, 1990. doi: 10.1109/2.58215.

Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971. ISSN 0001-0782. doi: 10.1145/362575.362577.

Jim Woodcock. The miracle of reactive programming. In Andrew Butterfield, editor, *Unifying Theories of Programming*, volume 5713 of *Lecture Notes in Computer Science*, chapter 12, pages 202–217. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14520-9. doi: 10.1007/978-3-642-14521-6_12.

Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *5th Irish Workshop on Formal Methods*, Workshops in Computing. BCS, 2001.

Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, chapter 10, pages 184–203. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45648-1_10.

Jim Woodcock and Ana Cavalcanti. A tutorial introduction to designs in Unifying Theories of Programming. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21377-2. doi: 10.1007/b96106.

Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

Jim Woodcock, Peter G. Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41 (4):1–36, October 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592436.

Michael Workman and John Gathegi. Punishment and ethics deterrents: A study of insider security contravention. *Journal of the American Society for Information Science and Technology*, 58(2):212–222, January 2007. ISSN 1532-2890. doi: 10.1002/asi.20474.

Aris Zakinthinos and E. S. Lee. A general theory of security properties. In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*, pages 94–102, 1997. doi: 10.1109/SECPRI.1997.601322.

Frank Zeyda, Bill Stoddart, and Steve Dunne. The refinement of reversible computations. In *2nd International Workshop on Refinement of Critical Systems*, 2003.