Patterns and model transformation tools for designing Contractual State Machines

Lishan Harbird

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

December 2011

Dedication

To Kevin Harbird

Abstract

Design methods for reactive systems may start with an abstract description of a proposed solution, which can be expressed in both an operational and declarative style. Typically these descriptions are then incrementally elaborated into executable programs. The aim of this research is to put this ad hoc design method on to a more systematic footing, thus helping engineers produce more reliable and robust systems. This is accomplished by providing a rigorous engineering process supported by engineer-friendly tools based on the application of refinement and refactoring patterns for Contractual State Machines.

Contractual State Machines (CoSta) are a simplified form of statecharts extended with temporal logic-based declarative specifications. A refinement pattern is an abstract way of representing a common type of refinement frequently required during the stepwise design of a system. Tool support is provided for system design and pattern application and is integrated with model checking technology for formal verification. To demonstrate the viability of the approach the new refinement and refactoring patterns are applied to the design of a system through a case study.

Contents

List of Figures

List of Tables

Acknowledgements

Firstly I would like to express my sincere gratitude to my supervisors, Dr. Andy Galloway, Prof. Richard Paige and Prof. Gerald Lüttgen who I am most grateful to for their invaluable support, guidance and mentoring throughout the duration of this degree. Special thanks go to my colleagues for the fruitful discussions and general insights that have contributed to shaping the views reflected in this thesis. I would like to thank my external examiner, Dr. Helen Treharne and my internal examiner, Prof. Susan Stepney for their insightful comments and feedback which have greatly improved the presentation and clarity of this thesis.

I am indebted to Dr. Carol Small and Prof. Ben Heydecker for truly inspiring to me to continue with my research studies and embark on a Ph.D. Last but not least, I would like to express my warmest gratitude to my family and friends especially my father Kevin Harbird and sister Rae Harbird for the stimulus and encouragement they have offered me.

Finally this work would not have been possible without the financial support of the EPSRC for the project "Refinement Patterns for Contractual Statecharts", reference EP/E034853/1, (2007-2010).

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2007 - 2011. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

I have acknowledged sources of joint and external work through explicit referencing. Some parts of this thesis have been previously published; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here.

- Lishan Harbird, Andy Galloway and Richard F. Paige. Towards a Model-Based Refinement Process for Contractual State Machines.

  In *Proceedings of the Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium*, pages 108-115, 4-7 May 2010. [100]

  A rigorous model-based approach to the stepwise design of Contractual State Machines is presented. Contractual State Machines are a simplified form of state charts extended with declarative specifications. The approach is based on application of a set of refinement patterns, that can be validated against a formal semantics and that are implemented using update-in-place model transformations.

Chapter 1

Introduction

Our modern world is dependent on complex software systems. They are embedded in components that are found in all kinds of electronic devices and other (e.g hydro-mechanical) engineering products. These range from commonplace personal gadgets and products that are essential for everyday living like mobile phones and computers, to highly complex, safety-critical control systems (e.g. full authority digital engine control [FADEC] systems for aircraft). These systems typically interact with other systems. The trend is that they are becoming more ubiquitous and interconnected [67].

There are many reasons why software quality is important, and the need to design correct software grows steadily as failures or deviations from intent could result in severe material losses (mission-critical systems) or in the extreme case even leading to loss of life (safety-critical systems) [45, 59, 124, 241].

It may be considered that design methods for reactive systems do not adequately support the way that software and systems engineers work. This thesis has been motivated by problems affecting software engineers designing real-world, safety-critical, reactive and concurrent systems [131, 152]. The main goal of this thesis is to improve on the methodological basis and tool support for current design languages for critical systems. It focuses on establishing a systematic model-driven approach to their design.

The research presented in this thesis concerns a theoretically justified specification and design method for Contractual State Machines [100] [1] which supports incremental derivation of operational designs that maintain desired properties. The specific focus is

[1] This paper presented a preliminary definition of Contractual State Machines. Forthcoming publications and technical reports will present the *de facto* final version.

on reactive systems that are in continual interaction with their environments. Reactive behaviour is usually associated with embedded systems, (e.g. aviation and automotive control systems), in which software is an integral part of devices.

Engineers who build reactive systems typically use various languages; these languages can mix operational and declarative styles. For example, they may use a mixture of descriptive English, diagrams, equations and algorithms. To express operational behaviour, such engineers often resort to a dialect of statecharts, such as Stateflow, within the MathWorks' MATLAB tool suite [248]. To express declarative constraints, engineers frequently use natural language. Occasionally this is supplemented with other notations to specify preconditions, postconditions and invariants.

Engineers may describe different features at different levels of abstraction set in the context of what has been modelled around the component being designed (e.g. in MathWorks' Simulink). However, arbitrary mixing of styles evidently lacks a sound formal underpinning and will not support the verification of design steps. To support more abstract, formal and declarative specification researchers have proposed several approaches such as the use of temporal logic [81, 161, 202, 203, 238, 239], state-based formal specifications [207, 234], and design by contract [33, 162, 182, 210].

The design process for reactive systems is generally characterised by incremental elaboration and decomposition. It is mostly ad hoc, stepwise and component-based, and potentially involves enriching abstract designs which may be both operational and declarative to fully operational designs. It is ad hoc in the sense that there is no common or standard approach used when designing a system and each design is treated as an individual case. It is stepwise, in the sense that the design process is incremental, with additional information added at each step; and component-based, meaning that a system is usually broken down into functional or logical components.

In recent years engineers have moved increasingly towards a model-driven approach to developing software for reactive systems [26]. Models are often initially designed at a high level of abstraction, and gradually enriched to a level in which they are deemed satisfactory and then converted to code. A design step can alter the behaviour of the model but consistency needs to be maintained between them. It is therefore important that automated mechanisms exist for comparing models at different levels of abstraction and validating consistency between them. These are typically not well supported by current statechart design tools. Consistency between models during the design process

is not normally verified formally but instead each model is validated independently, e.g, through simulation.

There are many reasons why both formal methods and model-driven engineering may potentially produce better evidence of correctness over traditional test-based approaches. Formal methods are precise and correctness can be proven. Potentially errors can be detected at an early stage, during the design process, rather than at the end when the design is complete [17, 57]. Model-driven engineering can be used to develop automated tools to support a rigorous design methodology that raises the level of abstraction at which engineers work from programs and components to models [16, 26, 34, 136].

The aim of the research presented in this thesis is to allow the design process to benefit from greater formality and a systematic model-based approach to refine an abstract, declarative specification to an operational design. The approach is based on Contractual State Machines, and a set of refinement and refactoring patterns which are model transformations that maintain consistency and preserve desirable properties.

Tools are implemented that support verification techniques for refinement of specifications to designs based on pattern application. Pattern application instantiates a pattern by matching wildcards in the pattern to syntactic and diagrammatic structures in the design. Contractual State Machines have a formal semantics and are a form of hierarchical state machines, embellished with contracts for expressing requirements. A case study demonstrates the methodological utility of these patterns and their supporting tools during the design process.

This chapter introduces the research area, giving the background and a brief overview of the current state of practice. The problems with conventional, current approaches are identified, and hence the motivations and rationale for this research are established. Furthermore potential solutions to the problems are discussed. A summary of the main contributions of this research is presented followed by an outline of the thesis structure.

1.1 Motivation

Reactive, concurrent systems are complex, often safety-critical and firmly established as a crucial and inevitable part of our everyday life [98,101]. The increase in the complexity of software has led directly to a higher risk of errors being introduced during the development process, and thus a greater need for precision in software engineering [114, 123]. This is particularly relevant for high-integrity systems where an extremely high assurance of

correctness is an absolute necessity rather than a desirable but non-essential requirement [152, 154, 241].

The term "software crisis" was first coined in the 1960's to describe the fact that software engineering projects were becoming too complex to manage, leading to problems ranging from extensive budget overruns to catastrophic software defects. With the rise of multi-core processors and in turn concurrent systems, the idea has been put forward that the software crisis has subsequently evolved into the "new software crisis" [254].

Formal methods can be used to achieve a high assurance of correctness. They focus on establishing mathematical and rigorous approaches to program construction and analysis. It has been argued that the use of formal methods is expected to lead to increased software quality and reliability [45]. The semantics of languages for formal methods are mathematically defined, and support tools can be used to reveal inconsistencies, ambiguities and incompleteness. Formal methods are theoretically appealing but a wider acceptance of formal methods is hindered for many reasons [108, 109, 261]. This is backed up by the fact that direct use of formal techniques, which rely heavily on the ability of engineers to work with unfamiliar notations, have not thus far been very successful in industry [3,4,35,46,95].

One such argument is that formal modelling techniques require a strong mathematical background and are therefore considered to be difficult to apply by mainstream software developers. In addition, conventional modelling is usually diagram-based; by comparison, formal specifications are considered to be less intuitive and it can be argued that therefore they are not as suitable as a communication tool [108]. This is a concern because comprehension is key to establishing validity, which is a vital consideration in safety-critical systems.

Formal methods provide a specification/design language, and possibly a process or methodology for applying the language. They can also be used for system verification and may support refinement techniques enabling models to be formally related at different intermediate levels of abstraction [63].

The correct-by-construction approach typified by refinement is appealing due to the rigour it brings to the development process and its ability to capture errors in the system specification/design at an early stage. However the stepwise refinement process is normally based on the application of very many small reasoning steps that usually entail proof obligations [17]. Jones [127] suggests that this may increase the cost of system development in terms of time and effort and for large reactive systems, this can quickly become

prohibitive. Automated support for the challenging task of discharging proof obligations for each refinement step is thus highly desirable.

Statecharts are a widely used language for reactive systems design. They are used in industry often in preference to languages with more formal mathematical foundations. They are considered easy to get to grips with initially, thus requiring less time and effort before productive results can be achieved. Statecharts are often classified as being a semi-formal language [102]. This may be due to them having a formally defined syntax but not semantics and thus no tool support for formal reasoning. Despite the fact that statecharts have been given numerous formal semantics in the literature, including a definitive one by its originator, Harel [104] none are in widespread use for formal reasoning. The diversity in the perceived meaning of diagrams has led to their interpretation varying subtly from tool to tool. Statecharts have a graphical syntax, intuitive notation and extensive tool support as exemplified by Statemate [103, 105].

When building reactive systems, statecharts are often designed in a top-down manner. First system interfaces are defined (e.g., focusing on input events and output events) and the external systems or context which supplies them is modelled using a suitable language, such as the MathWorks' Simulink, SysML, or a profile of UML [195,244,248]. The detailed design for these interfaces can then be modelled using state machines.

In essence, the state machines refine the interface model and describe how input events can be processed and how output (externally visible) events are generated. In practice the process of developing a large-scale state machine model is often done in an ad hoc way, which does not make it easy to provide evidence of correctness. Evidence of correctness is of vital importance for safety-critical reactive systems, which have substantial requirements for demonstrating the absence of defects.

Although statechart languages are a popular choice for designing reactive systems and describing operational behaviour, perceived weaknesses of particular relevance to this research include the fact that they typically do not support declarative styles of description which may be needed in the early design stages for expressing precise, high-level properties [123, 161, 162, 171]. State machine designs often become large and cumbersome; it can be argued that some parts of the design could be expressed more concisely and compactly using declarative descriptions, e.g., in the form of contracts.

Moreover top-down development by formal refinement, whether for statechart languages or other languages, is not yet common practice. The absence of support for formal

reasoning is not limited to statecharts. It is also a limitation of the models that contextualise statecharts. There are a number of reasons for this: the semantics for a language may be ambiguous or unclear or for abstract models such as those expressed in UML not defined at all. Although effort has been made to improve the situation, for instance the UML 2 semantics project [240], a uniform semantic model from requirements to implementation has not yet become commonplace.

Additionally, statechart languages do not fully support component-based software engineering which requires components to have well-defined interfaces, to enable encapsulation and reusability. Statechart components, however, may not be self-contained e.g., due to inter-level transitions. This makes it challenging to support a refinement-based design process [158, 183]. For a language to support component-based refinement a compositional semantics is needed. Currently statechart variants have subtly different informal semantics which tend not to be compositional.

It may be argued that to fully address the above issues and produce more reliable and robust systems a tool supported, rigorous, model-based approach to designing reactive concurrent systems with a formal, heterogeneous and compositional specification and design language is required. The contributions of the thesis will be discussed in more detail in the following section.

1.2 Contributions

The overall aims of the research are to enable the derivation of concurrent, data-rich, reactive programs using Contractual State Machines (CoSta) in a calculational style. The contributions of this thesis include a rigorous engineering process for developing Contractual State Machine models through application of so-called refinement and refactoring patterns, which are update-in-place model transformations. These patterns encode reasoning steps and transform a Contractual State Machine into a new model that preserves required properties.

Refinement and refactoring patterns are expressed as model transformations and are automated. The tool that supports this incorporates model checking technology to discharge side-conditions and thus offers greater analytical depth than a drawing tool and a simulator. A key rationale for the patterns and tool support is that they ensure the models that are produced throughout the development process preserve correctness to prevent the inadvertent introduction of mistakes and omissions.

This thesis builds on other work carried out within my research group which has focused on putting statecharts on a formal footing e.g. by defining a language, (Contractual State Machines) with a compositional semantics which includes declarative statements. First a brief introduction to CoSta, on which the work is based, is presented and then the key contributions are described.

1.2.1 Contractual State Machines

Contractual State Machines (CoSta) [100] [2] (see Section 2.13 for further details) are an intuitive design language that combines a simplified state machine language with temporal logic-based declarative contracts for safety-critical, reactive, concurrent systems. CoSta incorporates features of formal and semi-formal languages having a formal underpinning, a graphical and intuitive notation (state machines) for describing operational behaviour and a declarative specification language for expressing high-level temporal properties. Declarative constructs (contracts) are expressed in a language based on the μ-calculus [47,64,239].

CoSta has an integrated semantics for both data and behaviour which are freely intermixed. The statecharts dialect is based on Symbolic Transition Graphs with Assignment (STGA) [71, 155]. The semantics is currently expressed as labelled transition systems (LTS), which give meaning to state machines and contracts alike. CoSta's refinement relation and the language of contracts[3] is based on Lüttgen and Vogler's work on Logic LTS [157], which itself is derived from Ready Simulation [41].

The compositional, heterogeneous language supports component-based reasoning and modularity of specifications/designs. The heterogeneous nature of the language allows engineers to model system components at different levels of abstraction. The language provides a formal basis for a stepwise design process that refines contracts describing constraints on the behaviour of a component to state machines satisfying this prescribed behaviour or tighter contracts.

[2] This paper presented a preliminary definition of Contractual State Machines. Forthcoming publications and technical reports will present the *de facto* final version.

[3] Full formal details of the refinement relation are not yet in the public domain; this thesis does not rely on these formal details.

1.2.2 Refinement and refactoring patterns

This section describes the principal contribution of the thesis which is a systematic approach to component-based, stepwise refinement based on the application of patterns during the design process. The patterns constitute a comprehensive set of valid refinement and refactoring steps for CoSta to support the derivation, in a calculational style, of concurrent reactive programs. A *refinement pattern*, when applied, reduces nondeterminism in a specification; a *refactoring pattern* changes the structure of a specification without affecting its behaviour. The refinement process transforms an abstract (loose) specification into a concrete design that preserves the functionality of the original specification and introduces design language constructs as it proceeds. Designs can be correctly constructed in a stepwise fashion. Each step can be justified by the application of a *refinement or refactoring pattern* and possibly the discharge of side-conditions (Chapter 6). A pattern is an abstract way of representing a common type of refinement or refactoring that is frequently required during the stepwise design of a system. These patterns can aid and simplify the design process as they encapsulate refinements and refactorings whose correctness may have already been established thus reducing development effort over comparable approaches such as formal proof.

A catalogue of refinement and refactoring patterns has been proposed (in Chapters 4 and 5) and evaluated by the case study (in Chapter 7). Automated tools have been developed to support the application of patterns, model transformations and discharge of side-conditions. Patterns are a tool supported, model-based approach to the refinement process providing rigour but not at the cost of practicality. They can be pre-proven to preserve the refinement ordering, reducing the proof burden to the discharge of a proof obligation if required.

The refinement and refactoring patterns can be placed into two categories, *syntactic* or *semantic*. *Syntactic* patterns can be applied solely by interrogating the abstract syntax of the Contractual State Machine. *Semantic* patterns have side-conditions that require mathematical reasoning and thus the use of external verification tools. The side-conditions express the deeper properties that must hold for the pattern to be applicable.

It may be argued that patterns reduce the proof burden as it is not necessary to carry out a general refinement check between the designs. Instead it is only required to prove that the specific side-condition of a pattern holds which ensures that it is applicable to the design in the circumstance it is selected. The proof burden is also lessened by the

compositionality of the underlying refinement theory. Compositionality means that local refinements (e.g. parallel components or states within the hierarchy) ensure refinements of the model as a whole. This makes the refinement process efficient, and allows refinement and refactoring patterns to be applied in a local context rather than to the model as a whole.

1.2.3 Design process

The research contributions of this thesis include an exemplar design process, which is described in this section, to transform a high-level specification into an operational design. This involves gradually being more specific about the behaviour of the system, adding more detail to the design by introducing design constructs as it proceeds, and reducing nondeterminism whilst maintaining consistency with the original specification. The development will consist of a sequence of models where each model is a refinement or refactoring of a previous one in the series. A refinement corresponds to an elaboration of the more abstract model that preserves the refinement relation (and in the case of a refactoring the stronger equivalence relation).

Together the refinement and refactoring patterns provide a framework for the construction process as only valid correctness-preserving transformation rules can be applied at a certain time under certain circumstances when certain properties of the design hold. Collectively the patterns and the process of applying them will achieve the overall benefit of making the state machine refinement process more systematic and providing guarantees of correctness by construction. The automation and tools to support the design process are described in the next section.

1.2.4 Automation and tools

It is generally accepted that performing repetitive tasks, like applying refinement steps to models manually is time-consuming and error-prone and failing to complete such tasks correctly and precisely compromises the consistency and quality of the design. This has an even bigger impact in the context of model-driven engineering where models are processed to automatically produce an implementation [16]. An aim of the research is for a tool supported, model-based refinement process where the library of refinement and refactoring patterns are expressed as model transformations and integrated with model checking technology.

The tool suite supports the application of patterns and model transformations so that an engineer can highlight the component of the Contractual State Machine under investigation and select an applicable pattern. The tool will verify that the highlighted component is an instance of the abstract template of the selected pattern and if so will replace the abstract template with an instance of the concrete template. This tool holds a repository of patterns to which further patterns can be easily added.

Tool support for patterns and the engineering process has been provided, via use of the Epsilon toolset [136] and with external invocation of an SMT-based model checker to automate the discharge of side-conditions constraining the validity of application of a pattern. Thus the research presented in this thesis achieves a seamless integration of state of the art model transformation with model checking technology. The tools are used for the design of a system during the case study to validate and evaluate the refinement and refactoring patterns. This is described further in the next section.

1.2.5 Evaluation

A case study is conducted that applies the refinement and refactoring patterns to design a system. The case study presents the refinement of an abstract system description to a concrete design in order to provide empirical evidence that the rigorous development of data-rich, concurrent, reactive processes using CoSta's refinement and refactoring patterns and tool support is possible in practice. The case study illustrates the approach and demonstrates its viability. An evaluation of the refinement and refactoring patterns, systematic design process and automated tools is presented with suggestions for possible improvements and further work.

1.3 Thesis structure

Chapter 2 gives an overview of the research area, an analysis of closely related research for this thesis and identifies the remaining open research problems. Chapter 3 presents the research hypothesis, objectives and contributions for the thesis. Chapters 4 and 5 present the catalogue of refinement and refactoring patterns for Contractual State Machines. Chapter 6 addresses the implementation of software support for defining and applying the patterns. Chapter 7 evaluates the research hypothesis by applying the proposals to a case study and finally Chapter 8 evaluates the results of the research, summarises conclusions

and discusses proposals for further work.

Chapter 2

Research context

2.1 Introduction

This chapter begins with a description of reactive and concurrent systems followed by an analysis of the languages and approaches typically used for their specification and design. This chapter also incorporates a literature review, analysing closely related research. The chapter closes by identifying the remaining open problems and motivation for the research in this thesis. This is necessary to provide the context for the remaining parts of the thesis and the basis upon which it builds.

2.2 Reactive systems

This section will provide a description of the distinguishing features and characteristics of reactive systems and the approaches taken to their specification and design. Reactive systems are complex and often safety-critical. They are pervasive in modern society, and firmly established as a crucial and inevitable part of our daily life. Their application domain is very wide ranging from embedded systems for household appliances to automotive electronics, missile guidance and flight control [8, 101, 161].

The role of a reactive system is to maintain an on-going interaction with its environment, to monitor and control some physical phenomena, rather than produce some final value on termination. It is an event-driven system continuously having to change its actions, status or outputs in response to external and internal stimuli. In addition to discrete states, the specification of such a system may need to refer to physical quantities that change as continuous functions of time, such systems are referred to as hybrid.

It is a difficult problem that is not always solved adequately [14] to describe reactive behaviour in ways that are clear and understandable and at the same time formal and rigorous. Numerous formalisms and methodologies have been proposed in the literature for specification and verification methods for reactive systems [101, 103, 161, 162, 177, 178, 203, 219].

Design processes proposed for reactive systems typically transform an abstract, nondeterministic specification into a concrete, deterministic system with the aim of preserving the functionality of the original specification [8,48,114]. More specific details are gradually introduced to the design, reducing or removing nondeterminism. In industry, engineers may use a graphical modelling language for designing reactive systems such as MathWorks' Simulink and Stateflow within the MathWorks' MATLAB tool suite [248]. The latter of these is based on statecharts which is a subject of particular relevance to this thesis.

2.3 Concurrency

A concurrent system can be viewed as a collection of sequential processes, possibly running on different processors, that interact and exchange results with each other and the external environment [219]. Many concurrent systems are reactive and possibly safety-critical. Ensuring that concurrent systems are of high-quality is difficult and prohibitively expensive by conventional means (i.e. testing). Thus a lot of effort has been put into research on concurrency theory and the specification and verification methods for concurrent systems [112, 147, 198, 199, 209, 228].

The theory of concurrency is concerned with developing mathematical frameworks for modelling and verifying concurrent systems [211]. A distinguishing feature of concurrent systems is multiple threads of control that allow a concurrent program to perform computations simultaneously. This can lead to problems like race conditions, deadlocks and livelocks. Various languages and techniques have been proposed in the literature for specifying and verifying concurrent systems. The main approaches are based on temporal logic, model checking and process algebras. However there is often no distinction in practice as many approaches freely mix ideologies.

Process algebras offer an algebraic approach to the specification and verification of concurrent communicating programs [24, 25, 29]. Important formal algebraic languages for specifying and verifying concurrent systems are Bergstra and Klop's Algebra of Communicating Processes (ACP) [30], Milner's Calculus of Communicating Systems (CCS) [178],

and Hoare's Communicating Sequential Processes (CSP) [115]. Their common characteristics include the use of synchronisation on an atomic event as the basis for process interaction, and their ability to represent event occurrence, choice, abstraction and recursion.

An advantage of the algebraic approach with regards to program verification is that the same language is used for system specification and implementation. This means that system descriptions at different levels of abstraction (e.g., design and implementation) are easier to compare and relate. Using a single notation makes stepwise development from requirements to implementation conceptually more straightforward.

The semantics of CCS is based on the notion of labelled transition systems (LTS), and thus the algebraic properties of the language are usually defined in terms of bisimulation. Preorders exist for relating the abstract to the concrete, based on simulation [41]. Many are supported by workbenches such as The Concurrency Workbench [65] and its successors e.g. The Concurrency Workbench of North Carolina [66]. Labelled transition systems are important in the immediate context of the work presented in this thesis.

CSP is probably the most successful process algebra in industry with some commercial tools available to support the analysis and simulation of specifications. FDR2 [156] is a CSP model checker. The semantics of CSP can be expressed in terms of traces, failures and failures-divergence, all of which can be used to check refinement in FDR2.

There have been various promising attempts to combine CSP with other languages that fully support the ability to describe state aspects of systems such as Z [259] or B [52, 53], (and more widely, e.g. with time). Examples include, TCOZ [159], (Timed Communicating Object-Z integrates Object-Z and Timed CSP), CSP-OZ, (integrates CSP and Object-Z) [227], CSP||B, (combines CSP with B) [86]. Circus is an important example of a language that integrates CSP and Z, [187, 257], and supports a fully integrated refinement theory, to refine specifications as a whole as oppose to refining the parts relating to CSP or Z separately.

There are other examples of languages that combine formalisms, for example, Z and B [1]. State machines have been combined with action languages and various forms of temporal logic for specifying and reasoning about concurrent programs have been proposed [62]. Temporal logic are an extension of classical logic with specific operators that can be used to describe and reason about behaviour that changes over time.

CCS is sometimes used in combination with a modal logic which permits assertions

about the changing process state. In particular Hennessy-Milner Logic (HML), [238], adapts modal logic to LTS, and the full μ-calculus is an extension of HML with the introduction of maximal and minimal fixpoints. Formal properties describe the behaviour of the system under development, and must be satisfied however the system may evolve. Some of the properties that are of interest to a system designer concern liveness, deadlock freedom, security, safety, and functionality.

Spin (Promela) may be described as a model checking approach although the distinction is blurred due to its underlying theory [117]. All of the examples given above rely on model checking to a greater or lesser extent. The Promela language allows communicating guarded commands as well as CSP-like communication. Spin is its model checker.

2.4 Validation and verification

Validation is concerned with checking the intended software functionality against the actual software functionality, and is usually carried out up-front as well as post-implementation. Up-front validation involves establishing a specification/design is correct with respect to its intended behaviour before the development of the software is progressed and further commitments made. Simulation is very relevant to this form of validation. Post-implementation validation is usually achieved by *acceptance testing* where the product is tested by the end user or within its operating environment to ensure it performs as intended.

Verification involves showing a design or implementation meets its specification. Traditional industrial practice uses informal approaches ranging from peer reviews to testing and simulation. Unit and integration testing are by far the most widely used verification techniques in industry, however the cost of testing can be as high as 50% of the development costs. Testing may not be very suitable for verifying concurrent systems, which usually behave nondeterministically and where interference and race conditions can lead to an arbitrary execution order of program statements. The nondeterminism adversely affects the ability to write test cases, and their reliability; as well as the resources required to achieve adequate test coverage. Non-deterministic execution of concurrent processes leads to state-explosion problems, thus meeting adequate coverage criteria for concurrent systems is problematic.

By contrast, state of the art techniques such as a formal approach to verification involve showing that a design at some level of abstraction is consistent with designs,

specifications and properties at higher levels of abstraction according to some required notion of consistency, usually a refinement relation. Formal verification essentially provides a proof of consistency between two formal representations. These representations can be expressed with property-oriented or operational formalisms. Property-oriented formalisms specify the constraints of the system, and operational formalisms specify a model of the system in terms of mathematical structures. There are two main approaches to formal verification of desired properties of a specification, these are proof-theoretic and model checking. Proof-theoretic (or axiomatic) verification is where specifications are written in or translated into a notation of a proof system in which theorems may be proved using equational reasoning and term-rewriting for example.

Model checking involves a fully automated exhaustive search of a transition, state or inference graph [130]. Justified heuristics are used to minimise the search space needed to provide exhaustivity, or to converge on a solution faster [60, 181, 264–266]. It may be argued that formal approaches are difficult to construct and understand as they are typically carried out on mathematical representations of the system. For CCS with μ-calculus and CSP with temporal logic, model checking is a common approach. The μ-calculus or temporal logic express the property constraints or invariants which describe the functional or safety properties of the system.

Formal program verification for concurrent systems typically entails defining a deadlock and livelock free abstraction where the main action of the abstraction must preserve the state invariants (where the language supports them), and proving that the model refines the abstraction. Deadlock freedom guarantees that some action is possible, and deadlock properties usually require visibility of the action. Divergence arises when a system engages in an infinite sequence of internal actions and may result, for instance, from a computation being invoked outside its precondition.

2.5 Statecharts

Statecharts are a graphical modelling language for the design of complex reactive systems, that is, concurrent systems interacting with their environment. Statecharts are used by engineers for designing reactive systems in industry, as is typical in the avionics and aerospace domain. This is largely due to their extensive tool support and intuitive nature as exemplified by IBM Rational's Statemate [105,120]. There are many variants of statechart design tools, such as UML state machines [43,191,214,223], MathWorks' MAT-

LAB Stateflow [248], and IBM Rational's Rhapsody product [119]. Statecharts model behaviour in terms of states, transitions and actions. Their intuitive notation may be preferable, and more likely to be adopted by engineers than a more mathematically-based formal language [102].

The advantages of statecharts include their ability to express hierarchical information and concurrency in a clear and intuitive way. On the other hand there are inherent problems relating to their typically incomplete and imprecise semantics. In addition statecharts usually do not provide a declarative language for expressing desired high-level properties. Statecharts were developed in part to help deal with limitations of finite state machines (FSM) for modelling reactive systems. Finite state machine notation cannot represent hierarchical abstraction, component decomposition or concurrent states. Such systems can have very large state spaces rendering their use extremely difficult. Statecharts addressed these drawbacks by providing "depth" (hierarchy), "orthogonality" (concurrency) and "broadcasting" (communication) [102].

A state can be *basic* or *composite*. Composite states can be *AND* states and *OR* states. Graphically, hierarchy is represented as a superstate encapsulating substates. Hierarchy permits collections of states to be grouped together and entered or exited as a whole. This reduces the number of transitions between states and simplifies the model. Orthogonality is the *AND* decomposition of states which represents substates operating concurrently. Each substate has its own independent current state, thereby allowing the definition of systems having simultaneously active subsystems which communicate by broadcasting events. Orthogonality helps avoid unmanageably large state spaces and the potential exponential increase in the number of states.

The graphical notation of statecharts represents states as boxes, and transitions as directed arrows with a label consisting of an event, optional conditions and actions. Labels may be viewed as pairs of event sets. The first component of a pair is referred to as the trigger, which may include negated events, and the second as an action. Intuitively, a transition is enabled if the environment offers all events in the trigger but not the negated ones. When a transition fires it produces the events in its action. For example $a[P]/s$ has a triggering event a, a guarding condition P on the data state of the system, and an action s involving both events and data assignments to be carried out should the transition be taken. Usually transitions can originate and terminate at any level in the hierarchy.

Figure 2.1 is an example Stateflow statechart for a stopwatch [99].

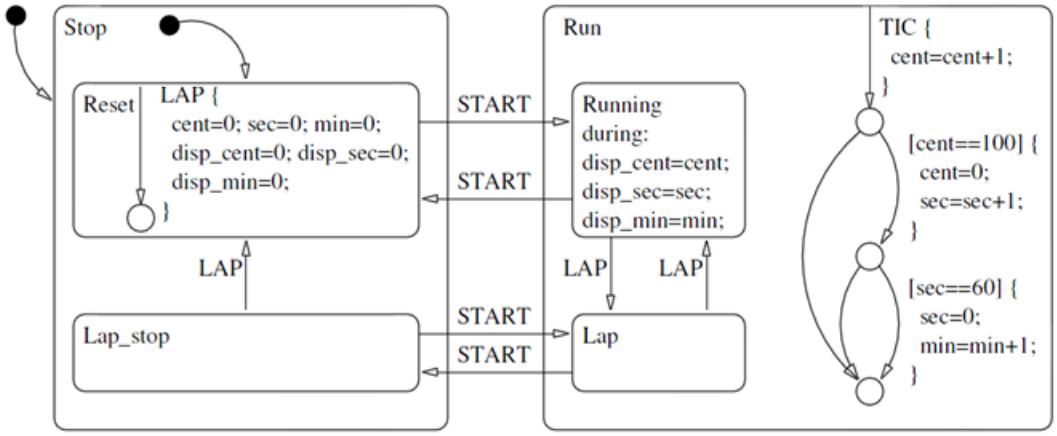

Figure 2.1: Stopwatch chart

Different dialects of statecharts support a variety of features (e.g., priority, condition and selection entrances, delays, timeouts, a history mechanism, actions and activities). There are many statechart variants which has led to a multitude of different interpretations for what diagrams mean despite their relatively common syntax. This makes it impossible to achieve a consistent interpretation.

A common point of variance in statechart semantics between different dialects relates to what is in a macro-step [205]. A macro-step is defined as all transitions within a reaction, these are observable time steps. Micro steps describe the causal chain within reactions. Every macro-step is divided into an arbitrary but finite number of micro-steps. Von der Beeck [28] compared statechart variants and the properties they supported. Von der Beeck's research showed that although most statechart dialects support a set of common properties (i.e. synchrony, hierarchy, concurrency, broadcasting, causality, etc.), some are only supported trivially. For example, typically causality is only trivially supported, (an exception to this is the Esterel language [84]) and problems are avoided in some languages (e.g., Statemate and Stateflow) by consuming emitted events in the next macro-step, or disallowing negated events (e.g. UML).

Limitations of statechart semantics also include shortcomings relating to compositionality [88, 158]. A compositional semantics is desirable as it is necessary for modular analysis of statechart designs and a component-based design process. Although the syntax for UML state machines is formally defined, the semantics is quite loose leading to possible ambiguity [89]. For real-time, safety-critical systems this is a problem. There are many approaches aimed at formalisation of UML such as ROOM [215] and Catalysis [73, 74]

which use OCL constraints to eliminate ambiguities. Although statechart semantics may be ambiguous, safety-critical systems could be modelled using a "safe" subset of state diagrams that avoid notations with an unclear meaning.

Statechart designs can become large and unwieldy, it can be argued that some parts of the design could be expressed more concisely with declarative descriptions in the form of contracts but these are not supported by the statechart design tools in common use. Statechart languages typically do not support a systematic, stepwise refinement process or a set of refinement patterns for capturing rules expressing how to refine an abstract declarative description into a fully operational design. Consistency between statechart models is not normally verified formally; each model is validated through simulation. Clearly this is not desirable or acceptable for safety-critical systems that have substantial requirements for demonstrating the absence of defects.

2.6 Tool support

The preceding sections discussed languages and theories for reactive systems. This section will discuss tool support that exists for designing reactive systems. Today, statecharts are used for the software development of complex, safety-critical systems such as the Airbus A380 and the Eurofighter Typhoon [9, 85]. Tools such as Simulink/Stateflow, within the Mathworks' tool suite [248], IBM Rational's Statemate [120] and Esterel Technologies/SCADE [84] offer specialised modelling and simulation environments. These typically include a graphical programming language for specification and design, a test suite for debugging and simulation and a code generation tool to generate program code or a hardware description (VHDL) from a model of the system.

Stateflow is an interactive design and simulation tool for event-driven systems. Stateflow extends Simulink with a design environment for developing state machines and flow charts. It is tightly integrated with MATLAB and Simulink products, providing an environment for designing embedded systems that contain control, supervisory, and mode logic. The semantics differ from that of a synchronous language such as Esterel/SCADE Safe State Machines. The term *synchronous* is overloaded and means subtly different things depending upon whether one is a hardware designer or software modeller/implementor. Similarly in different settings the term synchronous may relate to different things, for example in the context of statecharts it is a description of how time is modelled but in process algebra it describes how the parallel operator works, (i.e. in process alge-

bra synchrony means *lockstep*, while asynchrony means interleaving). In the context of this section of the thesis synchronous means models that obey the *synchrony hypothesis*. Berry's *perfect synchrony hypothesis* states that control transmission, communication, and elementary computation actions take no time [31], i.e. any responses from a stimulus that are supposed to occur at the same point in time actually do (despite the model having to perform a computation to work out what they are).

Stateflow semantics are not formally specified, only informally by the Stateflow manual. It supports a single-event, run-to-completion semantics. Stateflow run-to-completion semantics insists that if an event is broadcast, the active transitions triggered by this event are evaluated successively according to the execution order of their parent states (e.g parallel states have a predefined execution order, depending on graphical layout or user input). For each transition evaluation, new signals might be emitted by transition or condition actions. Each new event emission immediately calls this interpretation algorithm and runs to completion and only then resumes with the processing of the next transition for the original event. Exactly one event is evaluated when it occurs. Triggers with multiple concurrent events and negated events are not possible.

The two main development environments for UML-based languages are IBM's Rational Rhapsody [119] and RoseRT [118]. RoseRT implements a sublanguage of statecharts, for example, it does not support orthogonal state components. Rhapsody statecharts are object-oriented, asynchronous and intended for real-time embedded *software* systems. The action language of Rhapsody is a subset of the target programming language; so the events and actions defined along transitions and in states, etc., are fragments of C++ or Java. Model execution in Rhapsody is carried out solely by running code generated from the model, the high-level programming language is compiled down into executable code.

IBM Rational's Statemate [120] is not object-oriented and is intended for mixed hardware/software systems. Statemate has different step semantics depending on how the user configures the tool. There is a single step semantics (which Statemate calls *synchronous*), as well a micro-step/macro-step semantics that assumes the *synchrony hypothesis*. Statemate supports compound (joined with conjunction and disjunction) and negated events. Statemate has three views of a system representing its structure, functionality and behaviour. It uses activity-charts for the hierarchical functional structuring of the model. An activity-chart is an enriched kind of hierarchical dataflow diagram to represent the possible flow of information between the functions or activities. Each activity could be

associated with a controlling statechart, which would also be responsible for inter-function communication and cooperation. Statemate uses module charts to describe the structure of the system. They describe the main components in the implementation of the system and their connections. Statemate can execute statecharts directly, in an interpreter mode that is separate from the code generator.

Esterel/SCADE Suite is a development environment for safety-critical, embedded software. Esterel/SCADE has both a textual and graphical syntax (Safe State Machines). The main semantic characteristics of Safe State Machines (SSM) are synchrony and determinism. They are a reactive model of synchronous parallel systems with instantaneous broadcasting of signals. SSM allow multiple and negated events but do not permit inter-level transitions. Esterel programs are given an operational semantics as labelled transition systems and are compiled into finite state sequential machines [32]. SCADE has a graphical editor for the statechart dialect, (Safe State Machines), a graphical simulation and testing environment, a code generator and a compiler for C code production.

2.7 Contracts

The high-level declarative properties of a system during the design process are typically described in the form of a contract [171]. Dictionary.com defines a contract as, "An agreement between two or more parties for the doing or not doing of something specified". An advantage of contracts is that they can simplify a design by making it concise and more easily understood. Contracts may express the mutual rights and obligations of clients and suppliers.

	Obligations	**Benefits**
Client	Weight < 5kg pay £1.50	Delivery < 4 hours
Supplier	Delivery < 4 hours	No need to deliver unless weight < 5kg & paid £1.50

Table 2.1: Example contract for delivering a letter

Many languages support contracts (e.g. Z, JML, Eiffel) [153, 162, 172, 231]. Contracts [33, 182] describe what is required to use a component, and what benefits are obtained from its use, typically in a precise mathematical style. Design by Contract [171, 173] is

an approach to designing computer software. It prescribes that software designers should define precise, verifiable, interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of a business contract. Design by contract was first introduced by Meyer [172] with the Eiffel programming language, as an alternative to defensive programming.

Design by contract is intended to improve the reliability of reusable software components. Contracts permit the declarative descriptions of behaviour and describe how elements collaborate with each other. A contract consists of assertions (Boolean expressions) which are used to establish what a component does, clearly state service guarantees and provide parameters against which a component can be verified. Assertions can be preconditions or postconditions which apply to individual routines as well as invariants which constrain all instances of a given class.

There are different classes of contract [125]. For example, *basic* contracts are used to capture simple syntactic properties (e.g, operations that a component can perform). *Behavioural* contracts describe the permissible effects of operations similar to Eiffel's pre and post-conditions or OCL constraints [190]. *Synchronisation* contracts apply within distributed or concurrency contexts. They show synchronisations between method calls and the dependencies between services provided by a component. *Quantitative* contracts stipulate the quality of service levels such as quantifying expected behaviour or offering the means to negotiate these values (e.g. to define maximum response delay).

Contracts and concurrency is a complicated research area, with tricky issues to resolve, such as how to satisfy preconditions or determine whether postconditions have been met that depend on shared objects in a concurrent environment [179]. Contracts are a concept embraced by many formal methods. Morgan's specification statements consist of a precondition and postcondition pair [P,Q] [180]. The precondition represents a predicate over the variable state, inside which the postcondition is expected to hold. Outside P the behaviour is unconstrained (and might not even terminate). The postcondition is a *step* condition (or first-order condition) in that it can refer to variables before and after the required operation.

Jones's rely/guarantee conditions [126] are step constraints. A *step* predicate that defines what the state can "rely" upon from its environment is an effective way to anticipate how the environment operates on shared variables. Whilst Morgan's specification statements are usually thought of as a means to develop sequential rather than reactive

systems, they have been applied in a reactive setting. For example in [123, 249, 250] essentially the same mechanism is used within statecharts to express and verify assumptions on states.

In PFS [124], contracts such as this (called "current" and "next" conditions) are used to decorate the states in the machine. They scope outgoing transitions from the source state, and provide a proof goal for establishing the precondition (current condition) at the target state. Additional goals are generated vertically down the hierarchy to ensure annotations are consistent. The root node contains the abstract contract, i.e. set of conditions, on which the correctness of the whole machine rests.

Step contracts are first order, but control systems often require higher order constraints (both in pre and postcondition). Blow [42] proposed that differential extensions to the notion of a data contract for subsystems rendered them far more relevant to the control systems domain. The idea advocated was to extend contracts so that both preconditions and postconditions could range over arbitrary subsequences of inputs and outputs. These were represented symbolically by allowing them to refer to arbitrarily prime decorated variables rather than just the usual unprimed and single-primed variables. Differential contracts constrain arbitrary sub-sequences of input and output. A limitation of differential contracts is their inability to constrain entire state histories in a general way. Temporal logic supports general constraints on state histories.

Temporal logic can be linear time or branching time. Linear time logic supports the view that at each moment there is only one possible future and temporal modalities are provided for describing events along a single time path. Whereas branching time supports the view that at each moment there may be different possible futures and the modalities reflect the branching nature of time by allowing quantification over possible futures [81,82]. The features present in the logic are of primary importance such as the ability to specify nondeterminism and the presence of a negation operator to express what outcomes are not possible, as well as those that are.

Hennessy-Milner logic/μ-calculus is a multimodal logic used to specify properties of labelled transition systems. A theoretical background to contracts based on temporal logic is offered by Lüttgen and Cleveland's work on mixing temporal and process algebraic styles of specification [64] and Lüttgen and Vogler's work on the compositional operators and refinement orderings needed to support such a mixture of styles [157].

2.8 Patterns

Patterns and pattern languages in software engineering are ways to describe best practices, and capture experience so that it is possible for others to reuse this experience [10]. They document and support sound engineering architecture and design. The goal of patterns in software engineering is to help developers resolve recurring problems. They communicate insight and experience about these problems and codify their solutions.

There are different types of patterns. For example, a design pattern is a general reusable solution, at the level of modules and interconnections, to a commonly occurring problem in software design [97]. An architectural pattern in software is a standard design in the field of software architecture so it has a broader scope than a software design pattern, and addresses a wider range of issues such as computer hardware performance and minimisation of business risk [50].

Analysis patterns in software engineering are conceptual models, which capture an abstraction of a scenario that can often be encountered in modelling. An analysis pattern reflects conceptual structures of processes rather than actual software implementations. It can be represented as a group of related, meta-classes with attributes, behaviours and expected interactions defined in a domain-neutral manner [92].

2.8.1 Design patterns

Design patterns abstract reusable design experience and address system design issues [97]. They capture expert knowledge and design trade-offs [208]. Design patterns provide desirable software engineering benefits [125, 149, 197] such as modularisation, encapsulation, reuse, extensibility, clarity and maintainability. Patterns express design structures, i.e., they are building blocks for constructing more complex designs, and can help implement specific architectures and mechanisms present in the problem domain. The refinement and refactoring patterns for this research are conceptually similar to design patterns. Design patterns however only focus on guiding the process of implementation and not on verifiably correct transformations.

Design patterns generally consist of an abstract description of class/object collaborations and their responsibilities and determine the circumstances under which the pattern is applicable and the changes to be made. There are different classifications and catalogues of design patterns and many share some commonality. Patterns have been classified as compound, creational, structural or behavioural [97].

There are numerous design patterns for real-time, reactive and concurrent systems. Powel-Douglass [72] has identified many for UML. Some are based on collaborations of states or are for implementing certain behaviours, for example relating to task priorities and synchronisation. There has been research into design patterns for statecharts [243] with the aim of facilitating the reuse of object-oriented statechart implementations for the deployment of basic and hierarchical statecharts, optionally supporting orthogonality, broadcasting and history.

Bordeleau *et al* [44] have identified design patterns for state machine implementation. Unlike Douglass's patterns that offer reuse of state machines at the level of a class and its roles, these patterns relate to design decisions at the level of scenarios. They focus on the transition between scenario models and hierarchical state machines. Their *Uninterruptable Sequence of Actions* pattern uses deferred events to ensure while executing the sequence of actions in the composite state that all irrelevant messages are ignored. This pattern could be used to implement a critical section. Another example is their *Coregion* pattern which implements a set of messages whose temporal ordering is undefined. The pattern avoids a naive solution that leads to a combinatorial explosion of states. To achieve this it relies on state variables.

2.9 Refactoring

A refactoring is a behaviour-preserving transformation on an artifact constructed during system engineering. A refactoring is a horizontal transformation in the sense that the level of abstraction and thus external behaviour stays the same. The key purpose of refactoring is to clean up code separately from adding new functionality, using common refactoring methods. In Extreme Programming and other agile methodologies, refactoring is an integral part of the software development cycle, to improve the internal structure and clarity of code [27]. Fowler's refactorings [93] provide examples of evolutionary changes to object-oriented systems, (e.g. operational refactorings focus on restructuring behaviour). For example,

1. The *Encapsulate Field* refactoring is a technique that allows for more abstraction by forcing code to access a field with getter and setter methods.

2. The *Extract Class* refactoring is a technique for breaking code apart into more logical pieces by moving part of the code from an existing class into a new class.

More recently some research has proposed raising the level of abstraction from program refactoring to model refactoring which aims to apply refactoring techniques at the model level (e.g., UML models), rather than to source code [247, 253]. The refactoring and refinement patterns for this thesis are described in Chapters 4 and 5. They are model transformations to perform refactorings on Contractual State Machine designs.

2.10 Refinement

Refinement supports the construction of correct programs from their specifications through a sequence of steps. Stepwise refinement consists of developing a design through different levels of abstraction, from a high level to a more detailed design [18,51,54,58,255,256,262]. Iterative refinements are performed, producing a design in which the behaviour is fully described, which is consistent with the initial more abstract description. In general refinement is a transformation in a vertical direction from abstract to concrete. It is a relationship that preserves desirable properties. The properties may be varied such as simulation, equivalence, understandability, performance, or hierarchy. Refinement patterns are designed to maintain these properties. When such patterns are fully formal they are usually referred to as *laws*.

According to [57] there are two approaches to refinement. The verification approach to refinement allows the practitioner to offer a possible refinement of a program, and tools are applied to prove the correctness. On the other hand the calculational approach to refinement uses laws that may have associated proof obligations to guide the refinement process and prove that the new more detailed design refines the more abstract description. The transformation process from specification through to implementation consists of a series of correctness preserving refinements. The design is gradually elaborated in a stepwise fashion as the result of incremental manipulation of the specification/design using refinement laws. These may alter internal representation as well as introduce executable constructs. On completion both the design and its proof of correctness with respect to the original specification have been achieved.

The refinement calculi of Back and Morgan [17, 180] represent a typical framework of how to achieve formal software development through refinement. Central to this approach is the notion of a formal specification statement and the formal definition of the refinement relation. Formal specification statements are regarded as particular forms of programs and (non-executable) program statements in their own right. The refinement laws are

pre-proven to uphold the refinement relation between the abstract and more concrete designs. The correctness of the refinement process can be established by showing that the refinement relation is preserved all the way through the design process.

2.11 Model-driven engineering (MDE)

Model-driven engineering (MDE) is based on the idea of a highly automated environment for the specification, elaboration and management of models throughout the engineering lifecycle [16, 225]. MDE is a state of the art approach to software development that attempts to raise the level of abstraction at which software and system engineers carry out their tasks beyond the use of programming languages and components. To achieve this models are promoted to first-class artefacts in the development process [34]. They are precisely defined and successively elaborated until they are complete, then finally they are automatically transformed into an implemented system.

MDE automates the full range of model management tasks such as model transformation, validation, comparison and merging. In an MDE workspace the models are typically interrelated, possibly overlapping, and usually expressed using different modelling languages. MDE transformations can support refinement and refactoring.

MDE is not a development method or process, it can be implemented in a number of ways (e.g. Extreme Programming (XP) or by a refinement calculus [15, 160]). The Model-Driven Architecture (MDA) [135, 164, 165] is an initiative of the Object Modelling Group (OMG), aimed at providing a standard approach for MDE. MDA requires the use of a standard modelling language, UML and meta-steps that should be followed in the development of models and systems. MDA attempts to achieve the benefits of MDE while also improving interoperability to enable different projects to share models and standardizing the system development process.

The Object Constraint Language (OCL) [190] is a language for defining constraints on metamodels such as Meta-Object Facility (MOF) [193] and UML models. Constraints are expressed as invariants on classes and tested against class instances. There are a number of open source tools that facilitate evaluation of OCL invariants such as Object Constraint Language Environment (OCLE) [151] and Octopus [232]. An OCL invariant is restricted to expressing constraints on a single model, and thus can only achieve intra-model consistency or internal model consistency to check a model conforms to its metamodel and does not contain any contradictions. OCL does not support inter-model consistency (external

model consistency) which could be used to check consistency between models that capture different but overlapping aspects of the problem domain or to ensure consistency between models at different levels of abstraction.

Model transformation is a core concept in model-driven engineering (MDE) [68, 69, 134, 169, 226]. It is the automatic production of target models from source models based on a transformation definition, usually consisting of a set of transformation rules. A transformation rule is a description of how one or more constructs in the source languages can be transformed into one or more constructs in the target languages.

There are a wide range of different types of transformations in MDE, including model-to-model and model-to-text. Two specific forms of model-to-model transformations of relevance to the research presented in this thesis are horizontal and vertical transformations. Horizontal transformations are typically characterised by a change to the internal structure of software to improve certain software quality characteristics such as understandability, modifiability, reusability, modularity and adaptability without changing observable behaviour. A horizontal transformation is a transformation where the source and target models reside at the same abstraction level. A typical example is refactoring, which is a horizontal transformation where the source and target languages are the same [37, 167].

A vertical transformation is a transformation where the source and target models reside at different levels of abstraction. A typical example is refinement. Formal refinement in the sense of [221, 224] (which is defined in terms of reducing nondeterminism) is vertical and within a single language (i.e., endogenous), Model transformations preserve properties of the source models. In particular, refactorings preserve the external behaviour, while the structure is modified. By contrast, refinements preserve correctness [252].

Two critical types of model transformation are the *mapping* transformation which translates a number of source models into a number of target models and does not alter the source models. The other type is the *update-in-place* model transformation [69] which by contrast performs modifications to the source model itself. The effects of the transformation are visible while performing the transformation. They can be applied in a user-driven manner on model elements that have been selected by the user, (update-in-place transformations) [140]. This approach can support refactoring and refinement transformations, and in turn, patterns.

An essential part of an MDE process is to transform models into textual artefacts (model-to-text transformation). For the research reported here the tool suite for Con-

tractual State Machines provides model-to-text transformation to translate a graphical Contractual State Machine design into the STGA-like language required by the model checker (HST).

2.12 Eclipse-based development tools for MDE

This section discusses Eclipse development tools for MDE. The Eclipse Foundation [78] is an organisation formed by major software vendors, and supported by a large and active development and user community. Its main purpose is to provide an open platform for integrating interoperable development tools. Eclipse has a modular architecture consisting of a small core, its Rich Client Platform (RCP), and extension mechanisms that enable developers to contribute additional functionality in the form of plugins which are grouped by features and products.

A significant body of tools have been developed as plugins for Eclipse such as editors, compilers and launchers for many programming languages, management tools for source code repositories and most importantly, modelling tools. Productivity and quality benefits can be achieved as Eclipse offers a significant amount of stable and tested functionality for reuse. Eclipse supports model-driven engineering based on the OMG standard model-driven architecture (MDA) [135, 165, 175, 176, 194]. The Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework (GMF) support development of model-view-controller based graphical modelling tools, and were used to implement the graphical model editor for Contractual State Machines.

2.12.1 Eclipse Modelling Framework (EMF)

EMF is a modelling framework and code generation facility for building tools based on a structured data model [77]. It is a semi-formal object oriented modelling technology, and it can be used to define the underlying model (domain model) of visual editors. Metamodels describe the abstract syntax, symbol and link types (relationships) used in the diagram of the visual language and do not contain information about their concrete layout. Additional language constraints can be expressed by adding OCL constraints [190] to the EMF model. EMF provides a graphical editor for specifying metamodels with its Ecore M3 language. EMF generates executable code from metamodels. Given an EMF model, a set of Java classes and a set of adaptor classes that enable viewing and command-based

editing of the model can be generated. A basic, tree-based editor for model instances can be generated and directly executed in the Eclipse run-time workbench. EMF also provides a framework (EMF.edit) for generating new modelling languages (graphical editors) from a metamodel. EMF is aligned to MOF 2.0 [193] supporting model validation with OCL and XML Metadata Interchange (XMI) for interchange and serialization of models [192].

2.12.2 Graphical Editor Framework (GEF)

The Graphical Editor Framework (GEF) provides technology to develop graphical editors for the Eclipse workbench UI [76]. Basic and advanced editor functionalities are supported. GEF applications have a model-view-controller architecture. GEF provides facilities for loading a graphical model, creating controllers for each model element and constructing and associating views with the controllers. Figures for the concrete layout of diagrams in the graphical editor and commands to be used in the editor are specified, alternatively a graphical editor can be generated using GMF.

2.12.3 Graphical Modelling Framework (GMF)

GMF Tooling [75] provides a set of generative components and runtime infrastructures for developing graphical editors/modelling tools in Eclipse, based on EMF and GEF. It is a model-driven approach for creating a graphical editing surface for any domain model in EMF. It links a GEF diagram definition to an EMF domain model. GMF is used for concrete syntax development. It enables the definition of graphical syntaxes for metamodels by attaching visual elements such as lines and polygons to metamodel elements. Additional language constraints can be applied using OCL. GMF generated editors offer basic editing commands to create, edit, move and delete single model elements. GMF and EMF allow a developer to get started quickly when producing a domain specific modelling tool, however a drawback is that typically it is necessary to fine-tune or customise the generated software and there may be a high cost associated with maintaining the software as requirements change.

2.12.4 EuGENia

EuGENia is a front-end for GMF, its aim is to speed up the process of developing a GMF editor [83] and it is based on the principles of model transformation. It enables developers to generate a fully functional GMF editor by defining a few high-level annotations in

the Ecore metamodel. It automatically generates the .gmfgraph, .gmftool and .gmfmap models needed to implement a GMF editor.

2.12.5 Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [136, 143], is a suite of tools for MDE, built on top of the Eclipse platform. Epsilon provides a set of interoperable task-specific model management languages that enable composition and seamless integration of individual model management tasks as well as non-MDE tasks into coherent workflows. The family of model management languages are built on top of an extensible platform of reusable core facilities.

Epsilon supports a common model repository to provide centralised management of loading and storing models. This centralisation alleviates the resource and time consuming process of repeatedly loading and storing the same models when more than one model management task is to be performed on them. Epsilon provides an inter-task communication facility which enables different model management programs to communicate at runtime by exporting variables to enable the results of a complex query, for example, to be exported instead of being recalculated by subsequent tasks.

The architecture of Epsilon is shown in Figure 2.2. The Epsilon Model Connectivity layer provides the abstractions that free Epsilon from tie-in to a specific metamodelling technology. The core of the platform is the Epsilon Object Language (EOL) [139], a reworking and extension of OCL that provides support for model update, conditional and loop statements, statement sequencing, and access to standard output and error streams. The Epsilon task-specific languages are built on top of EOL, giving highly efficient inheritance and reuse of features.

The task-specific languages are, the Epsilon Object Language (EOL) [139], for the direct manipulation of models, the Epsilon Merging Language (EML) [137], for model merging, the Epsilon Comparison Language (ECL) [138], for model comparison, the Epsilon Transformation Language (ETL) [141], for model-to-model transformation, the Epsilon Validation Language (EVL) [91], for model validation, the Epsilon Generation Language (EGL) [212], for model-to-text transformation, and the Epsilon Wizard Language (EWL) [144], for in-place model transformations. In contrast to most other model management tools like XPand and MOFScript, which are standalone or only loosely integrated, all the languages of the Epsilon platform build on a common model navigation and manip-

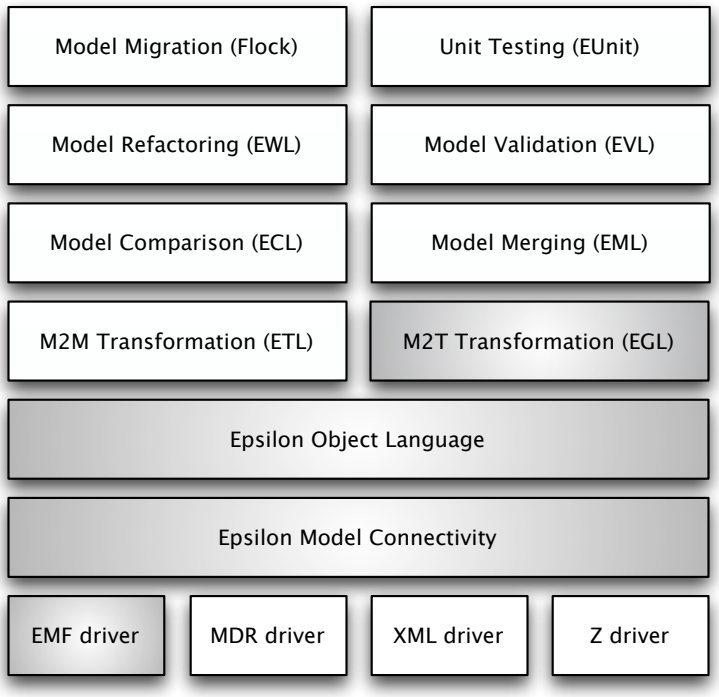

Figure 2.2: The architecture of Epsilon

ulation language (EOL) and run-time environment and are therefore interoperable, (i.e. an operation defined using EOL can be reused, without changes, by any of the Epsilon languages).

Epsilon supports a wider range of model management tasks than comparable platforms, and provides better integration due to the common infrastructure on which the languages have been built. There are other differences and advantages of Epsilon over comparable model management tools, for example, for each model manipulation task, the language is tailored to the task's specific requirements, unlike the Atlas Transformation Language (ATL) [128, 129], which uses a single language for all tasks. Epsilon languages provide excellent Eclipse-based tools which are supported by stable execution engines [142]. The differences between Epsilon and openArchitectureWare (oAW) [196] are that oAW is a proprietary framework for coordinating model management tasks, and the range of supported model management tasks is different to Epsilon. OAW does not support model comparison and merging and Epsilon does not support text-to-model transformation tasks.

2.12.5.1 Epsilon Wizard Language (EWL)

The Epsilon Wizard Language (EWL) is a language designed for performing interactive, in-place model transformations in the form of wizards that can be coupled loosely with arbitrary model-view-controller GUIs [140]. EWL plays a key role in this thesis being used to create wizards for the refinement and refactoring transformations proposed in Chapters 4 and 5. It is important for the catalogue of wizards to be extendable. EWL will allow additional wizards to be implemented for refinement and refactoring patterns tailored to needs not yet met. A wizard specifies the types of elements to which it applies and defines the actions it will perform when it is applied to a selection of model elements.
An EWL wizard defines a *name*, a *title* part, a *guard* part, and a *do* part.

wizard name {
guard ⟨*predicate*⟩
title ⟨*title*⟩
do {
⟨*transformation*⟩
}
}

The *name* of a wizard acts as an identifier (for the software engineer implementing wizards) and the *title* is the name of the wizard or a short description of its functionality (e.g. "Unfold Always") that acts as an identifier for the user when selecting wizards to apply to a design. A wizard defines the actions it will perform when it is applied to a selection of model elements in the *do* part. The guard is a predicate that specifies when a wizard is applicable to a specific selection of model elements, the approach here is a declarative one. The *title*, *guard* and *do* parts are expressed in the Epsilon Object Language (EOL). EOL can express simple declarative statements or blocks of imperative statements. It is an OCL-based language tailored to navigating, querying and modifying models and provides mechanisms for capturing user input [139].

The process of executing EWL wizards is user-driven. The EWL interpreter filters applicable patterns at design time. The user selects model elements in the modelling tool and each time the selection of model elements changes the guards of all wizards are evaluated by the EWL interpreter. If the guard of a wizard is satisfied the title part is evaluated (the EOL statement(s) are executed) and a string is returned. The string acts as an identifier for the wizard and is added to the list of applicable and available wizards

that is presented to the user in a pop-up menu. Filtering out irrelevant wizards reduces confusion and enhances usability particularly as the list of available wizards increases in size. Thus the user is only ever provided with the list of wizards that are applicable to the current selection of elements. The user can select one from the list and the interpreter executes the *do* part of the wizard to perform the intended transformation.

EWL successfully hides complexity from the software developer. Initially the engineer is required to get to grips with the UML2 metamodel, but this is an expected requirement for standardised model management languages. EWL is a very succinct language resulting in simpler code that is easy to understand and maintain.

There is one issue that may be seen as a possible drawback to EWL. It may be argued that currently EWL has low modularity and reusability as there is no easy way to combine patterns other than copying and adapting the code of existing wizards [247]. This is particularly restrictive in the context of refining and refactoring models where compound patterns build upon and reuse core patterns. Although EWL does not currently support reusability of wizards it does support reusability of operations which can be grouped into external libraries, as it is built on top of the Epsilon Object Language (EOL) layer.

2.12.5.2 Epsilon Generation Language (EGL)

The Epsilon generation language (EGL) is a model-to-text transformation language [212]. It is a model-driven, template-based code generator. EGL uses a template to generate text from an instance of an Ecore metamodel. Templates are constructed from sections. The contents of static sections appear verbatim in the generated text. Dynamic sections contain executable code used to control the generated text. In its dynamic sections EGL reuses EOL's mechanisms for program control flow, model inspection and navigation and defining custom operations. EGL could be used to translate a state machine design to a description of the model in a textual representation required by external systems. For example an EGL template has been implemented (as part of the software development for this thesis, see Chapter 6) to translate a state machine design into the STGA-like language required by the model checker, the Heterogeneous Specification Tool (HST). This could then be passed to the model checker for further processing.

2.13 Contractual State Machines (CoSta)

This section introduces the Contractual State Machine language (CoSta), its key syntactic constructs and an overview of the semantics (further details are given in Appendix B). The focus is on presenting Contractual State Machines from the perspective of an engineer and thus emphasises how the language is to be used rather than details of its semantics. The formal framework and refinement theory for Contractual State Machines was developed by other members of the group and remains unpublished. The formal foundations are not a contribution of the thesis but are summarised for completeness as they set the context for the research presented in this thesis.

CoSta's refinement relation and the language of contracts[1] is based on Lüttgen and Voglers' work on Logic LTS [157], which itself is derived from Ready Simulation [41]. It is a heterogeneous specification/design language to describe reactive systems in both declarative and operational styles. Contractual State Machines (CoSta) is a simple hierarchical state machine language that has been extended to incorporate a language based on a restricted form of μ-calculus (contracts). CoSta is an open language with shared variables.

It is envisaged that at the initial design stage before CoSta is used to specify open contracts, a top-level contract that is based on closed reasoning (the variables of the model are impervious to outside interference) and expresses model-wide properties may be specified. A closed contract language has not yet been defined. It is intended that the closed contract language will express properties in a temporal logic based on μ-calculus extended with data parameters for describing events and state. Subsequent design stages would implement the closed contract as a set of open CoSta contracts ensuring that the parallel composition of their interpretation as processes satisfies the closed contract.

The CoSta contract language supports the specification of high-level properties in an abstract and concise way. Contracts express very complex (highly nondeterministic) state machines succinctly. The syntactic structure of CoSta State Machines is based on Lin's Symbolic Transition Graphs with Assignment (STGA) [71, 155]. STGA are LTS's with richer labels consisting of a guarded action with assignment. The semantics is currently given as a "ground semantics" based on Labelled Transition Systems (LTS), where an LTS

[1] Full formal details of the refinement relation are not yet in the public domain; this thesis does not rely on these formal details.

is defined in the usual way for CCS-like languages as labelled directed graphs. Output actions in CoSta are primed as in CCS. When channels do not involve data the distinction between input and output actions is more intuitive. Actions which "report" information to the environment are primed, and those which "request" information from the environment are unprimed.

The notion of correctness preservation for CoSta is provided by a refinement relation, shared variable ready simulation on STGA/LTS transition systems. It holds if the behaviours of two systems are identical except for the reduction of nondeterminism at every step no matter what data state the environment places them in. It is compositional over shared-variable (i.e. statecharts-like) parallel and hierarchy operators. State is included up front in the language, and no refinement of the state will be conducted. There is no primary notion of equivalence for CoSta only a notion of refinement ordering in which the hidden actions are closed (effectively ignored). Two processes are equivalent when they refine one another. Equivalent processes both satisfy the same contracts.

The CoSta contract language has a dual interpretation. Logics, such as the one employed for the CoSta contract language, are usually related to LTSs by a satisfaction relation. However, following [157] contracts are also given a direct characterisation as a single STGA (LTS), allowing mixed descriptive styles within the same heterogeneous framework. The operators of the CoSta contract language are carefully chosen, so that a maximal characterising STGA (maximal agent) can be constructed, whose refinements are precisely those processes satisfying the formula, (any refinement of a process satisfying a contract also satisfies that contract). A maximal agent for each contract is the nondeterministic choice over all the processes that satisfy it. It is the largest LTS with least nondeterminism that the contract fully characterises (that satisfies the contract). It is effectively the nondeterministic choice between all behaviours that satisfy the contract.

The Heterogeneous Specification Tool (HST) is a model checker under development within the research group to provide automated refinement checks, and expand the contract notation into its dual interpretation. The tool (HST) currently provides animation facilities based on STGA agents and a data environment. It provides a predicate-style interface to an SMT solver (Boolector [49]) for deciding conjectures.

2.13.1 CoSta contract language

The CoSta contract language is based on a restricted form of μ-calculus (some μ-calculus operators cannot be defined within the heterogeneous framework adopted).

Formula	=	"True" | "False" | BasicForm |
		Formula "$\vee$" Formula | Formula "$\wedge$" Formula
BasicForm	=	TotalisedEnable | Disable | ExclusivelyEnable
		| IfAction | Always | Unless | BoundedEventually
TotalisedEnable	=	"$\langle$[" EventExpr "]$\rangle$"
Disable	=	"$\langle$" EventExpr "$\rangle$"
ExclusivelyEnable	=	"$\underline{\langle}$" EventExpr "$\underline{\rangle}$"
IfAction	=	"[" EventExpr "]" Formula
Always	=	"$\Box$" Formula
Unless	=	Formula "$\curvearrowright$" Formula
BoundedEventually	=	"$\Diamond_k$" Formula
EventExpr	=	EventSet | EventSet "[" VariableUpdate "]"
EventSet	=	Eventlist | "-" Eventlist | _
Eventlist	=	Event | Event "," Eventlist
VariableUpdate	=	Assignment | Guarded_Assignment
		| Establish_Assignment
Assignment	=	Varlist "=" Exprlist
Guarded_Assignment	=	Predicate "$\Longrightarrow$" Assignment
Establish_Assignment	=	Varlist ":" Predicate

The contract language has been extended with data (in a manner similar to Cleaveland's data-extended μ-calculus [64]), so that actions can have guards, and assignments and nondeterministic choice for assignment is supported. The operators of the contract language are based on those proposed by Lüttgen and Vogler [157].

- The *totalisedenable* operator specifies that one or more events in a set must be available. When accompanied by a variable update the event is enabled from every before state within the guard (or feasibility) of the assignment and must be capable of a consistent update.
- The *disable* operator specifies that none of the events in a set are available. When accompanied by a variable update the action is disabled for all the updates consistent with the expression. It may still be enabled within its guard as long as its update is not consistent.

- The *exclusivelyenable* operator specifies that one of the events in a set is exclusively available, whilst all other events are disabled. When accompanied by a variable update the event is only exclusively enabled within the guard (or feasibility) of the assignment and must be capable of a consistent update.

- The *ifaction* operator specifies constraints on the behaviour following a particular set of events (whatever the values of the variables, and for all events in the set). When accompanied by a variable update, the constraints need only apply from a before state satisfying the guard (or feasibility) of the assignment, and following a consistent update.

- The *always* operator specifies a constraint which must be true at all times, whichever events are taken.

- The *unless* operator behaves similarly to the always operator: the left-hand formula must be true at all times. However should the right-hand formula become true, the left hand formula (at this point and thereafter) no longer need hold.

- The *boundedeventually* operator specifies a constraint which must be true within k steps.

- The *Assignment* represents the simultaneous update of the variable on the left with the expressions on the right. The *Guarded Assignment* carries an additional enabling condition. The *Establish Assignment* is nondeterministic. It sets the variables in such a way as to satisfy the *predicate* on the right of the ":". The values of the variables before and after the assignment are distinguished by unprimed and primed variables respectively. The *predicate* can contain instances of any unprimed variable but *varlist* can only contain instances of primed variables. If no assignment can satisfy the *predicate* from a particular set of before-values then it is said to be infeasible (from those values).

The CoSta contract language can describe properties that are required to hold on some branch and properties that are required to hold on every branch. Conjunction ($\wedge$) specifies that more than one property is required to hold. A choice of behaviours is expressed with disjunction ($\vee$), and the weakest process in these terms is *True*. Only processes inconsistent from all states satisfy *False*.

Many of the operators are defined by translation into more fundamental forms. For example $\langle e_1 \dots e_n[A] \rangle$ is defined as $\langle e_1[A] \rangle \vee \dots \vee \langle e_n[A] \rangle$, $\langle e \rangle$ is defined as $\langle e[\overrightarrow{x} : True] \rangle$, $\overline{\langle a \rangle}$ is defined as $[a]False$, and $\Box L$ is defined as $L \curvearrowright False$.

2.13.2 Abstract syntax of Contractual State Machines

The Contractual State Machine language supports a common subset of statechart language features. Its syntactic model (figure 2.3), which was designed during the implementation of the Contractual State Machine design tool (described in Chapter 6), is included here for clarity. The metamodels for the contract and transition label languages are discussed further in Chapter 6. A full listing of the concrete syntax (described in an EBNF-based language) and abstract syntax (described in Emfatic) for the CoSta contract language is given in Appendix A.2 and for the transition label language in Appendix A.3.

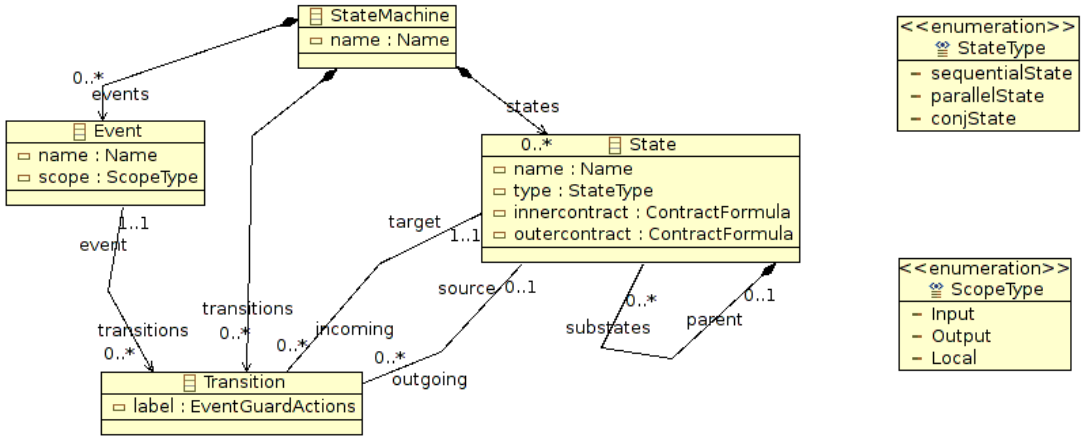

Figure 2.3: Contractual State Machine syntactic model

Primitive diagrams, comprising *OR* states and transitions, are interpreted as pictorial representations of STGAs. STGA does not allow nondeterministic assignment but CoSta extends STGA with nondeterministic assignment, based on the Generalised Substitution Language's [7] unbounded choice. Hierarchy is captured by a semi-static "interrupt" operator and captures the meaning of statecharts hierarchy, (composes the outer transitions and their destinations with the inner transitions and their destinations).

State types signify how the substates of a superstate are treated: sequential (OR) states and parallel (AND) states represent sequencing and orthogonality respectively. The parallel operator is a shared-variable version of CCS's two-way synchronising operator. It

was provisionally chosen for simplicity, with a view to adding complexity later on. CCS's restriction operator was also adopted, in order to enforce internal (hidden) synchronisation. The (shared-variable) CCS parallel operator was used in the case study, but at the top level, before the patterns apply. The refinement relation (SVRS) was designed to be compositional with respect to it.

Conjunction states are an addition and a new form of state decomposition required to support the approach to the design process. Conjoined sub-states are orthogonal in the same way as parallel states. Conjunction states represent the behaviour *common* to all their substates, (the intersection of the behaviours). Its definition is derived from that of [157]. They are not part of the target design notation[2].

The conjunction operator is a solution to the problem of dealing with conjunctions in the contract language, where the form of the conjuncts is critical in determining which refinements are permissible. Providing patterns directly for each form would prove impossible. A different approach is advocated where each conjunct is refined separately into a Contractual State Machine using patterns. The aim is to refine each conjunct separately to a behaviour common to both, i.e into the same Contractual State Machine, then merge the identical conjunct states into one.

The heterogeneous design language mixes contracts and state machines in a restricted and disciplined way. Contracts specify non-decomposed designs and can appear on any node in the hierarchy. The syntactic structure presented above distinguishes two uses of the contract notation associated with states, *inner* contracts and *outer* contracts. An inner contract is the behaviour that hasn't been decomposed yet and is potentially a succinct way of describing a highly nondeterministic diagram within the state. It is the declarative specification of the behaviour *within* a state and ignores the behaviour outside the state such as outgoing transitions or composed states. An outer contract specifies behaviour of a machine from that state onwards. It ignores behaviour outside the containing state but includes the outgoing transitions of the state to which it is attached.

If a state has no outgoing transitions then its inner and outer contracts characterise the same behaviour although the contracts may be at different levels of abstraction. The inner contract could be equivalent to or a refinement of the outer contract. An inner contract is not necessarily a refinement of an outer contract attached to the same state,

[2]The intention is that conjunction states will be eliminated before the final design is reached

as the outer contract also characterises the outgoing behaviour of the state. However, the inner and outer contracts must be consistent with one another. For example, an outer contract must not say a particular action is disabled, whilst the inner contract says the same action is enabled. Consistency is ensured by the patterns that manage the contracts.

Outer contracts do not specify behaviour in the same way as inner contracts; they keep track of what contracts have already been satisfied and specify behaviour of a machine from that state onwards. They should be viewed as labelling annotations rather than specifications with a record of where the behaviour came from. They are important for applying certain patterns, principally those to introduce cycles into the state machine. Note that the syntax is slightly more liberal than is strictly required by the refinement process (e.g. an arbitrary state need have neither an incoming nor outgoing transition). This allows conventional use of the tool to construct models free-hand as well as via the refinement process.

2.13.3 Transition syntax

Currently, the transition syntax is based, for simplicity, on that of the underlying semantic objects, STGA. The intention is to extend/modify the syntax and its semantic interpretation in the future to capture recognisable dialects of statecharts. The transition syntax has two elements: an event and an action. Actions represent operations on the data, (updates on the variables). They correspond to relations, which are applied to the current data state, choosing nondeterministically where there are multiple outcomes. Event reception and emission are kept separate based on the choice of parallel operator, therefore events are distinguished as either input events or output events (primed).

Label	=	Event “[” Action “]”
Event	=	EventName | “ ’ ” Eventname
Action	=	Assignment | Guarded_Assignment
		| Establish_Assignment
Assignment	=	Varlist “=” Exprlist
Guarded_Assignment	=	Predicate “$\Longrightarrow$” Assignment
Establish_Assignment	=	Varlist “:” Predicate

The action part of the transition provides the enabling condition for the transition, (the guard which is sometimes implicit) and the corresponding update on the variables, which occurs when the transition is taken. The Action syntax is based on B’s generalised sub-

stitution language [2], but restricted to the expressions and predicates. The language is restricted to the constructs supported by HST, which in turn are determined by those supported by the SMT solver (Boolector).

The *Assignment* represents the simultaneous update of the variable on the left with the expressions on the right. *Guarded Assignments* carry an additional enabling condition, the transition is disabled unless the current variable values make the condition true. *Establish Assignments* are nondeterministic assignments. They set the variables in such a way as to satisfy the *predicate* on the right of the ":". The values of the variables before and after the assignment are distinguished by unprimed and primed variables respectively. The *predicate* can contain instances of any unprimed variable but *varlist* can only contain instances of primed variables. If no assignment can satisfy the *predicate* from a particular set of before-values then it is said to be infeasible (from those values) and the transition is disabled. [3]

2.13.4 Example CoSta model

To briefly summarise, CoSta is a simple hierarchical state machine language that incorporates a language based on a restricted form of μ-calculus (contracts). The contract language has been extended with data so that actions can have guards and assignments with nondeterministic choice. Primitive diagrams, comprising hierarchy, *OR* states and transitions, are interpreted as pictorial representations of Contractual State Machines. State types signify how the substates of a superstate are treated: sequential *OR* states and parallel *AND* states represent sequencing and orthogonality respectively. Conjunction states represent the behaviour *common* to all of their substates, (the intersection of the behaviours). They are required to support the approach to the design process, and are not part of the target design notation. The transition syntax has two elements, an event and an action. The action part of the transition provides the enabling condition for the transition (the guard which is sometimes implicit) and the corresponding update on the variables, which occurs when the transition is taken.

[3] Predicates are constructed from True, False, &&, ||, !, ==, ! =, >, >=, <=, <

To illustrate some of the possible constructs of the language, figure 2.4 shows an example CoSta model (partially designed). The model shows the different types of states that may occur (e.g. substates, start states, composite states, conjunction states, states with unelaborated inner contracts), transitions and contracts (inner and outer).

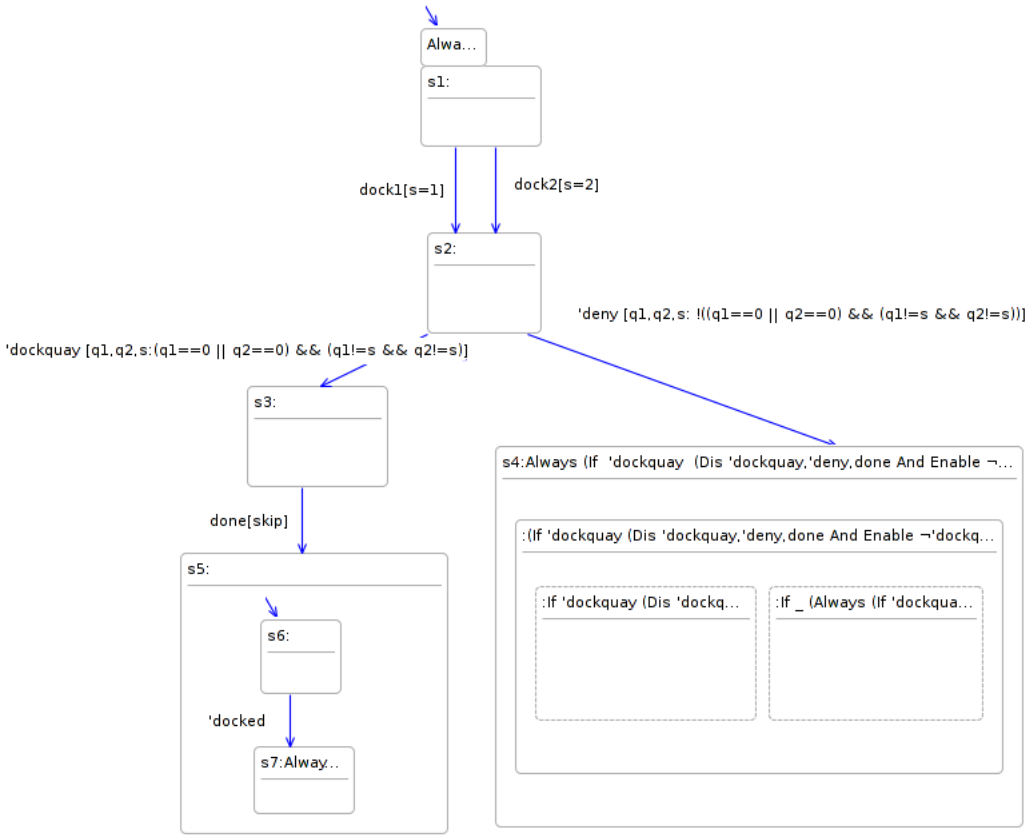

Figure 2.4: Example Contractual State Machine model

2.13.5 CoSta's refinement relation

This section gives an overview of CoSta's refinement relation. The purpose is to aid understanding of the refinement and refactoring patterns that are presented later on in Chapters 4 and 5, and clearly delineate the research contributions for this thesis from the work of the group as a whole.

The notion of correctness preservation is provided by a compositional refinement relation. The theory of refinement is based on *logic LTS* and its associated derived (tau-closed) transition system [157], which itself is derived from Ready Simulation [41].

Definition of Ready Simulation

$E \sqsubseteq_{rs} F \Leftrightarrow$

$(\forall a.(E \xrightarrow{a}\Rightarrow F \xrightarrow{a}))\wedge$

$(\forall a, F'.(F \xrightarrow{a} F' \Rightarrow \exists E'.(E \xrightarrow{a} E' \wedge E' \sqsubseteq_{rs} F')))$

Intuitively E and F have the same ready sets, (permissible actions), and if F performs an a then there must be a consistent transition in E (leading to respective states which are also ordered by Ready Simulation). Ready Simulation holds if the behaviours of two systems are identical except for the reduction of nondeterminism at every step. Ready Simulation preserves the availability of actions whilst reducing nondeterminism.

Shared Variable Ready Simulation (SVRS) extends Ready Simulation to incorporate shared variables since ready simulation only deals with the reachable state space and fails to deal with situations relevant to shared variable parallel composition. The notion of ready simulation is generalised into an ordering which ensures monotonicity of shared variable parallel composition (under any initialisation). The new ordering must consider the unreachable parts of the state space of a component as well as the reachable. It must take into consideration all variable states reachable or otherwise, as the environment we place any component in parallel with, may interfere with the variable space in such a way as to make such states relevant to the composition.

Definition of Shared Variable Ready Simulation

$E \sqsubseteq_{svrs} F \Leftrightarrow$

$(\forall a, \sigma.((\sigma, E) \xrightarrow{a}\Rightarrow (\sigma, F) \xrightarrow{a}))\wedge$

$(\forall a, F', \sigma, \sigma'.((\sigma, F) \xrightarrow{a} (\sigma', F') \Rightarrow \exists E'.((\sigma, E) \xrightarrow{a} (\sigma', E') \wedge E' \sqsubseteq_{svrs} F')))$

Intuitively, a refinement under SVRS must ensure that for any variable state the ready sets must be the same. For any transition from any variable state in the original model there must be a consistent transition from the same variable state in the new model leading to behavioural states that are also ordered by SVRS. The transition must be consistent in that its variable update must be present in the original model (from the data state in question). State is included up front in the language and no refinement of the state will be conducted.

The CoSta contract language has a dual interpretation. Contracts are also given a direct characterisation as a single STGA (LTS) allowing mixed descriptive styles within the same heterogeneous framework. A *maximal agent* for each contract is the nondeterministic choice over all the processes that satisfy it. Contracts can be expressed as maximal agents

which can then be parallel composed with other agent expressions.
For example, the meaning of the contract $\langle a \rangle$ (a is enabled) in terms of an LTS is the nondeterministic choice between all ready sets that include a, e.g.
$(a.True \Box b.True \Box c.True) \sqcap (a.True \Box b.True) \sqcap (a.True \Box c.True) \sqcap (b.True \Box c.True) \sqcap ...$
The above description uses CSP style notation rather than the more verbose CCS style language the original work was set in. $a.A \Box b.B$ and $a.A \sqcap b.B$ in CSP correspond to $a.A + b.B$ and $t.a.A + t.b.B$ in CCS respectively. The characterisations of contracts in CoSta are tau-pure, meaning choices between observable and unobservable actions never arise. Like Logic LTS, Costa LTSs are extended with distinct *False* transition systems representing the empty choice of behaviours.

This section has given a short intuitive description of the refinement theory for Contractual State Machines. Although the contract language and refinement orderings were developed by other members of the group, the diagram syntax was developed jointly. The syntax was adapted (e.g. introduction of *AND* states, outer contracts) as part of the work on patterns, which forms the body of the thesis.

2.13.6 Design strategy

This section describes an exemplar design strategy for CoSta. The usual approach for using μ-calculus and a process algebra is to create a design in CCS/CSP then define some safety properties using μ-calculus and use a tool such as the Concurrency Workbench to verify that the design satisfies the safety properties. The approach here is a top-down one to firstly express the safety properties and refine this into a design.

This approach where the design is gradually elaborated in a top-down, step-wise fashion as the result of incremental manipulation of the specification/design arguably reflects the overall approach that systems engineers typically follow to design systems, by starting with requirements and progressively adding further detail as design decisions are made [73, 98, 113, 150, 171].

The design process has two stages, an initial stage and a main stage. The initial stage of the process is to devise the top-level contract which is based on closed reasoning and expresses model-wide properties. The closed contract language has not yet been specified, this will require further work. It is intended that the closed contract language will express high-level properties in a temporal logic based on μ-calculus extended with data parameters for describing events and state. The variables of the model are impervious

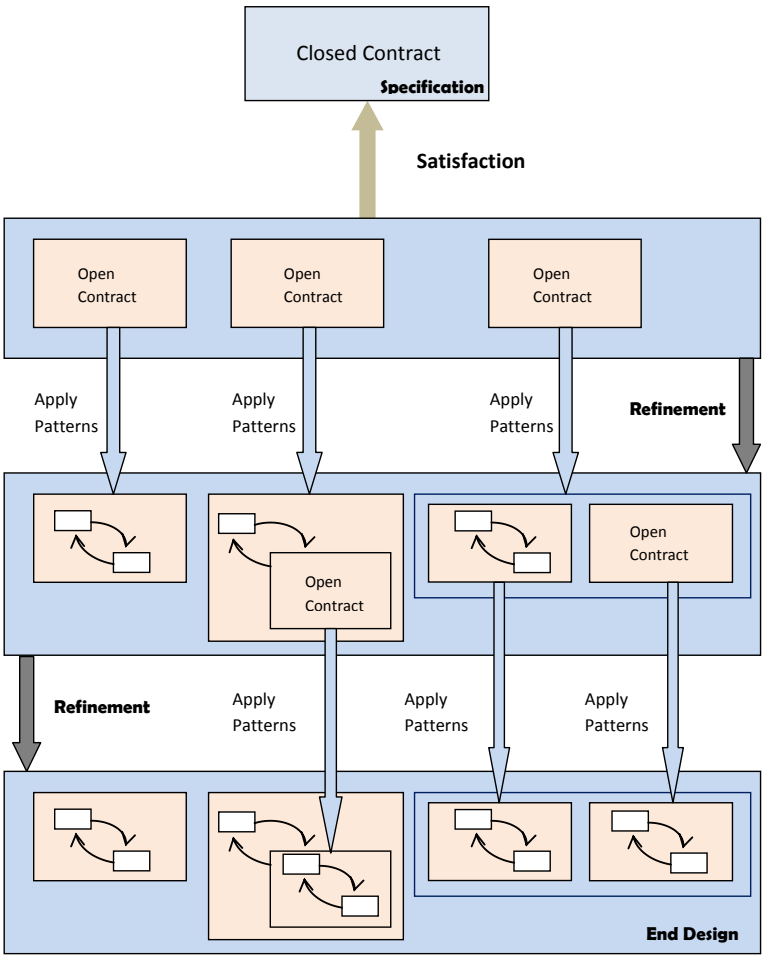

Figure 2.5: The design strategy

to outside interference. A closed contract is implemented by a set of open contracts if the parallel composition of their interpretation as processes satisfies the closed contract.

The main stage of the design is based on open reasoning, which leads to expression of properties of the subcomponents of the model, where variables are subject to interference from the other subcomponents. Once the open contracts have been devised and shown to imply the property of interest (top-level closed contract) then refinement can begin on the open contracts through the application of the refinement and refactoring patterns presented in Chapters 4 and 5. Refinement as opposed to satisfaction now proceeds based on the required preorder (shared-variable ready simulation).

A contract is typically nondeterministic and can be implemented by different deterministic state machines or refined to a less nondeterministic but behaviourally equivalent contract. The finished design is a state machine without contracts. Outer contracts can be removed at any time without affecting the behaviour. The same is not true of inner contracts, the absence of an inner contract denotes no further transitions i.e. deadlock. It is not permitted to remove an inner contract only refine it out (e.g. to deadlock or a looping transition) using the patterns presented in Chapters 4 and 5.

2.13.7 Examples

2.13.7.1 Example 1 : Simple synchronising concurrent processes

A simple example is given next to demonstrate input and output separation of the synchronisation channel. The example shows two concurrent agents *C1* and *C2* communicating via actions *'Pass* and *Pass*. Processes communicate if they both have complimentary actions available. A pair of actions are complimentary if both have the same identifier but one is primed and the other is not.

In the example system when *C1* has communicated via *In* it will tell the other process *C2* that it has received a communication via its synchronising action *'Pass* and then wait for the next communication. Process *C2* is then able to communicate via its synchronising action *Pass* and will report the fact that process *C1* received a communication to the environment via *'Out* and then return to waiting for *C1* to inform it of the next communication.

The two processes *C1* and *C2* may operate independently or if their actions permit may operate concurrently and synchronise with one another. The *Pass* action is similar to a *restricted* action in CCS. It is assumed that *C2* only communicates via *Pass* with *C1*

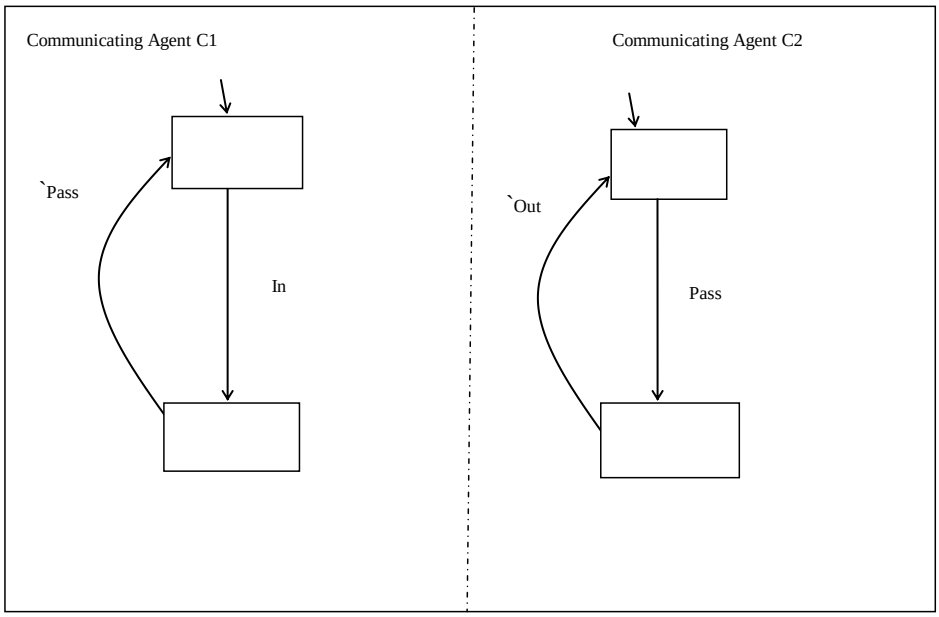

Figure 2.6: Synchronising concurrent processes

and with no other process in the environment. Thus *C2* can perform no action at all until it synchronises with *C1* via the action *Pass*.

The top-level, closed contract is $\Box\langle\_\rangle$, it says that an observable action must be enabled so it is not possible for the system to deadlock (as only internal actions can lead to deadlock). The top-level contract expresses high-level properties such as deadlock freedom and invariants. The open contracts express high-level design decisions, for example relating to how actions synchronise or how to sequence the synchronising actions and permitted updates to shared variables. The system is defined as the parallel composition of the two agents *C1* and *C2*. Different open contracts could be specified being more or less specific about design details.

Version 1 (more constraining than Version 2 that follows)

Open contract for communicating agent *C1*.

$C1 \stackrel{def}{=} \langle[In]\rangle \wedge \overline{\langle -In\rangle} \wedge \ \Box[In](\langle[']Pass]\rangle \wedge \overline{\langle -'Pass\rangle}) \wedge$

$\Box[']Pass](\langle[In]\rangle \wedge \overline{\langle -In\rangle})$

Open contract for communicating agent *C2*.

$C2 \stackrel{def}{=} \langle[Pass]\rangle \wedge \overline{\langle -Pass\rangle} \wedge \ \Box[Pass](\langle[']Out]\rangle \wedge \overline{\langle -'Out\rangle}) \wedge$

$\Box[']Out](\langle[Pass]\rangle \wedge \overline{\langle -Pass\rangle})$

Version 2 (less constraining than Version 1 above)

Open contract for communicating agent *C1*.

$$C1 \stackrel{def}{=} (\langle[-'Pass]\rangle \wedge \overline{\langle'Pass\rangle}) \curvearrowright (\overline{\langle-'Pass\rangle} \wedge \langle['Pass]\rangle) \wedge$$
$$\Box['Pass]((\langle[-'Pass]\rangle \wedge \overline{\langle'Pass\rangle}) \curvearrowright (\overline{\langle-'Pass\rangle} \wedge \langle['Pass]\rangle))$$

Open contract for communicating agent *C2*.

$$C2 \stackrel{def}{=} \langle[Pass]\rangle \wedge \overline{\langle-Pass\rangle} \wedge \Box[Pass]((\langle[-Pass]\rangle \wedge \overline{\langle Pass\rangle}) \curvearrowright (\overline{\langle-Pass\rangle} \wedge \langle[Pass]\rangle))$$

2.13.7.2 Example 2 : Mutual exclusion

This example describes a system which only allows one of two processes to perform certain critical actions at a time. The two processes are unable to perform their critical actions in parallel. One process *Sem* acts as a semaphore to control access to critical regions of the processes' behaviour. Processes *P1* and *P2* have critical actions *c1* and *c2* respectively. *P1* and *P2* will only be allowed to perform their critical actions if they can gain control of the semaphore. The semaphore is free if the next action *Sem* can perform is *get* and similarly is occupied if the next action it can perform is *put*. Control of the semaphore can be achieved by *P1* or *P2* successfully synchronising with *Sem* on the *get* action. After gaining control of the semaphore a process can perform its critical action and on completion it must relinquish control for future use. CoSta does not allow multiple synchronisations on the same action. The parallel operator is a shared-variable version of CCS's two-way synchronising operator.

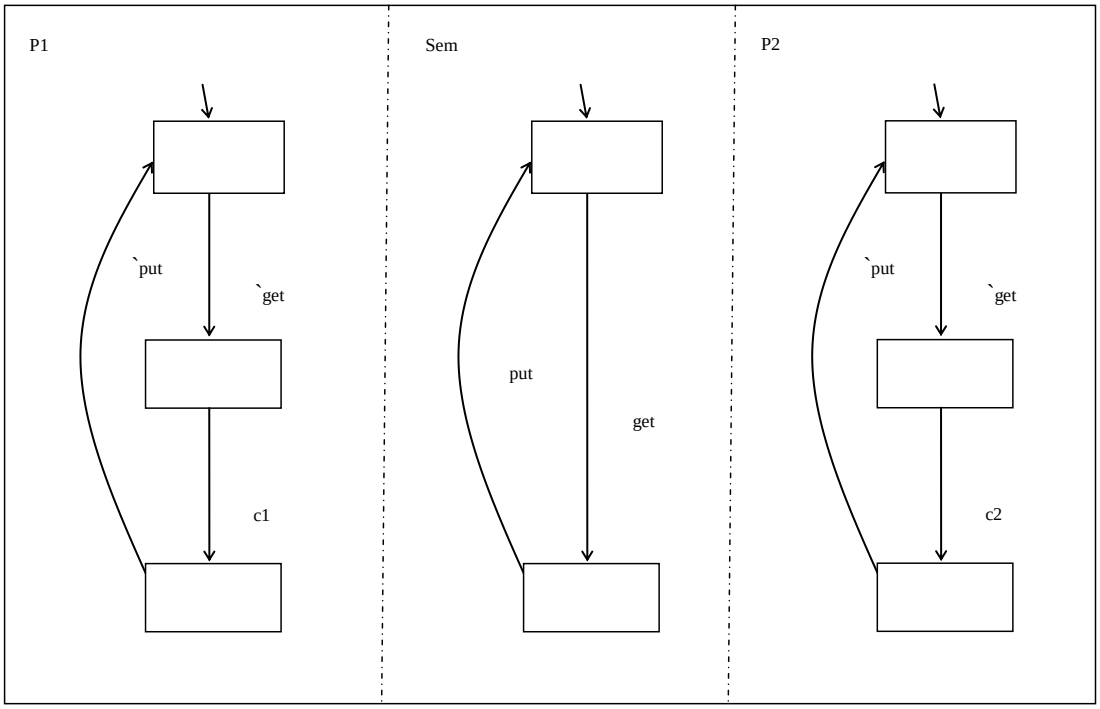

Figure 2.7: Mutual exclusion

The top-level, closed contract says that it must always be the case that an action is enabled and the actions *c1* and *c2* cannot both be enabled at the same time.

$\Box((\langle -c1, c2\rangle \wedge \overline{\langle c1, c2\rangle}) \vee (\langle c1\rangle \wedge \overline{\langle c2\rangle}) \vee (\langle c2\rangle \wedge \overline{\langle c1\rangle}))$

The system is defined as the parallel composition of the three agents *P1*, *P2* and *Sem*.

Version 1 (more constraining than Version 2 that follows)

Open contract for *P1*.

$P1 \stackrel{def}{=} \langle [' get]\rangle \wedge \overline{\langle -' get\rangle} \wedge \ \Box[' get](\langle [c1]\rangle \wedge \overline{\langle -c1\rangle}) \wedge$

$\Box[c1](\langle [' put]\rangle \wedge \overline{\langle -' put\rangle}) \wedge$

$\Box[' put](\langle [' get]\rangle \wedge \overline{\langle -' get\rangle})$

Open contract for *P2*.

$P2 \stackrel{def}{=} \langle [' get]\rangle \wedge \overline{\langle -' get\rangle} \wedge \ \Box[' get](\langle [c2]\rangle \wedge \overline{\langle -c2\rangle}) \wedge$

$\Box[c2](\langle [' put]\rangle \wedge \overline{\langle -' put\rangle}) \wedge$

$\Box[' put](\langle [' get]\rangle \wedge \overline{\langle -' get\rangle})$

Open contract for *Sem*.

$Sem \stackrel{def}{=} \langle[get]\rangle \wedge \overline{\langle -get\rangle} \wedge \ \Box[get](\langle[put]\rangle \wedge (\overline{\langle -put\rangle}) \wedge$

$\Box[put](\langle[get]\rangle \wedge \overline{\langle -get\rangle})$

Version 2 (less constraining than Version 1 above)

The open contract for *Sem* remains the same as above. The new open contracts for *P1* and *P2* are:

$$
\begin{aligned}
P1, P2 \stackrel{def}{=} & ((\langle[-'get,' put, c1, c2]\rangle \wedge \overline{\langle 'get,' put, c1, c2\rangle}) \\
& \qquad \curvearrowright (\overline{\langle -'get\rangle} \wedge \langle['get]\rangle)) \wedge \\
\Box['get](& ((\langle[-'get,' put]\rangle \wedge \overline{\langle 'get,' put\rangle}) \\
& \qquad \curvearrowright (\overline{\langle -'put\rangle} \wedge \langle['put]\rangle)) \wedge \\
\Box['put](& ((\langle[-'get,' put, c1, c2]\rangle \wedge \overline{\langle 'get,' put, c1, c2\rangle}) \\
& \qquad \curvearrowright (\overline{\langle -'get\rangle} \wedge \langle['get]\rangle))
\end{aligned}
$$

2.14 Related work

This section will provide a thorough review and critical analysis of work already done in the research area, covering the main contributions (strengths and weaknesses) and open issues. The work that is most closely related to the research for this thesis is that of stepwise refinement of concurrent systems [51]. Research by [87] introduces a refinement calculus for state machines. Schmidt in his PhD thesis [218] defined a variant of state diagrams with a precise operational semantics and provided refinement patterns that allow the reduction of nondeterminism for the verifiable top-down development of state diagrams. The approach did not consider contracts or data, the patterns were not phrased explicitly as model transformations and tool support was not considered.

Paige *et al* [200] have considered refinement in the context of model-driven engineering (MDE). Refinement is a key practice and part of the OMG model-driven architecture (MDA) initiative but it is loosely defined and overloaded. MDA's languages are UML-based and generally do not provide the level of formality required for refinement-based design. Also the languages are multi-view, thus causing difficulty in maintaining consistency during refinement between different model views. To improve this situation the research concentrates on formulating a precise and lightweight definition of refinement in terms of model consistency. Model consistency checks are categorised as structural, behavioural, cross-model (PIM-PSM), and domain specific (PSM-domain). Model refinement

is achieved through verifying model consistency defined by consistency rules expressed in OCL. A disadvantage with this approach is that soundness and completeness are an issue due to there being no underlying formal theory.

Maraninchi has incorporated logical-time contracts with reactive embedded components [162]. The contract language, however, is not a first-class citizen within the mixed design notation. There are several strands of research concerned with extending OCL with temporal logic. Ziemann *et al* [267] have extended OCL with temporal operators to formulate temporal constraints. A set of refactorings for UML models using OCL pre and postconditions was presented by Sunye *et al* [242]. There is no underlying formal theory and no mechanism was described for automating the refactorings.

Research based on transforming semi-formal models such as UML statecharts into formal notations such as Z and B is plentiful, for example [201, 229]. The motivation for this approach is the ability to validate the formal specifications using stable and proven tools. For example UML is combined with Object-Z in Fusion [40]. In very few cases is a refinement calculus constructed to formalise the design process.

Lano *et al* [148] use the Z-based semantic model of Syntropy as the basis for a semantic framework for UML and refinement transformations for state machines. Contracts and patterns are not considered for this research.

Kempen *et al* [253] focus on refactorings for UML models. The method they use to prove that a refactoring preserves the behaviour of the system is to map a statechart to CSP and show that the behaviour of the processes before and after the refactoring are identical. They define primitive pre-proven statechart refactoring operations, (e.g. for combining and splitting states and transitions) as the basis for more complex refactorings. This research concentrates on refactoring operations for state machines but refinements, contracts and patterns are not considered.

Sun Meng *et al* [166] give a coalgebraic semantics to UML state machines and define a notion of behavioural equivalence and refinement. Refinement laws are specified that provide a syntax-based approach for constructing correct designs from intermediate state machine designs. A set of elementary refinement steps were described, (e.g. to introduce states and transitions) that can be combined to achieve more powerful refinements. This research did not consider refinement for a heterogeneous language combining contracts with statecharts, patterns or tool support.

Khriss *et al* [133] present a pattern-based approach to the stepwise refinement of UML

models. They propose the use of refinement schemas that implement tradition design patterns for designing UML models. The schemas are supported by tools for their application. They also describe a number of smaller transformations called micro-refinements that can be composed to produce new refinement schemas for other design patterns. A refinement schema is parameterised by model elements instantiated by the designer. It takes an abstract model and produces the corresponding detailed model. Patterns can be classified as either structural or behavioural. Structural or equivalence patterns preserve meaning and may thus be applied in both directions. They are refactoring patterns and may be used to simplify structure for example. Behavioural patterns on the other hand refine behaviour. Micro-refinements do not have a formal underpinning. This research focuses on the steps to achieve the target design for a design pattern in UML, as oppose to patterns for this thesis that refine a high-level, declarative contract to a concrete, operational state machine design.

Frank and Eder [94] in their work on equivalence transformations for statecharts firstly formalize the semantics of statecharts and give a definition of equivalence. Statecharts are described using a logic language with pre/postconditions called TQL++. They describe twenty-three equivalence preserving transformations for statecharts, complex transformations can be established from this basic set of transformations. This research did not consider refinement from contracts to statecharts, patterns or tool support.

Scholz [221] specifies a formal semantics and refinement calculus for a dialect of statecharts, (μ-Charts). This research is for a very restricted statechart language with only three principal syntactical concepts, sequential automata, hiding and the composition operator, to simplify the semantics. The calculus permits refinements to μ-Charts (e.g. add, delete, modify transitions), providing they do not introduce additional nondeterminism. This research does not consider contracts, patterns or tool support.

Research by Rumpe *et al* [216] provide a denotational and operational semantics and a refinement calculus for total automata. Refinement corresponds to the reduction of nondeterminism. Contracts, patterns or tool support are not considered.

Park *et al* [201] give a process algebraic semantics to IBM Rational's Statemate statecharts based on the Algebra of Communicating Shared Resources (ACSR). Statemate statecharts are described in ACSR, a real-time process algebra with a well defined semantics and a notion of equivalence based on bisimulation. VERSA is an interactive tool that can be used to verify properties of ACSR specifications and determine behavioural

equivalence. Contracts and patterns were not considered for this research.

Schonborn *et al* consider refinement patterns for hierarchical UML state machines [222]. This work gives a formal semantics to UML state machines and describes refinement steps in terms of state machine to state machine syntactic transformations. Their aim is for a tool-supported refinement theory. The refinement patterns are for state machines only and the research did not consider contracts. This work makes explicit the notion of a refinement pattern as a behaviour-preserving transformation and they do not consider patterns that reduce nondeterminism.

Porres demonstrates an approach to expressing refactorings as a collection of transformation rules or guarded actions [206]. Each rule accepts one or more model elements as parameters and performs a basic transformation action based on these parameters. The transformations are not interactive or user-guided. The guarded action language may not be expressive enough to capture complex applicability criteria based on several model elements.

In [36–39] an EMF model transformation framework is presented. It supports EMF model refactoring based on update transformations implemented using graph transformation techniques. Transformations are visually defined by rules on object patterns on EMF models. The transformations are interpreted by the AGG graph transformation environment [245]. The transformation language lacks a full model query language and therefore may prove to be inadequate for complex transformation scenarios. Also this framework assumes that models must at all times comply with their metamodel during a refactoring process. Justification for why this is necessary is missing. Arguably this is too inflexible and makes it more difficult to support complicated (compound) refactorings.

Previous work by Lüttgen and Cleaveland [64] developed a logical process calculus that supports heterogeneous system specifications, permitting both operational and declarative styles and a behavioural preorder that allows formal, component-based refinement. The work by Lüttgen and Cleaveland studied a combination of the process algebra CCS and the Linear Temporal Logic (LTL) but based the refinement preorder on the theory of testing rather than bisimulation. The mixed language CCS+LTL is targeted at concurrency theoreticians as oppose to engineers. The semantics of the language is complex, and not very engineer friendly, leading to difficulty in implementing tool support. No attempt at axiomatising the refinement preorder or providing refinement patterns was made.

Previous work by Galloway and Toyn [123, 249] have extended the MathWorks' State-

flow language to include an annotation language on states to define assumptions (pre-conditions), which can capture information about differentials. Healthiness conditions are formulated to check that the model conforms to the assumptions, and that the assumptions are mutually consistent. Aspects of the statechart model and assumptions are translated to Z which is used for formal validation. Failure to prove healthiness conditions reveals mistakes such as inappropriate trigger conditions, missing transitions, and contradictory and missing assumptions.

A drawback of the Practical Formal Specification language and the Simulink/Stateflow Analyser (PFS/SSA) is that it is domain specific and not a general approach. It targets specific kinds of control system and only covers a subset of Stateflow behaviour. For example there is no support for events (only data conditions), no transition actions (only a restricted form of actions to set outputs) and no *And* states. Moreover there is no support for structural refinement at the chart level. The language can express only simple, propositional assumptions and it is not equipped with a theory and tool support for refinement checking.

In another line of research, Sowmya and Ramesh have extended statecharts with Lamport's Temporal Logic of Actions TLA [146, 233]. They present an approach where two independent specifications of a real-time system are created. Behaviour is described using statecharts. A specification of system properties is given using FNLOG which is highly abstract and verifiable through logical rules of deduction. Their approach to integrating the two specification methods consists of specifying a semantic equivalence between statecharts and FNLOG specifications and generating an "equivalent" FNLOG specification from the statechart model (The trace semantics of statecharts is matched to the linear temporal logic of FNLOG) and then verifying that the system properties hold. This research does not focus on component-based refinement, nor do they propose refinement patterns.

Stepney *et al* [235–237, 251], have conducted research into refactoring to design patterns at the specification level. Here patterns provide steps for refactoring an unstructured Z specification to the target Promotion structure. The set of patterns that collectively represent the well-known Z structure, Promotion, is presented, then the process of using refactoring to take an unstructured Z specification in to a Promotion structure is demonstrated. This research shows that refactoring can also be fruitfully applied at the specification level and it is not just a programming technique. This research focuses on

refactoring patterns to achieve a particular design pattern for a formal specification language (Z), whereas the patterns for our research are for a mixed heterogeneous language and are more general (not focused on achieving a particular design pattern).

The idea of design patterns for formal languages has also been considered by Abrial *et al* [5, 121] for the Event-B language. Tool support is provided by the Rodin platform which is an Eclipse-based development environment with support for refinement and mathematical proof [6]. Rodin has a library of patterns that transform an abstract Event-B specification to a concrete design for common design patterns for safety-critical systems. They include the Triple Modular Redundancy Pattern, the Recovery Block Pattern and the N-version Programming Pattern. Event-B supports formal refinement patterns but not in terms of state machines.

More distantly related research is on refinement calculi and tools, e.g. [54], which attempted to provide support and a collection of strategies for refining specifications into programs.

Woodcock *et al* [55, 56, 186–188, 217, 257] have developed Circus, a combined formal language that focuses on both data and behavioural aspects of a system. Circus combines Z and CSP with a refinement calculus to support development of reactive systems. The language does not have a graphical concrete syntax and is thus not as intuitive as a statechart language.

A key problem with expressing refinement patterns is the form used to describe them. Clark *et al.* [61] proposed a component-based form of pattern description with parameters, wherein missing units of functionality such as classes, objects or constraints are provided as parameters. They are equivalent to preconditions on when the pattern can be applied and are encapsulated within one UML-like component. A similar approach to this is used for expressing patterns for CoSta.

Statecharts have been extended with temporal logic to express general liveness and fairness properties [88]. Fecher *et al* have proposed top-down refinements of statecharts based on under and over-approximations of execution traces. This research is less general as it considers only contracts expressing safety and bounded liveness properties but additionally focuses on refinement patterns and emphasises tool support.

Darimont and van Lamsweerde [70], conducted research into formal refinement patterns for goal-driven requirements elaboration expressed in a real-time temporal logic (KAOS). The aim is to provide constructive formal support for the refinement process whilst hiding

the underlying mathematics.

The principle is to reuse generic refinement patterns from a library for guiding the refinement process. The patterns provide formal support for building goal refinement graphs and are proved correct and complete. This research offers tactics to the requirements engineer for grounding pattern selection on semantic criteria. The patterns are for a temporal logic language but the research does not consider state machines.

2.15 Summary

This chapter has presented the research area which includes reactive and concurrent systems, statecharts, contracts, patterns, refinement, refactoring and model-driven engineering. For each, the state of the art tools and languages have been evaluated and the significance of their advantages and shortcomings assessed. Contractual State Machines were introduced along with an exemplar design strategy. The literature that was reviewed identified approaches already taken in the field, and their strengths and weaknesses have been analysed. This has enabled the identification of remaining open problems.

In conclusion, model-based formal languages (e.g Z and B) are not graphical and do not have explicit support for concurrency. Process algebras are not rich enough for the description of operations on data, they are not graphical but they do explicitly support concurrency. Temporal logics have the advantage that they can be used in conjunction with other approaches e.g process algebras. They are not graphical and not well suited to express operations on data. Graphical approaches have several advantages one of which is that their formal underpinnings can be based on process algebras, model-based formalisms or temporal logic. They can potentially encompass the benefits of each approach and most importantly they are graphical and intuitive.

In summary, for this previous related research, the languages of use are typically not heterogeneous (integrating contracts with state machines) and did not consider data or shared variables. Specifically they do not integrate contracts with state machines or the contract language is not a first-class object in the mixed design language. Other drawbacks included a lack of an underlying formal theory and refinement calculus. Refinement patterns in terms of model transformations have not been considered. Additionally there is an absence of tool support for pattern application and verification of refinement steps.

Remaining open research problems include the need for a graphical specification/design language for safety-critical, concurrent, reactive systems with an underlying formal theory

that supports stepwise refinement with techniques for abstraction and the ability to express both functional behaviour and high-level properties. In addition, supported by a set of refinement and refactoring patterns to assist a stepwise, calculational style of refinement and software tools to automate the design process.

Hence this research focuses on providing a graphical concrete syntax for Contractual State Machines and refinement and refactoring patterns for the heterogeneous language, (a state machine dialect with contracts) as automated update-in-place model transformations where patterns resolve design choices or reduce nondeterminism and can take an abstract contract to a concrete state machine design. There has been limited related work encompassing all of these facets.

Chapter 3

Analysis and hypothesis

3.1 Introduction

This chapter presents the research hypothesis, objectives and contributions for the thesis. During the field review, in the previous chapter, a number of open issues for research have been identified. These provide justification, context and motivation for the research hypothesis and objectives presented in this chapter. Finally, this chapter discusses the thesis contributions that will follow in the subsequent chapters.

Recall from Chapters 1 and 2 that although formal methods offer the rigorous verification techniques demanded by safety-critical systems, on the whole a wider acceptance of formal methods within industry has been hindered for many reasons, including usability and scalability issues [3, 4, 35, 45, 46, 63, 95, 96, 108–111, 152, 154, 163, 241, 261]. Semi-formal languages are intuitive but generally lack support for a systematic refinement process. Statecharts, for example, have no rigorous process for constructing models which support abstraction and the systematic and stepwise introduction of detail.

Related research so far has considered the stepwise refinement of concurrent systems [51–53, 228] and refinement calculi for formal languages for concurrent systems [18–21, 55–57, 187, 217, 257, 258, 262]. Previous research has considered pattern-based approaches to refinement [121, 122, 133, 145, 222]. Research by [94, 206, 222, 242, 253] has concerned refactorings for state machines. Refinement transformations for variants of state machines have also been proposed [131,148,166,216,221]. In particular, research has been conducted into a refinement calculus for state machines [87–89] and refinement patterns that allow the reduction of nondeterminism and support verifiable top-down development of state diagrams [218]. Research into system specification and refinement in temporal logic and

hennessey-milner logic has been carried out [116, 204].
Notably, in this previous related research,

- The languages of use have typically not been heterogeneous, with techniques for abstraction as well as the ability to express functional behaviour; then also did not consider data or shared variables. Specifically they do not integrate contracts with state machines, or the contract language is not a first-class object in the mixed design language, where contracts specify high-level properties and are potentially succinct ways of describing highly nondeterministic state machines.

- Importantly, other drawbacks in all of this research included a lack of an underlying formal theory and a refinement calculus to formalise the design process. Some previous research has proposed a formal refinement calculus [221] but refinement and refactoring patterns in terms of model transformations have not been considered.

- Additionally there is an absence of tool support for automated pattern application and verification of refinement steps.

Contractual State Machines support compositional refinement with techniques for abstraction and the ability to express both functional behaviour and high-level properties. This thesis develops the methodology behind Contractual State Machines (developed by Galloway *et al*) to address the above challenges. It does this by identifying, specifying and providing tool support for a set of *refinement and refactoring patterns*. Refinement and refactoring patterns, as discussed in the sequel, provide a systematic and structured way of using Contractual State Machines for rigorous development. Consider the following scenario: an engineer takes a specification and proposes a final design, but they want to prove that the design satisfies the specification.

A model checker could potentially be used to prove this refinement, but a general refinement check can be computationally intensive and may not succeed given time and space constraints. By comparison, small, easily proved *refinement and refactoring patterns* should be easier to validate, and also provide a systematic way to achieve a design from a specification that breaks the refinement into simple steps. This enables the engineer to see that the safety properties are preserved at each step or if this is not the case, to see where the error lies in the new design as only a small change has been made. Whereas if the engineer proposed the final design immediately and conducted the refinement in a

single step, a model checker may only be able to report that the design is not a refinement and the engineer would have no idea why.

From an engineering perspective, this provides a means to manage complexity via a capability to manipulate manageable chunks of models. This thesis focuses on identifying, specifying and implementing refinement and refactoring patterns for Contractual State Machines (CoSta), along with tool support for their application. The implementations and tool support are based on automated update-in-place transformations, where the transformations apply the refinement and refactoring patterns to resolve design choices or reduce non-determinism.

As discussed in Chapter 2, there has been limited related work encompassing all of these facets. The challenge here is that the definition for refinement i.e. CoSta's refinement relation, does not lead directly to refinement and refactoring patterns. The refinement theory specifies how to relate processes in a refinement relation, but it does not necessarily help identify the required refinement steps, or express refinement and refactoring steps as model transformations needed to capture them as patterns. The research presented in this thesis addresses these challenges and the thesis hypothesis is described in more detail in the next section.

3.2 Research hypothesis

This thesis proposes a tool supported, model-based approach to a systematic and stepwise design process that supports refinement of Contractual State Machine designs from abstract specification to concrete model through the application of refinement and refactoring patterns. Such a process arguably reflects the overall approach that systems engineers typically follow to design systems, by starting with requirements and progressively adding further detail as design decisions are made [73, 98, 113, 150, 171].

However, current reactive systems engineering practice provides limited *systematic* support for a process like this. One way of adding rigour to the process of introducing design detail, and the one advocated by Model Driven Engineering (MDE), is the use of model transformations to implement patterns that ensure consistency is maintained between models with regards to, for example, intent, quality and healthiness. One benefit of being disciplined in design is that the traceability relationships from design to requirement should be easier to identify and use.

In this context and to address these needs, the hypothesis of this thesis is stated as follows:

We can identify for Contractual State Machines, a comprehensive set of refinement and refactoring patterns that ensure consistency between designs and enable the stepwise refinement of an abstract contract (that specifies high-level properties) to a fully specified concrete design (that preserves the high-level properties). Furthermore we can express each refinement or refactoring step as a pattern, in terms of model transformations. The aim is to ensure the completeness, correctness, utility and consistency of the patterns. We can automate the systematic engineering process for Contractual State Machines by implementing software with an emphasis on usability.

General aims for the pattern catalogue suggested in the hypothesis are for a compact and extensible set of rules. The patterns themselves should encourage well-formed designs (e.g. without disconnected states). The aim is for patterns that are of practical utility to the engineer. This aim can be fulfilled in many ways, for example a pattern for a frequently required refinement or refactoring step, or a pattern that is valuable in terms of proof reduction. A pattern that simplifies a design (e.g. for combining states) or simplifies the refinement process (e.g a pattern that combines refinement steps) is also of practical utility. Each pattern should be justified over and above a full refinement check.

Pattern comprehensibility is another goal: to keep patterns as simple to understand and apply as possible. Ideally the pattern catalogue will be a set of patterns that cover specific cases, which require little by way of accompanying proof, with a few general patterns in a supporting, rather than an essential role should the more specific patterns not cover what is required.

The objectives of the thesis are:

1. Identify, for Contractual State Machines a comprehensive set of core refinement and refactoring patterns for the common types of refinement or refactoring that are frequently required during the top-down, stepwise design of a system. Where each refinement or refactoring step preserves the functionality of the original specification and elaborates the design as it proceeds to permit the gradual introduction of more specific details about the behaviour of the system, by introducing design constructs,

and reducing nondeterminism.

2. Specify a catalogue of patterns for the refinement and refactoring steps. A pattern consists of a source model, a target model, and a logical description of a relationship between them.

3. Implement software to support a repository of refinement and refactoring patterns and their application (to validate side-conditions and perform model transformations) during the refinement of a design. The software will also include a graphical editor for Contractual State Machines and a catalogue of patterns. The patterns are implemented as update-in-place model transformations and integrated with model checking technology (HST) for validating side-conditions. Side-conditions ensure consistency is maintained between models and that the pattern is applicable to the design in the circumstance it is selected.

4. Further validate the hypothesis by conducting a case study that refines an abstract Contractual State Machine design (expressed as a contract) to a fully elaborated state machine model using only the refinement and refactoring patterns (via the tool support). The case study will gauge the practicality of the approach and suitability of the patterns.

The following sections of this chapter describe the approach adopted to evaluate the thesis hypothesis, the scope of the research and the overall contributions.

3.3 Approach

An exploratory approach was adopted to evaluate the validity of the hypothesis [184,230]. The exploratory approach was preceded by an initial evaluation of the contract language followed by iterations of exploration and elaboration to identify patterns. Although the contract language is suitable for expressing high-level properties of a design (e.g. safety properties), as it is based on μ-calculus, it was informative to evaluate how expressive the contract language is for characterising state machine designs or finding their strongest abstractions, to inform the research on patterns. The purpose was to find out which designs, nondeterministic or otherwise, could be completely characterised by the operators of the contract language or if this was not possible what their strongest abstractions were, or most constraining contract that captured the solution but not necessarily uniquely.

It was important to follow a systematic method to identify patterns to help achieve a complete set in terms of coverage of transformations, and ensure that the catalogue consists of a comprehensive set of patterns for the common types of refinement or refactoring that are frequently required during the stepwise design of a system.

There are three main stages to the approach. The first stage aims for completeness of the set of patterns by ensuring operator coverage and consideration of refinement rules for statecharts in the literature [94, 148, 166, 216, 221, 242, 253]. The second stage of the approach concerns the specialisation and generalisation of patterns. Specialisation leads to a simpler side-condition and generalisation to better coverage. For example a specialised pattern will only be applicable in certain specific circumstances whereas a more general pattern will be more widely applicable. The final stage of the approach was to verify the integrity and utility of the proposed set of patterns through a case study.

Software tools were implemented to create a repository of patterns and support the design process and pattern application. The patterns were used experimentally in the case study. The purpose of the case study is to give us empirical confidence that the patterns are correct. Implementing the software tools and conducting the case study enabled further exploration of the patterns and design method which led to ideas for further research.

3.4 Research scope

The scope of this research was constrained in order to impose sensible boundaries. It was limited to defining and implementing a minimal and complete set of core refinement and refactoring patterns to ensure that any state machine design that is a valid refinement of the abstract contract can be achieved through application of patterns from the catalogue.

The patterns that can be identified are constrained by the availability of realistic case studies and examples which are restricted by the lack of access to proprietary material. The patterns can be proven correct with respect to the refinement relation, but this was considered beyond the scope of the work as the formal definitions of the refinement ordering are not yet mature enough to support formal proof. Additionally the decision was made to focus on MDD tool support to help to access practicality in the case study.

3.5 Contributions

The contributions of this thesis include refinement and refactoring patterns to assist a stepwise, calculational style of refinement. The patterns enable the verifiable top-down development of a design from a nondeterministic contract to state machine implementation that is consistent with the original high-level specification. Contracts express formal properties that describe the behaviour of the system under development and refinement and refactoring patterns guarantee that the properties are satisfied as the system evolves.

Patterns reduce development effort over comparable approaches such as formal proof. The vast majority of formal development in this domain (concurrency) is achieved with model checkers, but it may be argued that mature model checkers (CWB, FDR, Spin etc.) are not good with data-rich systems.

Breaking the formal proof into application of patterns achieves two things:

1.) Reduces the overall proof/automated checking process into a sequence of steps requiring considerably less proof effort to discharge them. Refinement and refactoring patterns should be contrasted to a general refinement check which demonstrates that two models are related by a refinement relation. It checks the raw refinement ordering and requires both source and target models to be supplied. Patterns deal with specific applications of a refinement theory, and thus reduce the demonstration of correctness to showing that a specific side-condition is valid rather than performing a full refinement check. Reducing general applications of the theory to specific applications is made possible by, amongst other things, compositionality in the underlying theory of refinement.

2.) Shifts the effort from development to up-front verification of patterns. A refinement or refactoring pattern can be validated independently against CoSta's refinement relation, and this validation only needs to take place once. Patterns construct refinements and refactorings in a very similar spirit to refinement calculi (e.g. Morgan's). A refinement or refactoring pattern consists of a source model, a target model, and a logical specification of a relationship between them. The relationship, when implemented in an executable language (see Chapter 6), can be used to automatically produce the target model from the source model. Patterns take a model, run precondition checks (prioritised by easiest first) and apply model transformations, when the preconditions are satisfied.

Tools have been provided for patterns and the engineering process via use of the Epsilon toolset and model checking technology (HST). Automation reduces design and proof effort over comparable approaches and thus improves usability. A tool suite is

developed that supports verification for the refinement of abstract specifications to designs based on pattern application. Refinement and refactoring patterns are expressed as model transformations and the conditions under which patterns can be applied are defined. Tools are implemented to automate pattern application, model transformation and the validation of side-conditions.

The validity conditions for a pattern are checked to ensure the refinement or refactoring is applicable. This may require conjectures supplied as predicates on the data variables of a transformation to be proven using the model checker (HST) (e.g. to ensure that the conditions associated with two transitions are disjoint). There is nothing in principle restricting side-conditions to predicates on the data variables in the model. The work therefore includes patterns with more general side-conditions, such as those that apply refinement checks on substructures of the model.

To summarise, the contributions of the thesis are:

1. Refinement and refactoring patterns to assist a calculational style of refinement. Patterns support a top-down, stepwise design process from a contract expressing high-level properties of a design to a state machine that is guaranteed to preserve the properties. Patterns reduce development and proof effort over comparable approaches such as formal proof.

2. A tool suite is developed that supports verification for the refinement of abstract specifications to designs based on pattern application. Automation further reduces development and proof effort over comparable approaches and thus improves usability.

3. A case study is conducted to provide proof of concept.

The contributions include a rich set of patterns covering the different types of model components, i.e *contract to contract*, *contract to mixed design* and *mixed design to mixed design*. The patterns range from specific to general with side-conditions having different levels of complexity. A *specific* pattern, with respect to a more *general* pattern, is a pattern constrained for use in less situations, which usually requires less information from the user or has a simpler side-condition. Pattern constraints range from simpler syntax-based checks to more complex verification of side-conditions (e.g relating to conditions/actions on data) that require the model checker (HST) to discharge them.

The contribution of the case study is to provide proof of concept, identifying benefits of the approach and areas for improvement. The case study evaluates the patterns and demonstrates their viability.

3.6 Contrast with existing approaches

Through a comparison with related research and the existing literature this section summarises what has been achieved so far and describes how this thesis builds on the state of the art. Closely related research to-date has achieved refinement calculi for restricted forms of state machines with an underlying formal theory [166, 218, 221]. The research in this thesis is an incremental improvement on what has been achieved so far (i.e refinement steps that maintain consistency between state machine designs) as it considers refinement for a heterogeneous, data-rich (shared variable) language that combines contracts with state machines.

Our approach provides the engineer with the ability (through contracts) to express high-level properties of a design (e.g. safety properties) and abstract functional behaviour. Patterns support a top-down, stepwise design process from a contract to a detailed operational design that is guaranteed to preserve the properties of the original specification. This research provides refinement and refactoring steps from contracts to contracts and contracts to mixed designs as well as refinement and refactoring steps between state machine designs for the data-rich, shared variable language.

In addition the research in this thesis is an evolution on research to-date by expressing refinement and refactoring steps as patterns in terms of model transformations. Patterns lead to the possibility of further reducing the development and proof effort through automating the model transformations and the discharge of side-conditions and integration of the software with model checking technology.

In contrast to previous research, our approach is multi-faceted combining temporal logic contracts, state machines, and shared variables. It offers refinement and refactoring steps that maintain consistency between designs and preserve high-level properties. Refinement and refactoring patterns stepwise refine contracts to state machine designs as well as state machine designs to state machine designs, to better support top-down development from an abstract specification to a concrete design. Refinement and refactoring steps are expressed as patterns in terms of model transformations and automated. No other method has all of these features.

3.7 Summary

In this chapter we have proposed the key strands of work for this thesis, argued their novelty and discussed their relevance to the outstanding open questions in the research area. This chapter discussed the research challenges identified during the review of related work in Chapter 2, established the research hypothesis and objectives and outlined the contribution of the thesis.

In the following chapters we present the details of the contributions introduced in this chapter. In particular, Chapters 4 and 5 present the proposed catalogue of refinement and refactoring patterns for Contractual State Machines. Chapter 6 will address the implementation of the software. Chapter 7 describes the case study and Chapter 8 summarises the conclusions of the thesis. The formal foundations for the refinement process are not a contribution of the thesis as they were developed in conjunction with other members of the research group. However the formal basis for the work is summarised in Chapter 2 in Section 2.13 on Contractual State Machines, for completeness.

Chapter 4

Basic refinement and refactoring patterns

4.1 Introduction

This chapter presents the first part of the proposed catalogue of refinement and refactoring patterns for Contractual State Machines. Patterns form the basis of our disciplined engineering process which proceeds in a step-wise manner from root states with contracts to designs that implement them. The design process begins with a single state and a contract that can be satisfied by many different designs. The top-level contracts, which can express safety properties, are then refined to specify more detail about the behaviour of the system, (in terms of sequencing and data constraints) and are structured into hierarchical components.

A design is developed through successive refinement and refactoring via the application of patterns. Chapter 3 described the utility of refinement and refactoring patterns when compared to a full shared variable ready simulation check. The end design could be proposed immediately and the model checker used to prove a refinement, but an automatic proof may not be feasible [117]. A general refinement check can be computationally intensive and may not succeed given time and space constraints[1], whereas small easily proved patterns are a systematic way to achieve a rigorous and justified design.

An advantage of the underlying compositional mathematical theory of the Contractual

[1]The refinement check for Contractual State Machines is not yet implemented in the CoSta project's model checker

State Machine language is that parallel and composite states can be refined in isolation without considering the model as a whole. This permits the engineer to elaborate the design piecemeal by introducing new contracts and state machine diagrams composed together using the operators of the language. The contracts are refined into mixed designs with state machines and contracts. The design process typically continues by refining the elements of the new model that are still abstract, such as nondeterministic diagram fragments and contracts, until finally all the contracts have been refined into deterministic designs that implement them. Alternatively the aim may be to achieve an abstract solution (e.g. for reuse) where the final design is a nondeterministic model.

The reader is referred to Sections 2.13.1, 2.13.2 and 2.13.3 for a description of the key syntactic constructs for the Contractual State Machine language. In particular a definition of the modelling language in terms of its abstract syntax is given in Figure 2.3 in Section 2.13.2.

A brief intuitive explanation of the underlying refinement theory (ready simulation, shared variable ready simulation and ready sets) is given in section 2.13.5 to aid understanding of the refinement and refactoring patterns (and the refinement relation that each pattern must uphold) that follow in this and the next chapter.

As explained in section 2.13.5 the notion of correctness preservation is provided by a compositional refinement relation. The theory of refinement is based on *logic LTS* and its associated derived (tau-closed) transition system [157], which itself is derived from Ready Simulation [41]. The aim is for a pattern to ensure consistency between designs so that the refinement or refactoring step preserves meaning. Informally this has been checked for each pattern by comparing the semantic interpretation of the target model with that of the source model and ensuring that the ready sets (permissible actions) do not change for every possible data state, and new nondeterminism is not introduced. State is included up front in the language, and no refinement of the state will be conducted. The only refinement requirement is not to increase nondeterminism, this is a feature of the language. Reduction in nondeterminism is the means by which consistency is deemed correct. This can be compared to CSP where refinement is expressed in terms of *failures* rather than *ready-simulation*. A "full FDR" check corresponds to a "full ready-simulation" check. Notably the patterns presented in chapters 4 and 5 are designed to avoid the need for this. This is only necessary for the most general patterns (e.g. where the user "suggests" a refinement). There are potential reasons for checking more than just nondeterminism,

for example to prevent divergence and deadlock. Divergence is however a verification issue at the top (closed) level, since this is where parallelism and hiding is introduced. It therefore lies outside the scope of the work for this thesis. Deadlock is a kind of behaviour, if it is a possible behaviour in the abstract model, it is also a possible behaviour in the implementation. Although it is a kind of behaviour that we may like to identify and prevent, reduction in nondeterminism is still the means by which consistency is deemed correct whether deadlock is possible or not. The next section presents an example of the process of designing a CoSta state machine from an open contract to demonstrate how patterns are applied (see Section 2.13.6 for further details of the design process).

4.1.1 Example to illustrate pattern application

This section illustrates how patterns are applied to refine a CoSta contract to a state machine design. It is based on the example presented in section 2.13.7.1. The contract for the communicating agent *C2* is refined to a state machine model through pattern application.

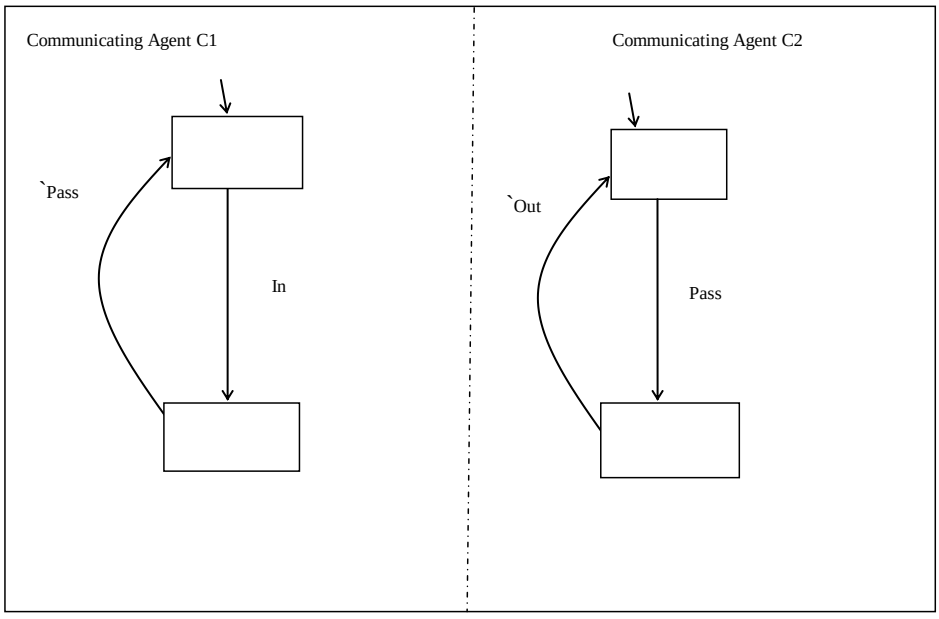

Figure 4.1: Simple synchronising concurrent processes

Open contract for communicating agent *C2*.

$$C2 \stackrel{def}{=} \langle [Pass] \rangle \wedge \overline{\langle -Pass \rangle} \wedge \Box [Pass]((\langle [-Pass] \rangle \wedge \overline{\langle Pass \rangle}) \curvearrowright (\overline{\langle -Pass \rangle} \wedge \langle [Pass] \rangle))$$

1. The pattern *Conjunction Introduction* is applied to the state with contract A. In the tool a state with the inner contract A needs to be created initially then selected and a right-click on the state displays a list of applicable patterns. The pattern *Conjunction Introduction* is selected. This pattern has one argument, a state with an inner contract that is a conjunction operator expression. The selected model element, (i.e. the state with inner contract A), is bound to this argument. When the pattern is applied it automatically performs the model transformation to introduce two new conjunction states with contracts A and B.

A. $\langle [Pass] \rangle \wedge \overline{\langle -Pass \rangle}$

B. $\Box[Pass](((\langle [-Pass] \rangle \wedge \overline{\langle Pass \rangle}) \curvearrowright (\overline{\langle -Pass \rangle} \wedge \langle [Pass] \rangle)))$

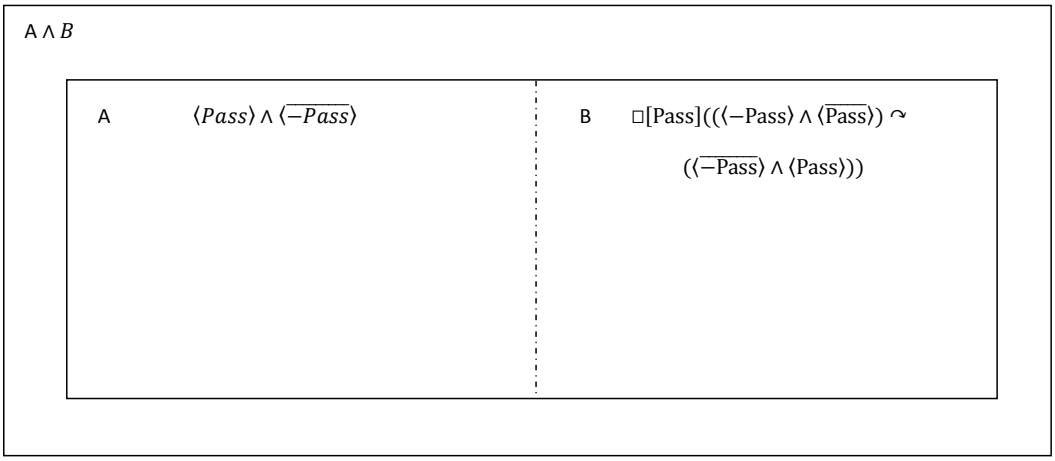

Figure 4.2: Conjunction Introduction applied to create two conjunction states

2. The pattern *Conjunction Introduction* is applied to the state with contract A to introduce two new conjunction states with contracts $A1$ and $A2$.

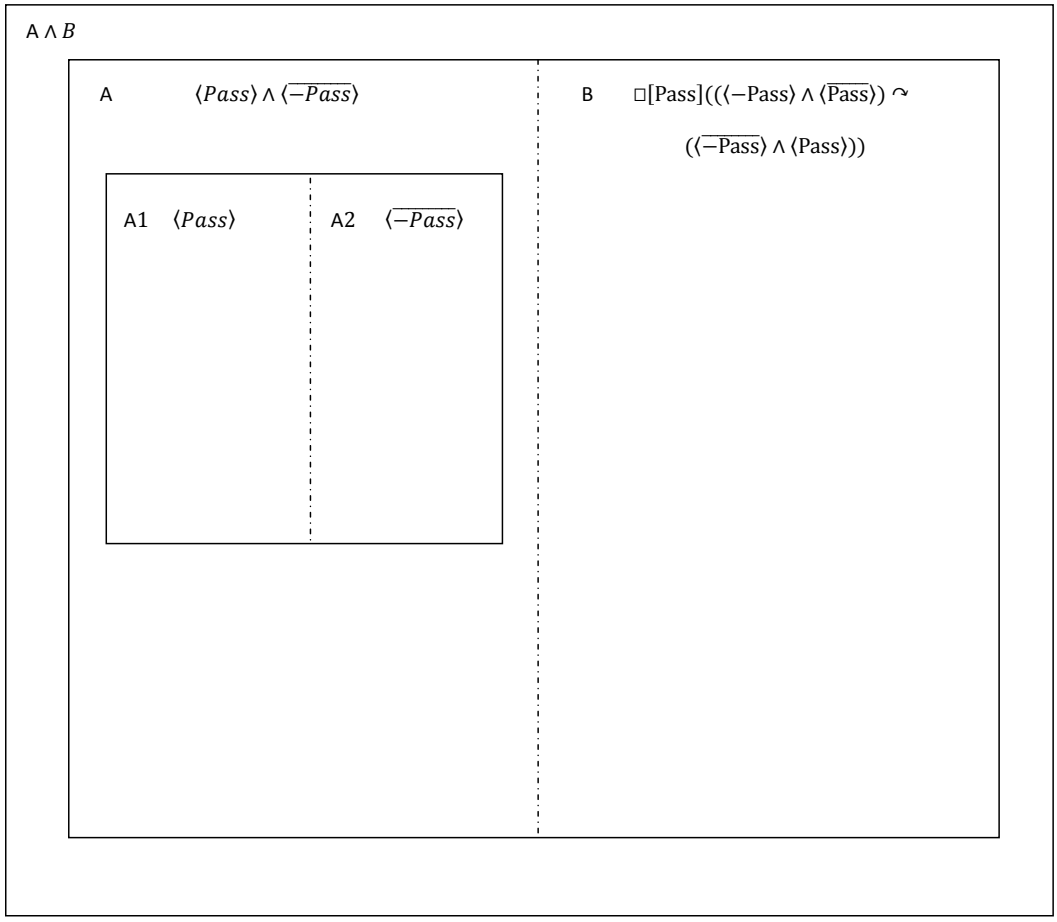

Figure 4.3: Conjunction Introduction applied to contract A

3. The *Enable* pattern is applied to the state with contract *A1* and the *Disable* pattern is applied to the state with contract *A2* to introduce the *Pass* transition. In the tool to apply the *Enable* pattern to the state with contract *A1* the state needs to be selected and a right-click displays the list of applicable patterns. The *Enable* pattern is selected. This pattern requires two arguments, a state with an inner contract that is an *Enable* operator expression and a list of new transitions to be added, specifying for each, the event and its data update expression. The selected state, (i.e. the state with inner contract *A1*), is bound to the first argument and the user is prompted to input the required new transitions. A call to the model checker may be required to determine if the transitions are permitted by the contract and if so the pattern performs the model transformation and adds the new transitions.

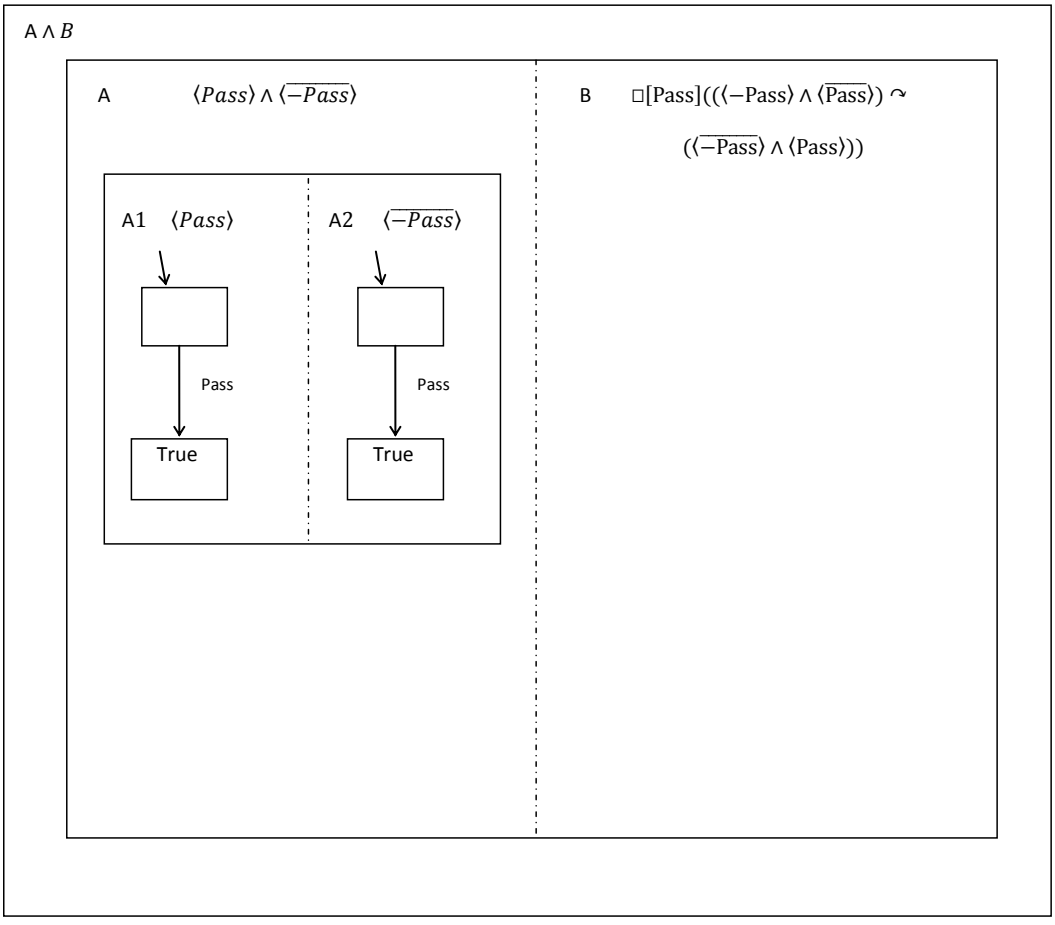

Figure 4.4: Enable is applied to contract A1, Disable is applied to contract A2

4. The *Conjunction Elimination* pattern is applied to state with contract A to eliminate all but one of its identical substates.

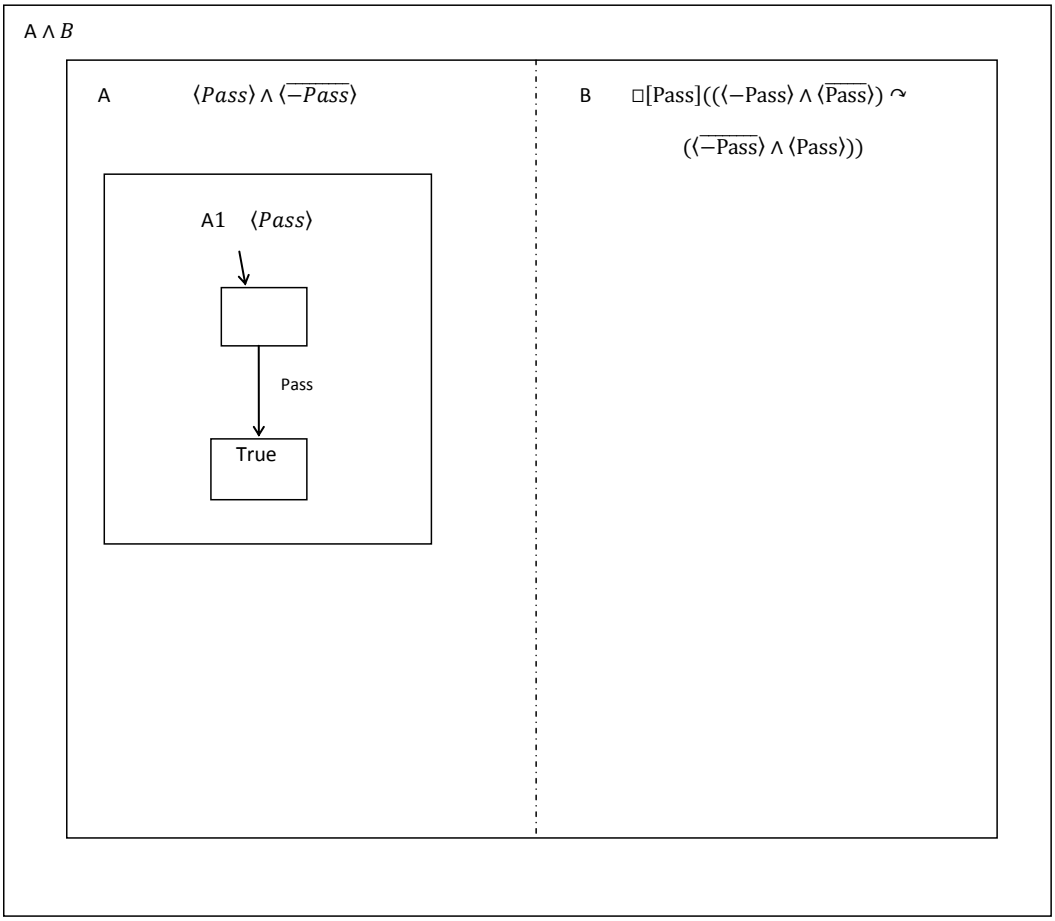

Figure 4.5: Pattern Conjunction Elimination is applied to state with contract A

5. The *Unfold Always* pattern is applied to contract B.

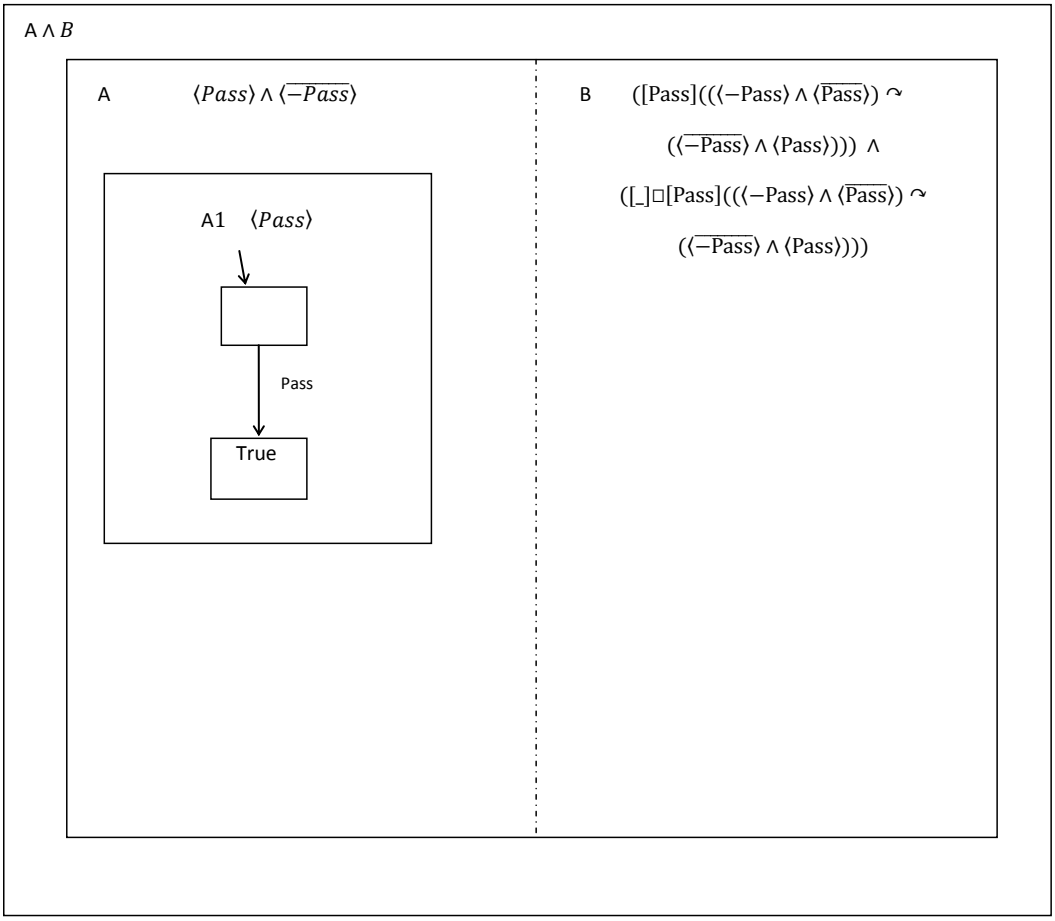

Figure 4.6: Pattern Unfold Always is applied to contract B

6. The *Conjunction Introduction* pattern is applied to contract B to introduce two new conjunction states with contracts *B1* and *B2*.

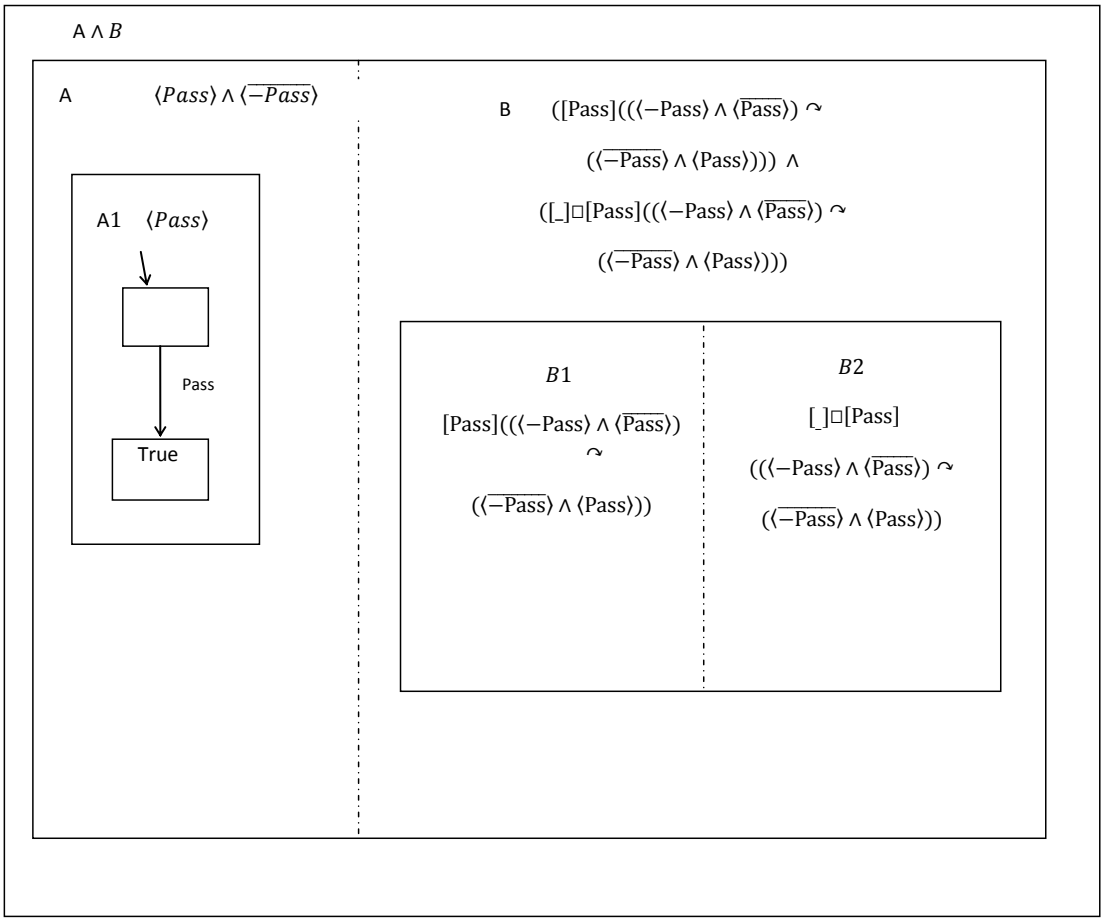

Figure 4.7: Pattern Conjunction Introduction is applied to contract B

7. The *If* pattern is applied to contracts *B1* and *B2* to introduce the new *Pass* transition.

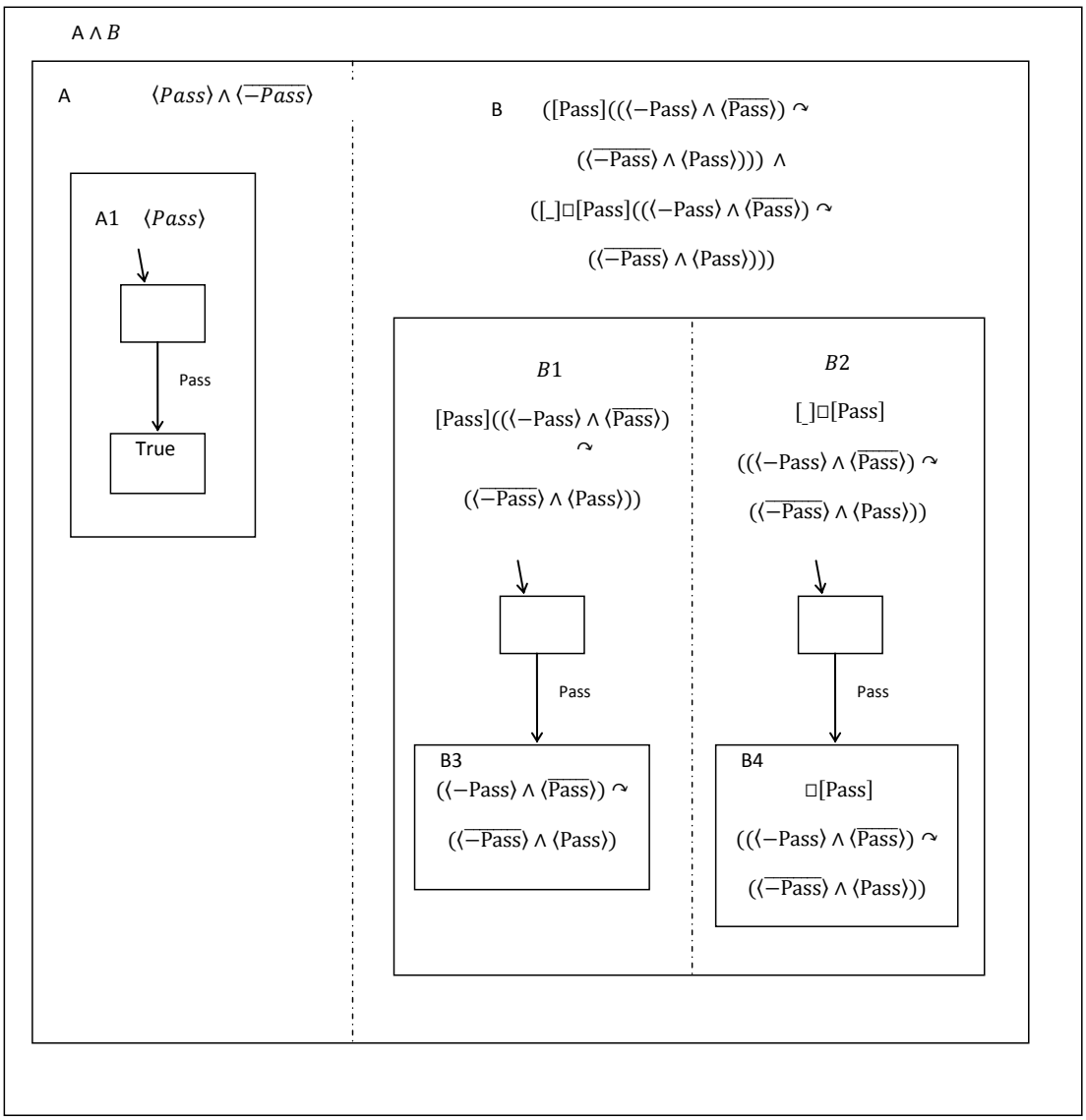

Figure 4.8: Pattern If is applied to contracts B1 and B2

8. The *Unfold Unless* pattern is applied to contract *B3*.

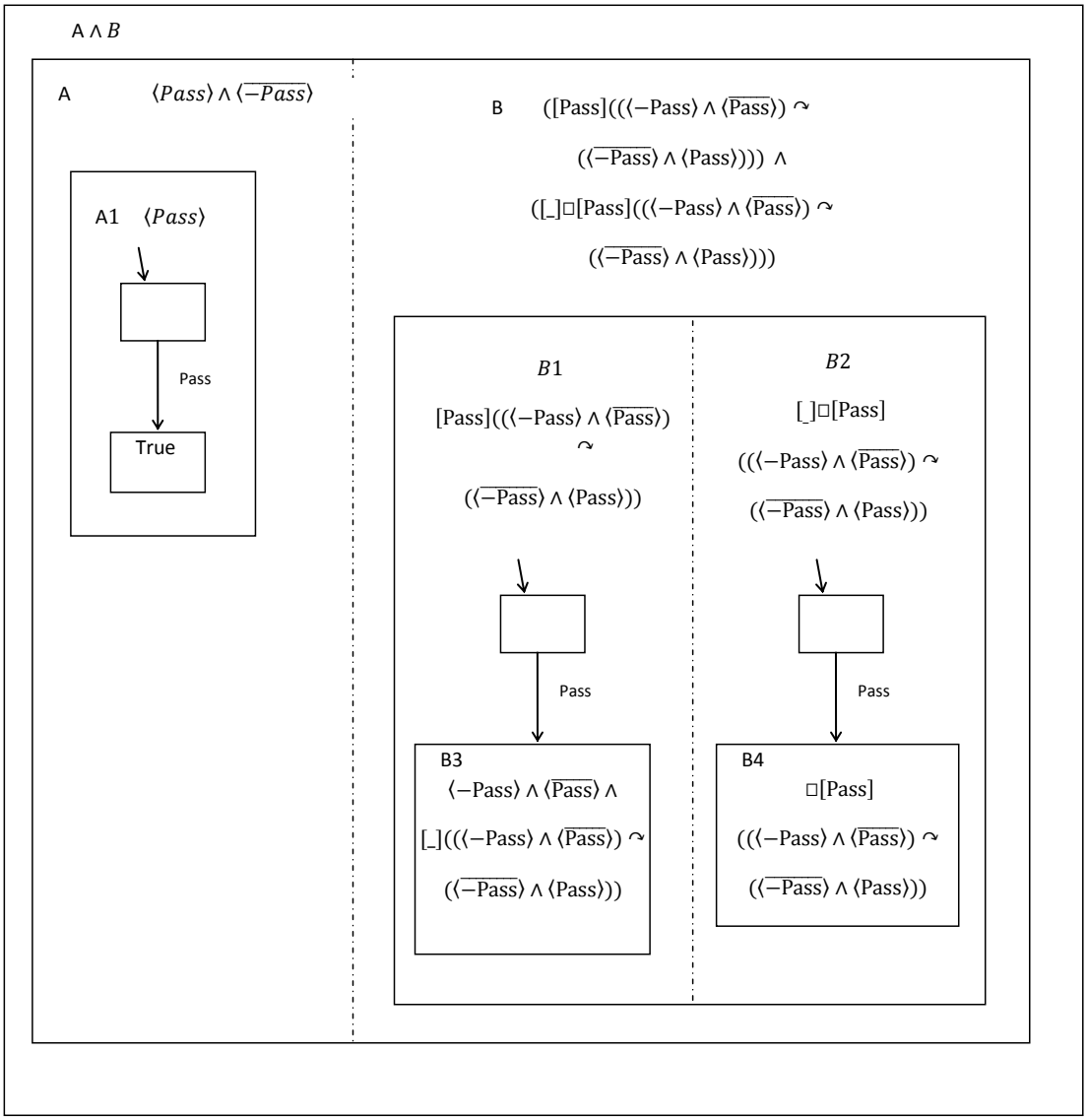

Figure 4.9: Pattern Unfold Unless is applied to contract B3

9. The *Conjunction Introduction* pattern is applied to contract *B3*.

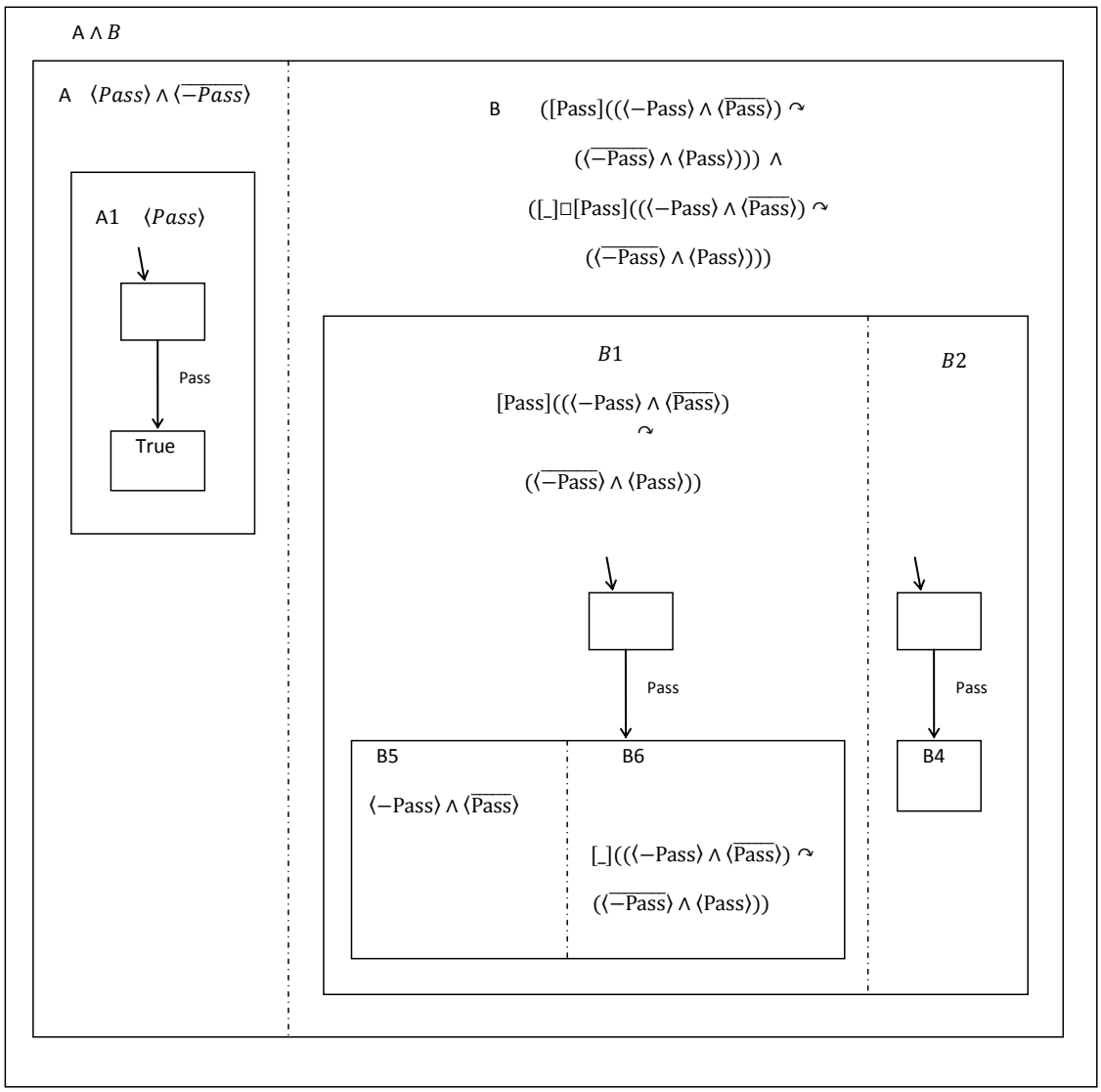

Figure 4.10: Pattern Conjunction Introduction is applied to contract B3

10. The *Conjunction Introduction* pattern is applied to contract *B5*.

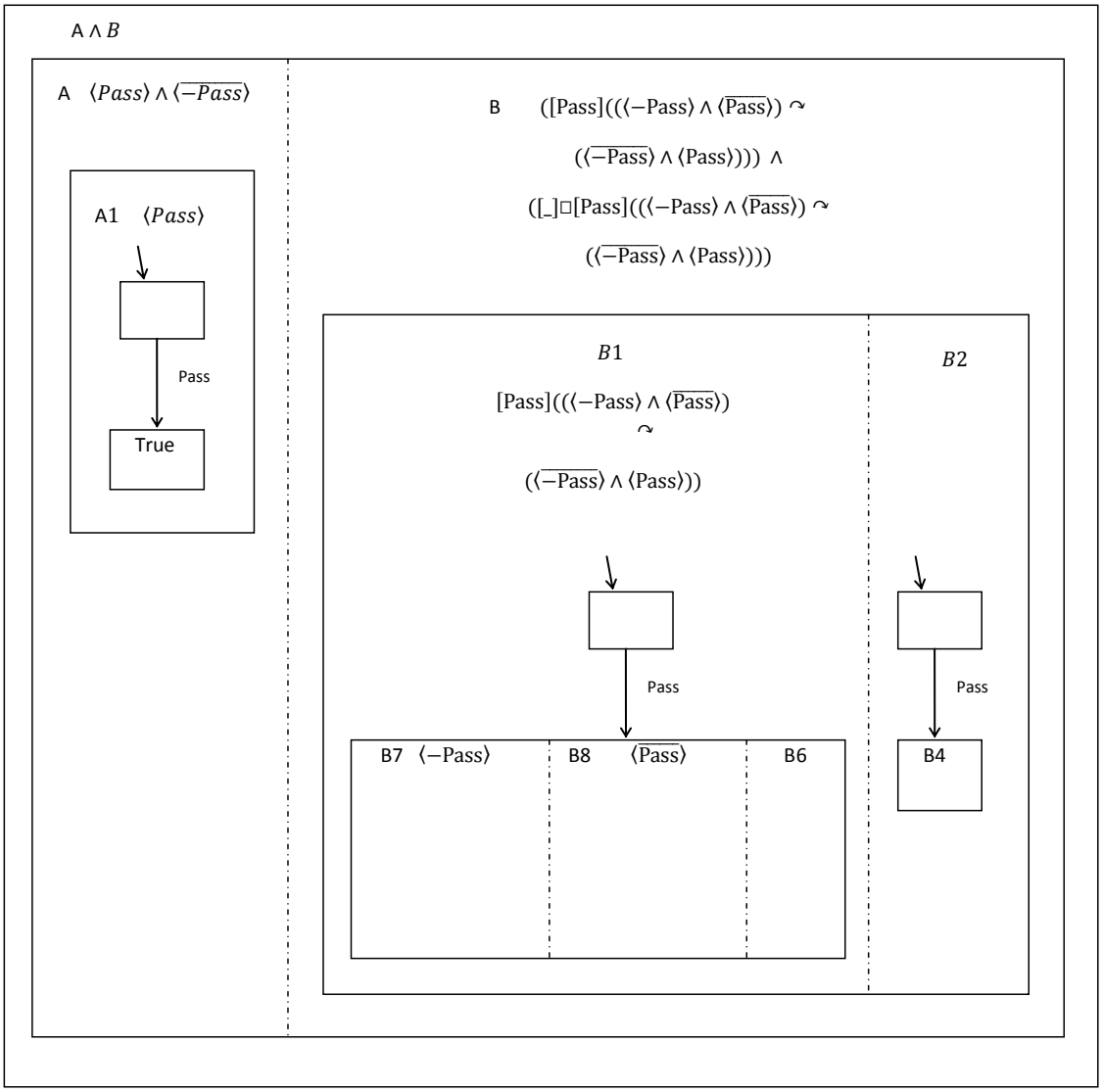

Figure 4.11: Pattern Conjunction Introduction is applied to contract B5

11. The *Enable* pattern is applied to contract *B7*, the *Disable* pattern is applied to contract *B8* and the *If* pattern is applied to contract *B6* to introduce the *'Out* transition.

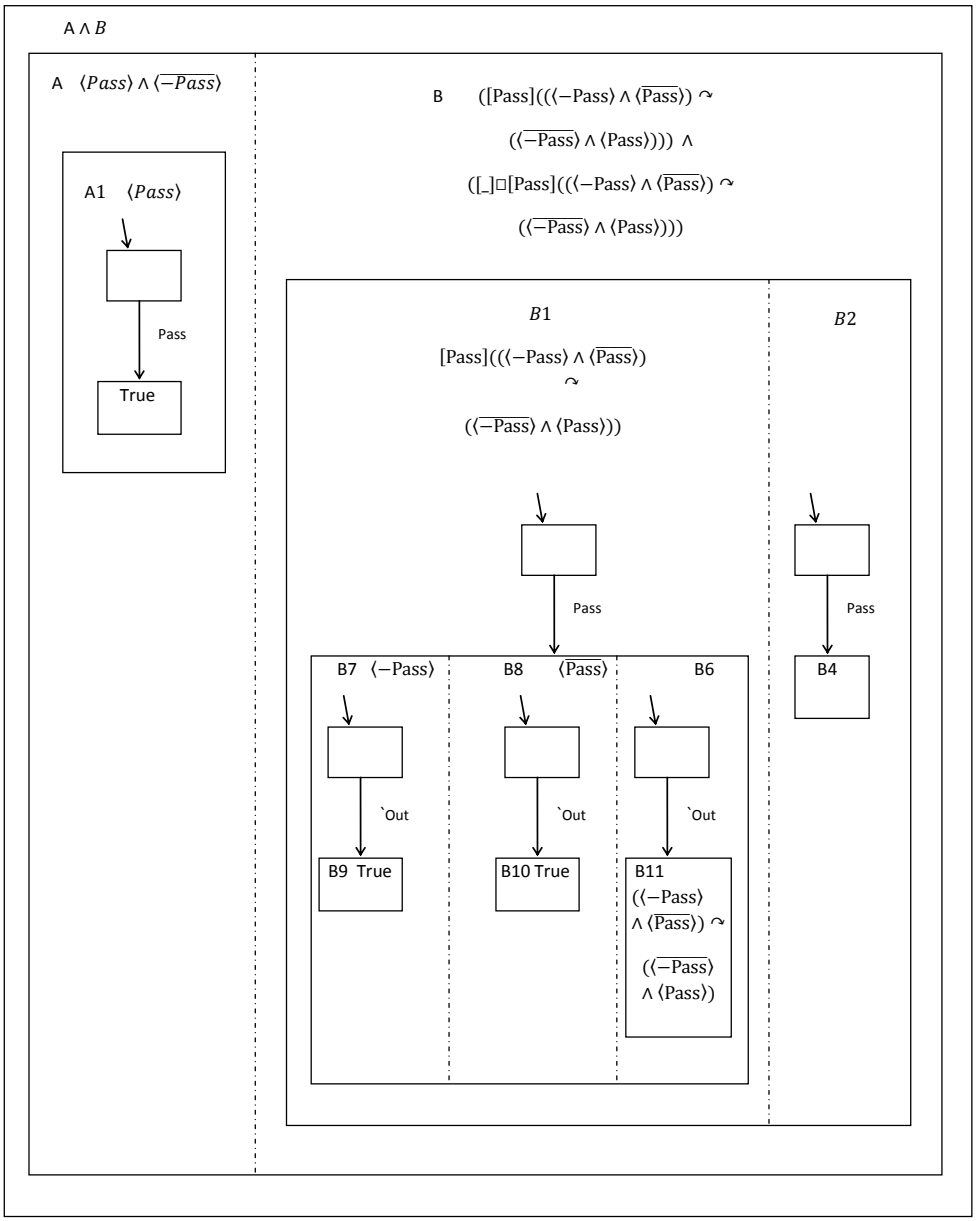

Figure 4.12: Enable, Disable and If are applied to introduce transition *'Out*

12. The pattern *Strengthen Contract* is applied to contracts *B9* and *B10*.

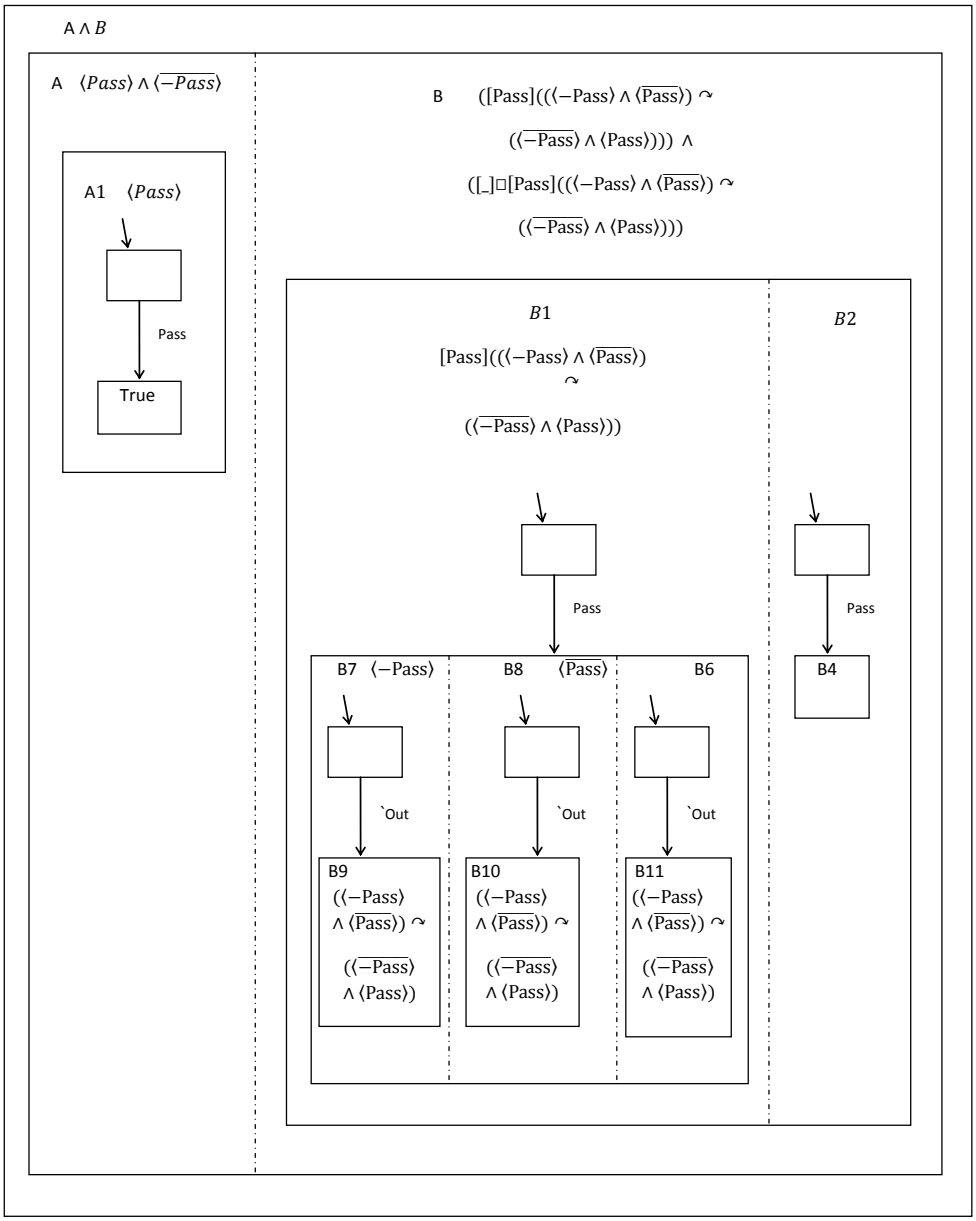

Figure 4.13: Pattern Strengthen Contract is applied to B9 and B10

13. The pattern *Conjunction Elimination* is applied to *B6*, *B7* and *B8*.

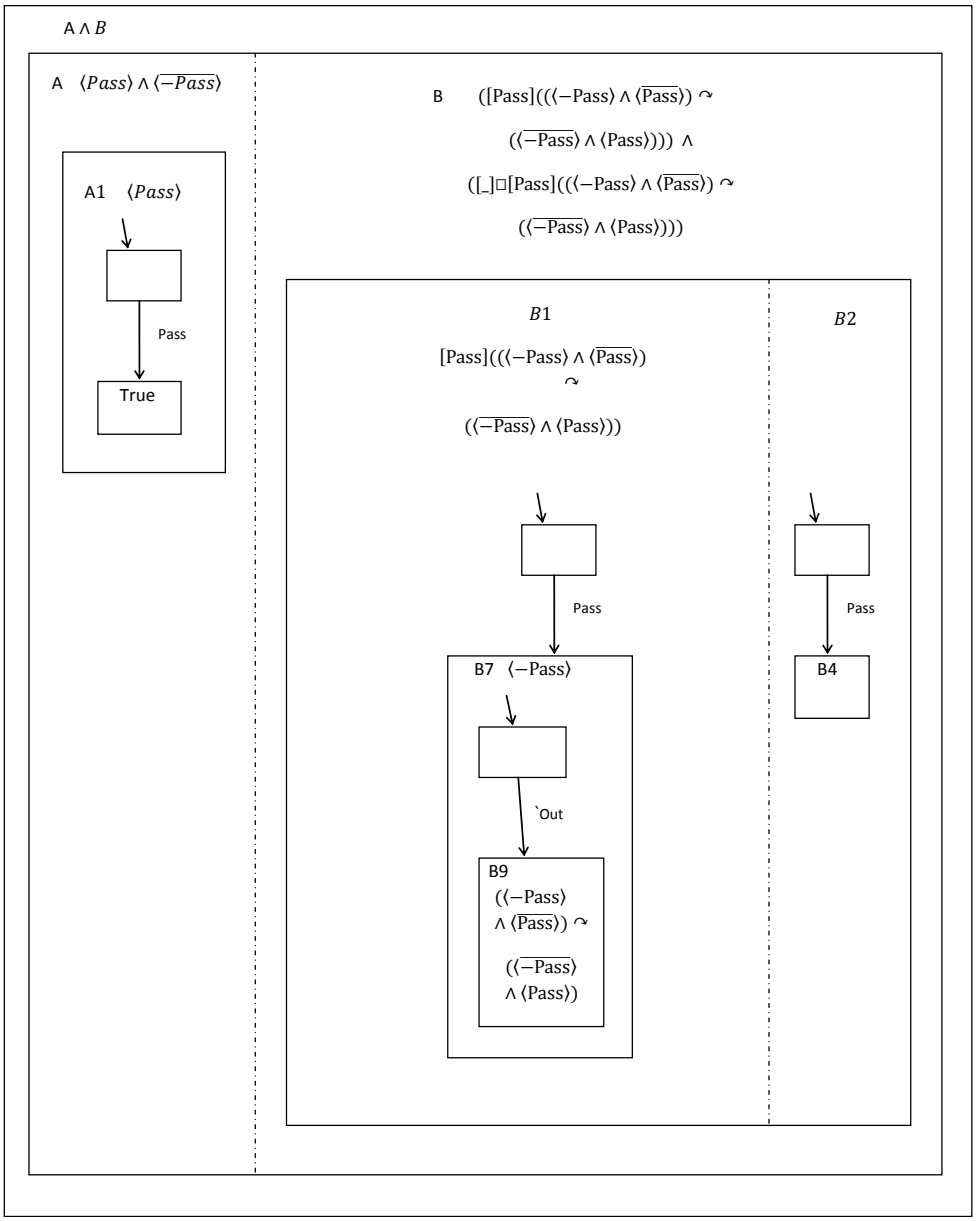

Figure 4.14: Pattern Conjunction elimination is applied to B6, B7 and B8

14. The pattern *Unfold Unless* is applied to contract *B9*.

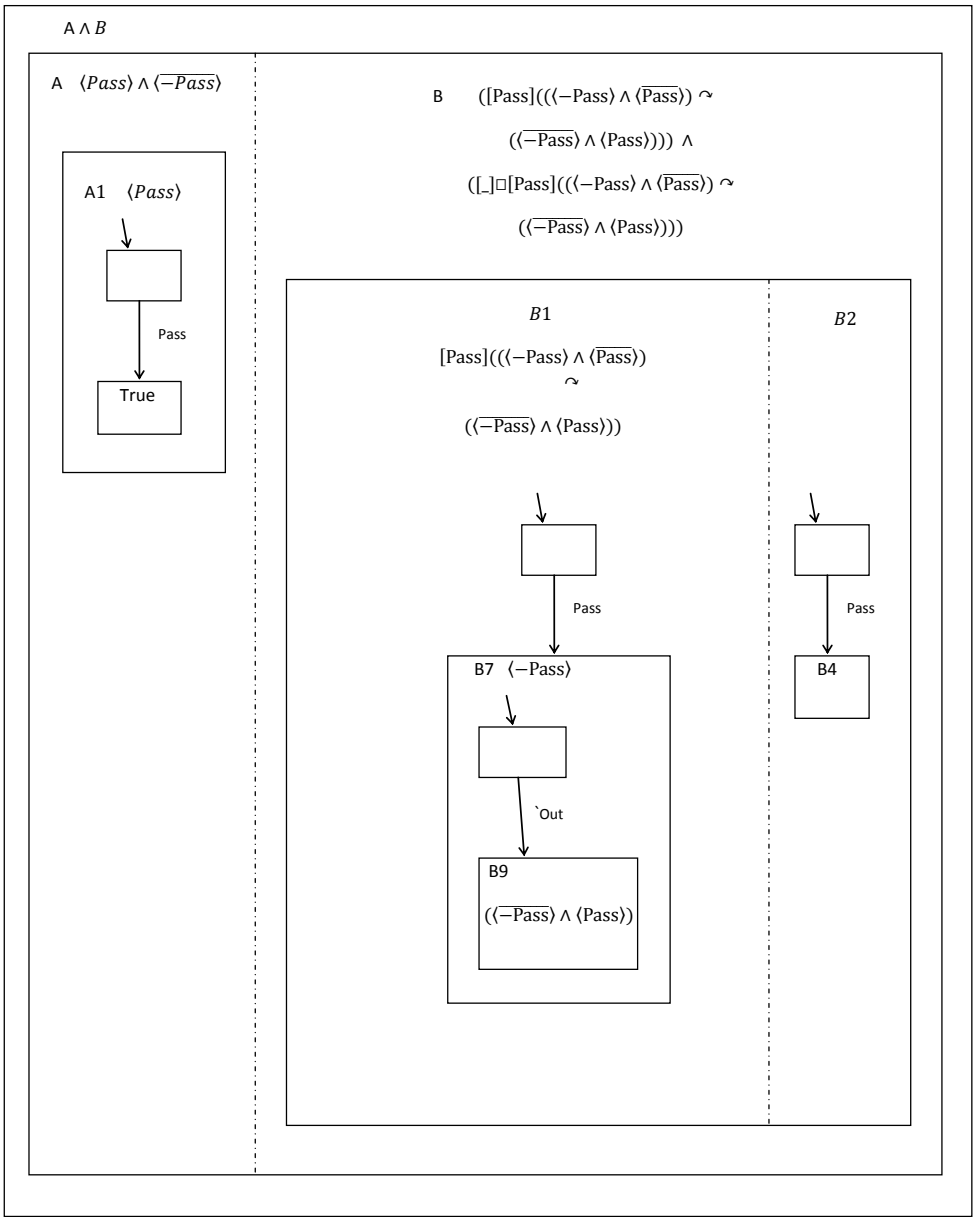

Figure 4.15: Pattern Unfold Unless is applied to B9

15. The pattern *If* is applied to contract *B4* to introduce the *'Out* transition.

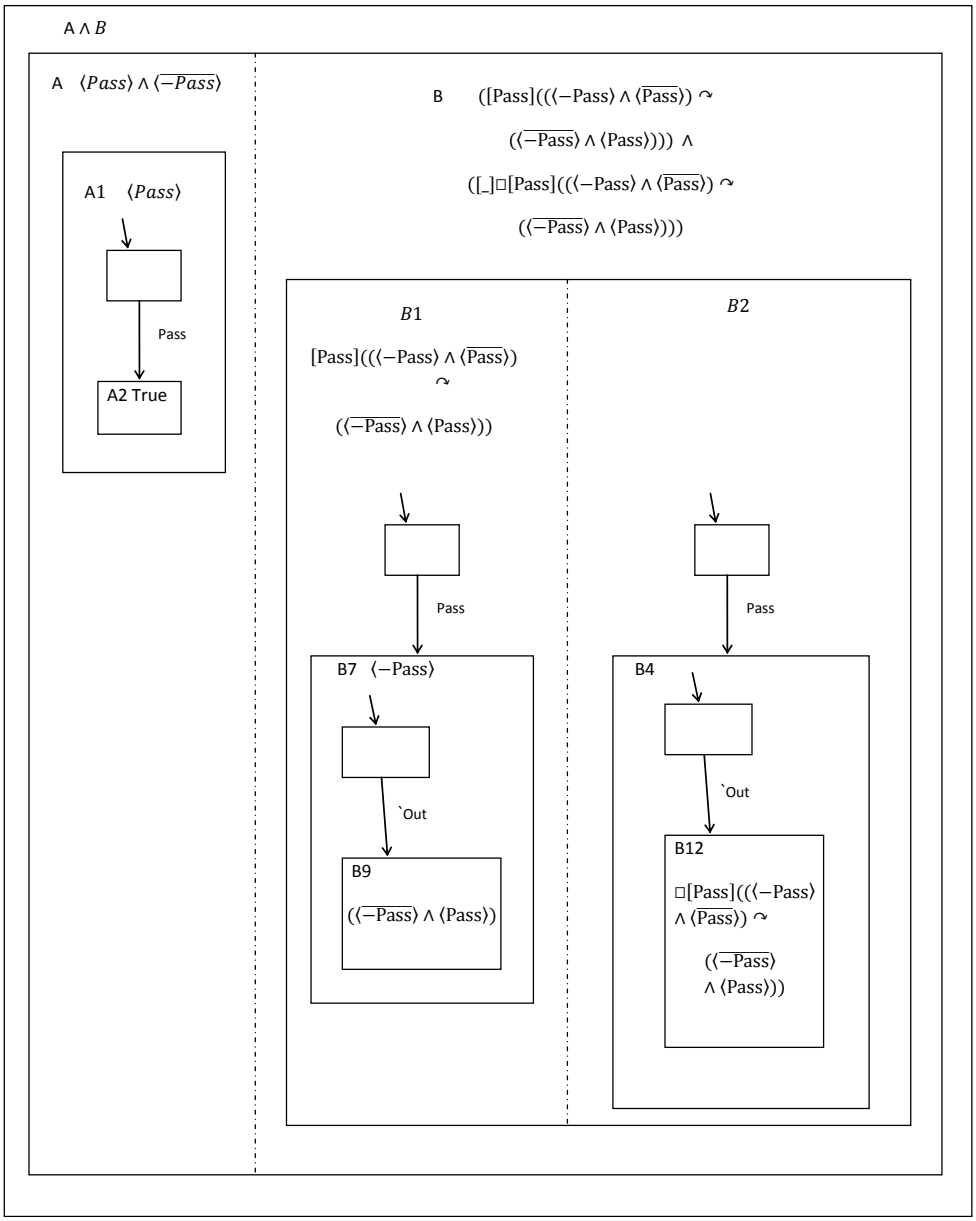

Figure 4.16: Pattern If is applied to B4

16. The pattern *Strengthen Contract* is applied to contract *B9* and *B12*.

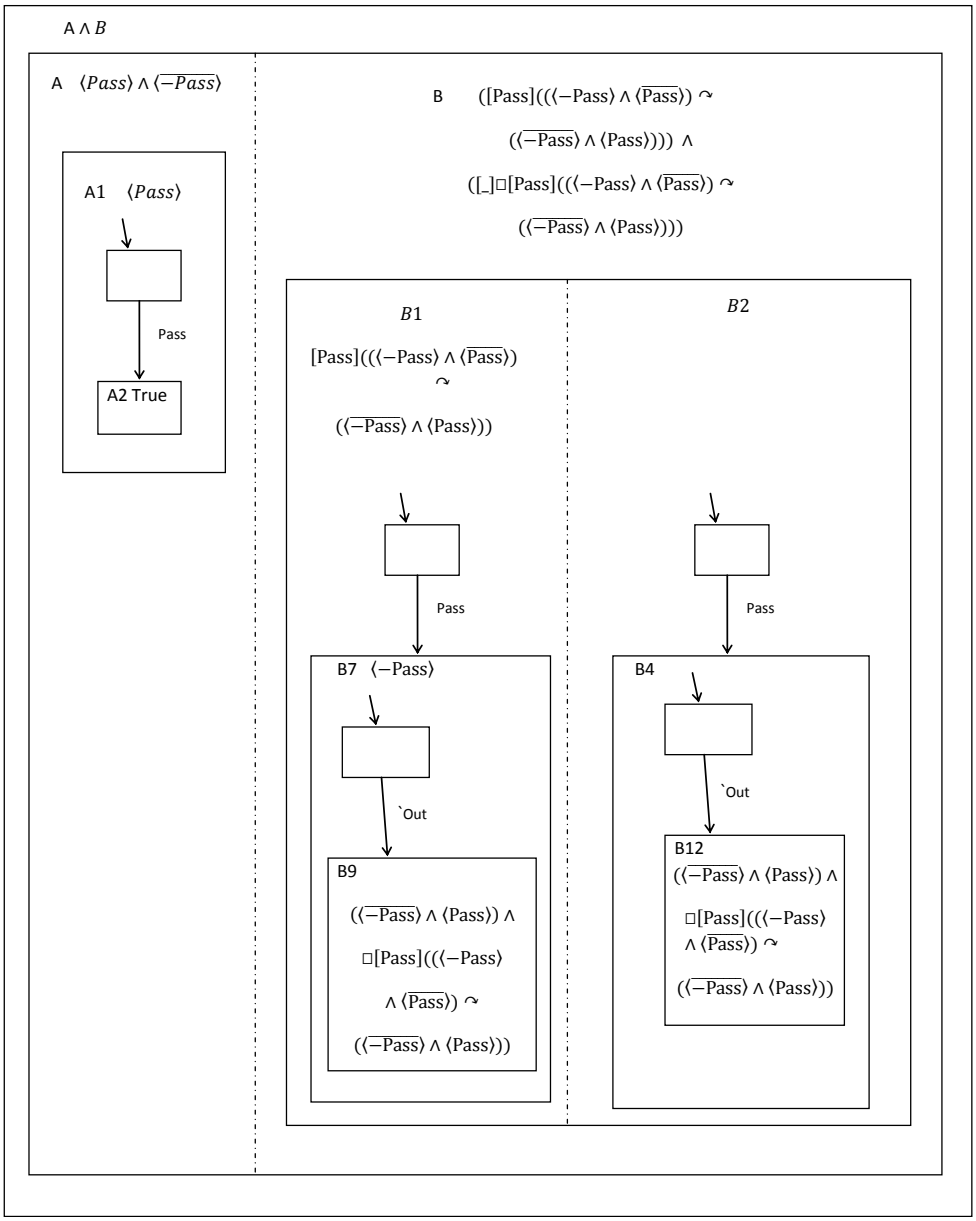

Figure 4.17: Pattern Strengthen Contract is applied to B9 and B12

17. The pattern *Conjunction Elimination* is applied to substates with contracts *B1* and *B2*.

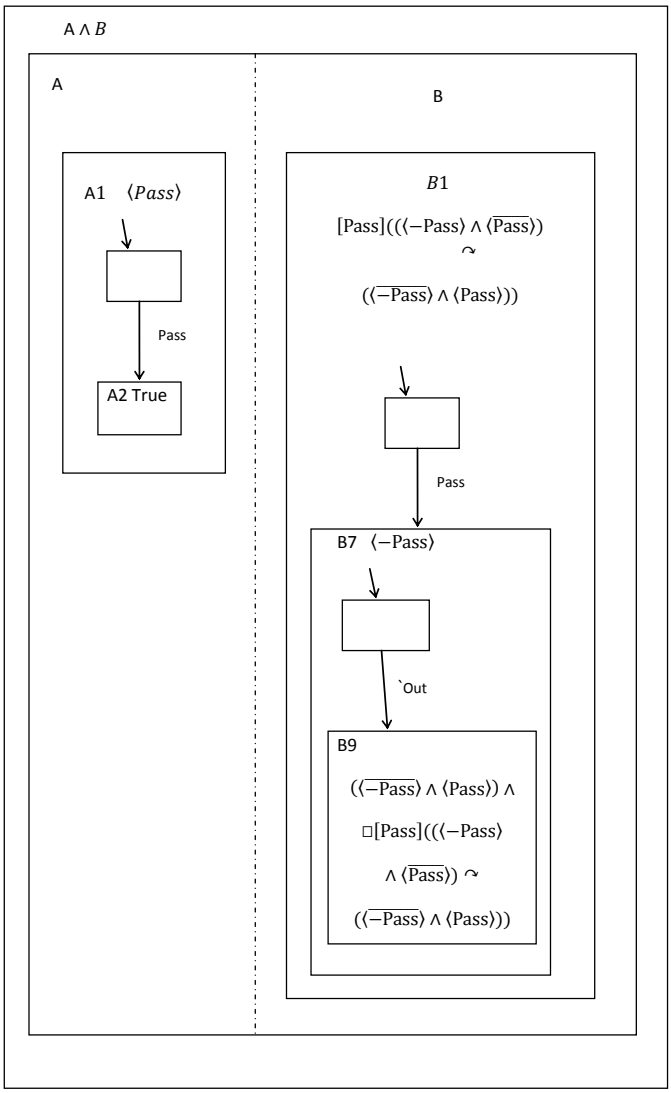

Figure 4.18: Pattern Conjunction Elimination is applied to B1 and B2

18. The pattern *Flatten Hierarchy* is applied to state with contract *B7*.

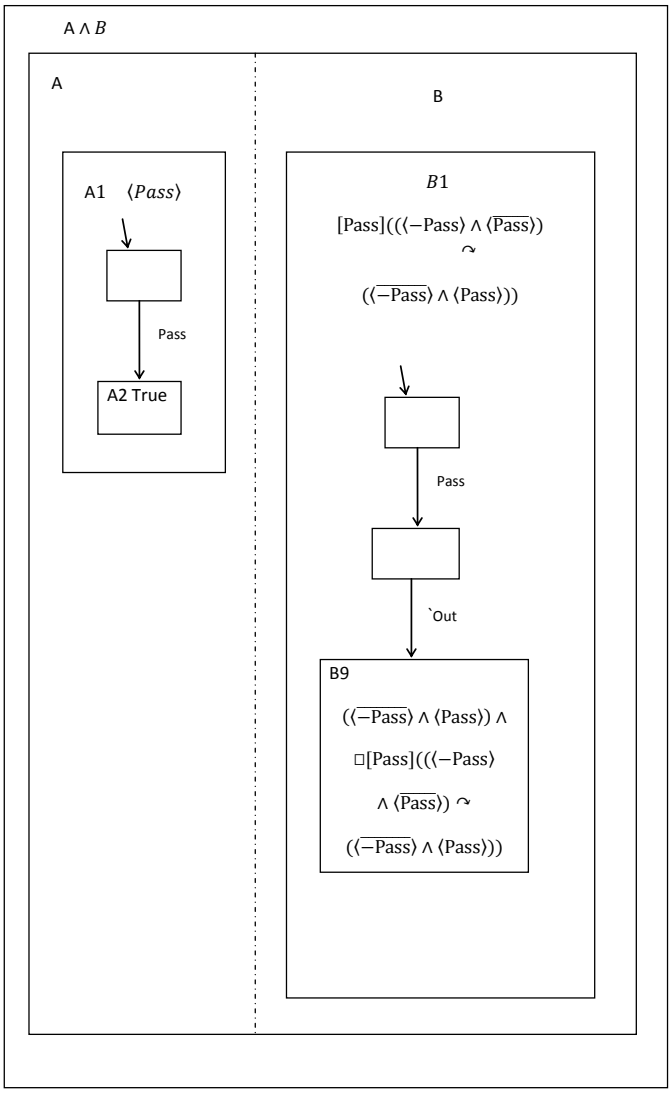

Figure 4.19: Pattern Flatten Hierarchy is applied to B7

19. Patterns are applied to the state with contract *A2*. A contract *True* can be elaborated into any state machine model and in this instance it is refined to the same model as the state with contract *B1*.

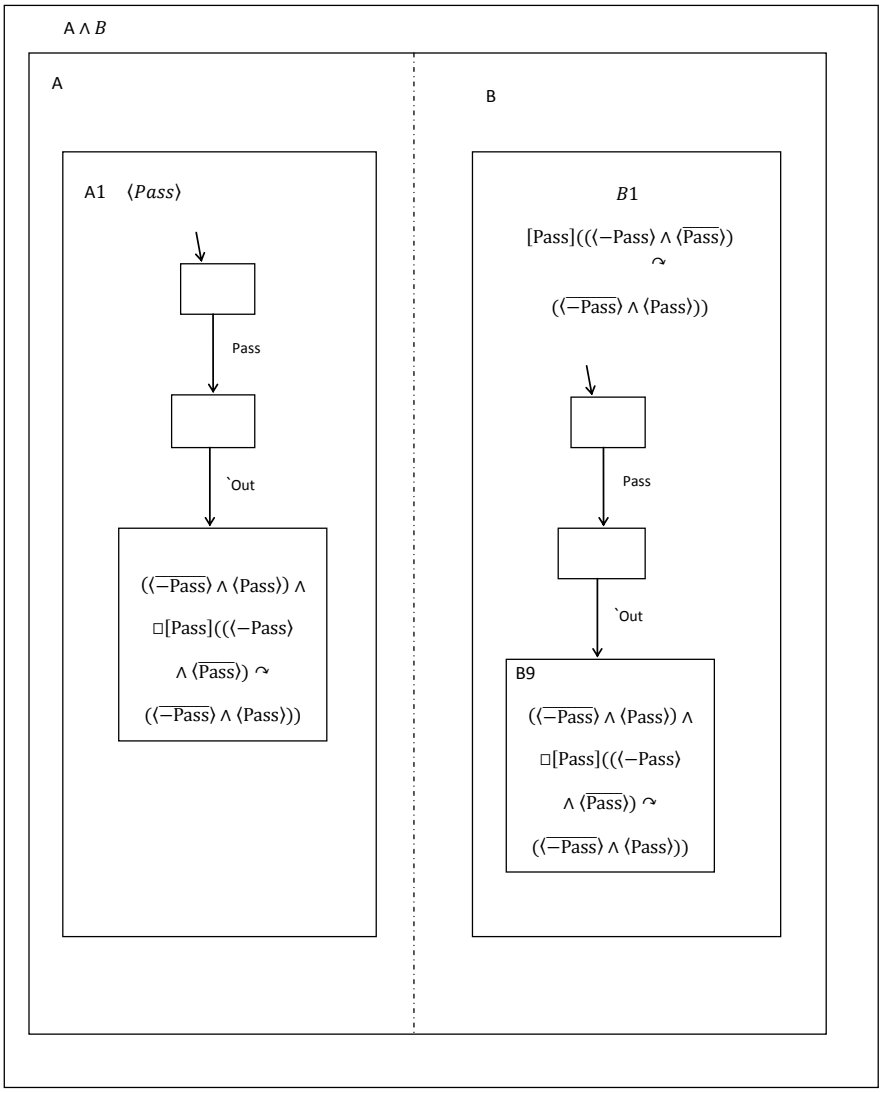

Figure 4.20: Contract A2 refined into same model as that for B1

20. Pattern *Conjunction Elimination* is applied to substates with contracts A and B.

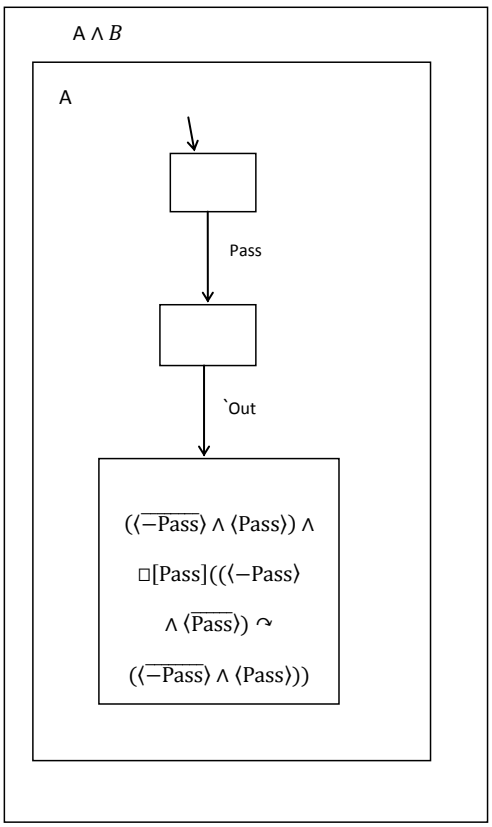

Figure 4.21: Pattern Conjunction Elimination is applied to A and B

21. Pattern *Flatten Hierarchy* is applied to state with contract A.

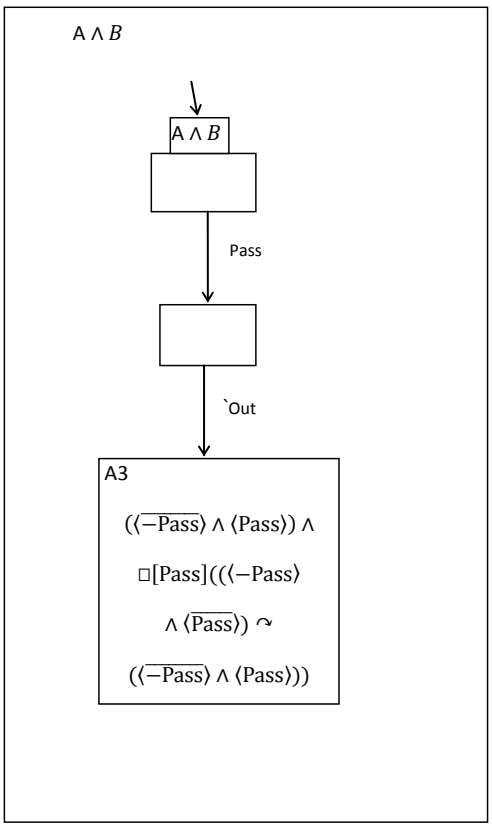

Figure 4.22: Pattern Flatten Hierarchy is applied to state with contract A

22. Pattern *Reroute* is applied to state with contract *A3*.

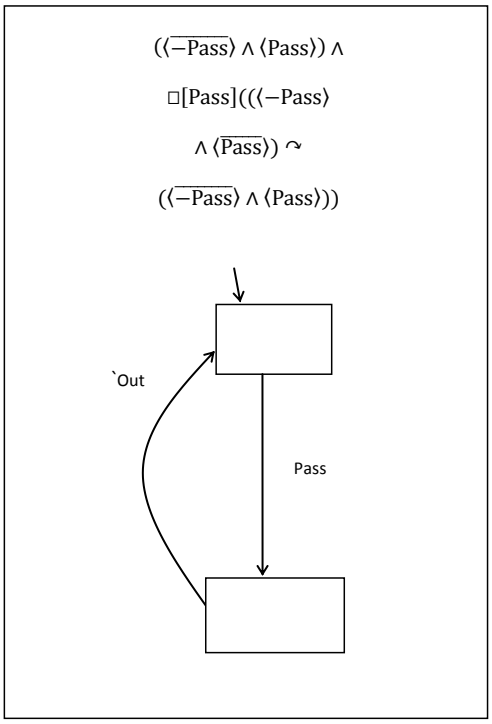

Figure 4.23: Pattern Reroute is applied to state with contract A3

The next section discusses the systematic approach adopted to identify refinement and refactoring patterns.

4.2 Systematic approach

In the remaining sections of this chapter a set of refinement and refactoring patterns that are predominately structural in nature with no semantic side-conditions are presented. This section firstly describes the method used for pattern discovery and the goals to be achieved, which will later be used to judge success when evaluating the research outcomes. Finally this section describes how the method for pattern discovery works in practice.

Inspired by the work of others, e.g. [180] the process of considering refinement and refactoring steps was "inductive" in the sense that it is assumed that there are patterns dealing with each new set of operators that may be introduced in a design step and it is unnecessary to consider refinements beyond that point as they would be elaborated

in due course. This effectively reduces an infinite number of possible refinements to a finite number of applicable steps. For example it is not necessary to consider all possible refinements of a contract $A \vee B$, only the refinements for the current step (i.e A is a refinement and B is a refinement) and assume there are refinement and refactoring patterns to subsequently take A and B on to all of their possible refinements.

Contracts may have arbitrarily many conjuncts; this makes it difficult to establish patterns for them. It is not possible to match against a subset of the conjuncts and rewrite those because the conjuncts that have been disregarded might constrain the solution space in a way that has not yet been anticipated. Patterns can be specified to deal with a few cases but new patterns would be required for every new contract with an extra conjunct that had not been previously considered.

Thus a better approach is required to establish patterns for contracts with conjuncts. An effective solution for refining a contract with conjuncts is to compose the conjuncts together using the diagrammatic conjunction operator (&) discussed in Chapter 2. A conjunction state is introduced for each conjunct in the contract.

For example if the contract has the form $L \wedge M$ two new substates are introduced one with inner contract L and the other with inner contract M. The new substates are *conjunction* states. As described previously they are an addition to the state machine language and a new form of state decomposition required to support the approach to the design process. Conjoined substates are orthogonal in the same way as parallel states. Conjunction states represent the behaviour *common* to all their substates.

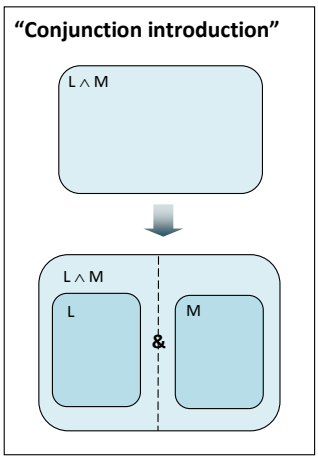

Figure 4.24: Conjunction introduction

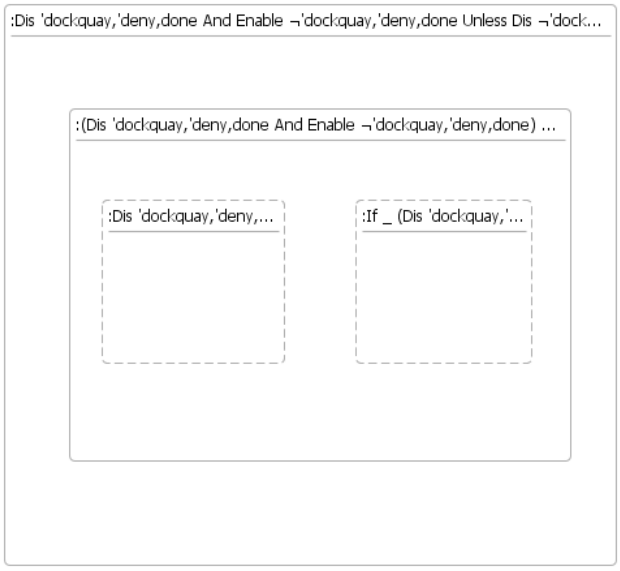

Figure 4.25: Introduce conjunction states for each conjunct in the contract

The design process then proceeds by refining each conjunct separately to mutually consistent (usually identical) designs and finally removing the conjunction operator and identical conjuncts (apart from one). The composite conjunction state containing the equivalent substates is replaced with one of the conjuncts and the copies are thrown away.

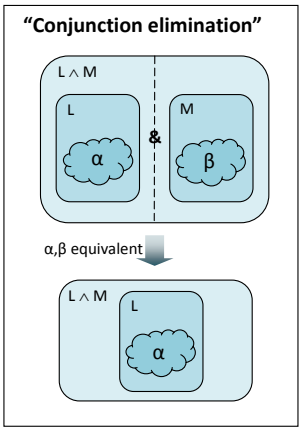

Figure 4.26: Equal

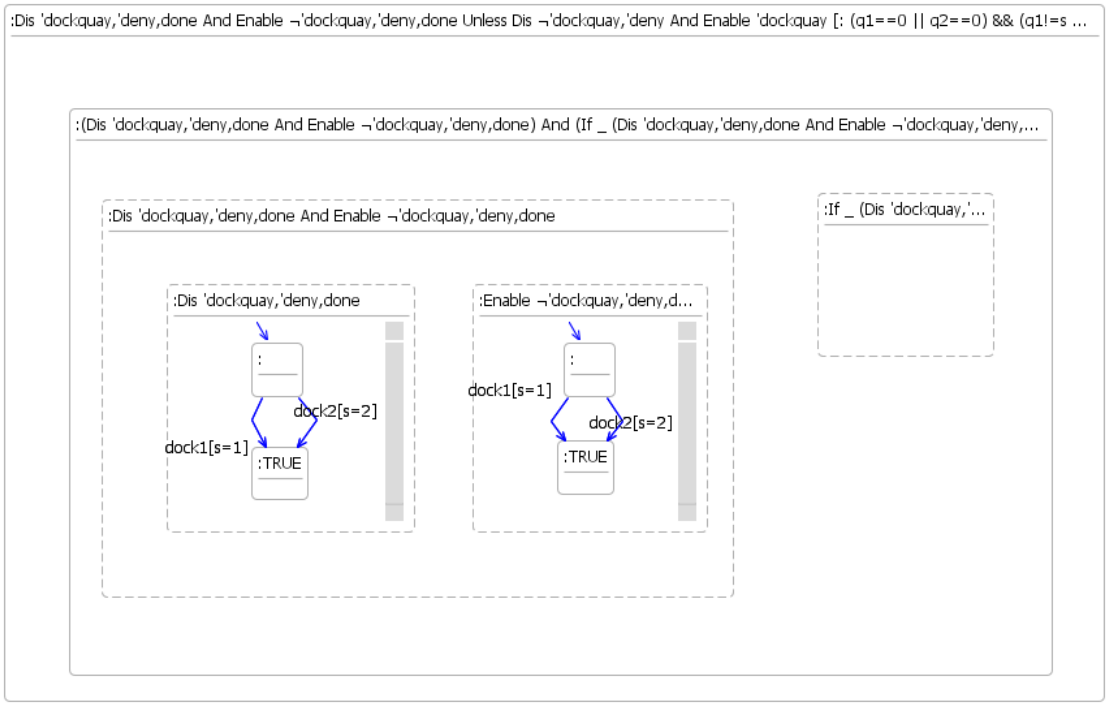

Figure 4.27: Conjuncts refined to equivalent designs representing the common behaviour

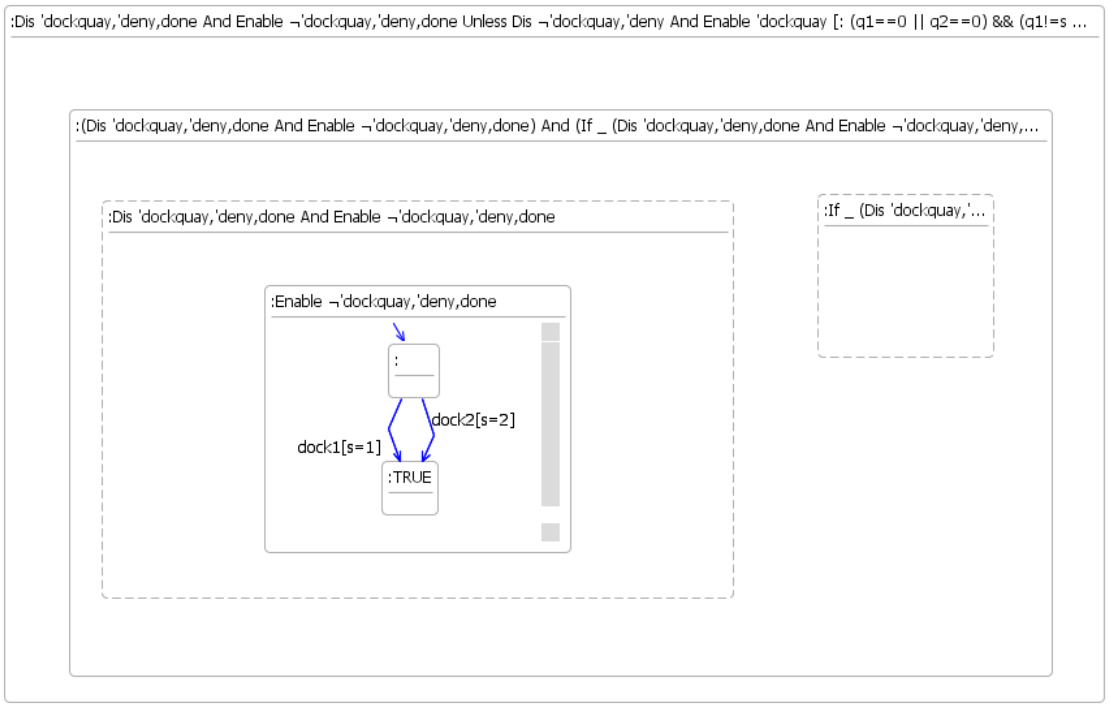

Figure 4.28: The duplicate conjunction states are eliminated

If no common behaviour exists then the contract is infeasible and there is no fully developed state machine (i.e containing no contracts) that implements it. It would be desirable to check contracts for feasibility initially (e.g as an optional side-condition to the conjunction pattern) otherwise the refinement may never succeed as a *False* contract is unimplementable. *True* describes all behaviours, it can be refined into anything including deadlock and is at the top of the refinement ordering. *False* describes no behaviours, it refines everything and is at the bottom of the refinement ordering. *False* is the empty set of behaviours, it is unimplementable and cannot be refined any further. (It excludes deadlock as deadlock is a type of behaviour).

A systematic approach has been adopted to identify patterns. It was important to follow a systematic method for pattern identification to help achieve a complete set in terms of coverage of transformations, and ensure that the catalogue consists of a comprehensive set of patterns for the common types of refinement or refactoring that are frequently required during the stepwise design of a system.

There are three main stages to the approach. The first stage aims for completeness of the set of patterns by ensuring operator coverage and consideration of refinement and refactoring rules for statecharts in the literature. The second stage of the approach concerns the specialisation and generalisation of patterns. Specialisation leads to a simpler side-condition and generalisation to better coverage. The final stage of the approach was to verify the integrity and utility of the proposed set of patterns through a case study which is presented in Chapter 7.

4.2.1 Aims

General aims for the pattern catalogue are for compactness, (maintaining a small catalogue size), and pattern comprehensibility to ensure they are as simple as possible to understand and apply. There is also a desire for patterns that encourage well-formed designs (e.g. without disconnected states). Patterns must be of practical utility for the engineer. This criterion can be fulfilled in many ways, for example a pattern for a frequently required refinement or refactoring step, or a pattern that is valuable in terms of proof reduction. A pattern that simplifies a design (e.g. by combining states) or simplifies the refinement process (e.g a pattern that combines refinement/refactoring steps) is also of practical utility.

Other important considerations were to keep to a minimum patterns for general cases

where the reasoning would be computationally intensive, incomplete or unavailable. There is a trade-off and a balance to be struck between having a few general patterns and many specific patterns. Although specific patterns minimise proof burden and usually require less information from the user, they can potentially lead to a multitude of patterns which may be difficult for the engineer to navigate. General patterns, as stated above, may have more complex side-conditions, which are potentially harder for the engineer to supply information for, and for the model checker to discharge.

The aim is for a pattern catalogue that consists of a set of patterns that cover specific cases, which require little by way of accompanying proof, with a few general patterns in a supporting rather than an essential role should the more specific patterns not cover what is required. The aim is to ensure the completeness, correctness and utility of the patterns. The completeness criteria discussed here concerns coverage of transformations and ensures that the catalogue consists of a comprehensive set of patterns for the common types of refinement or refactoring that are frequently required during the stepwise design of a system. A systematic approach to the discovery of patterns helps to fulfill the completeness criteria.

The case study in Chapter 7 provides evidence of pattern utility. A strong argument that the patterns are correct is that they have been inspected by peers, including those with expertise internal to the department, collaborators, supervisors, and peer reviewers of the publication. They have been used successfully on a case study, which gives us confidence that they are correct. Correctness is also indicated by familiarity/similarity with patterns from related research.

Further work can be carried out in the future to increase confidence that the patterns are correct. For the patterns that have not been preproven, for example, the model checker (HST) could compute the refinement relation, on a case by case basis for instances of a pattern being applied in order to verify that it did indeed produce a correct refinement. At the time of writing, the formal underpinning was not available in a stable, complete form to attempt formal proof and the CoSta project's model checker does not yet support the refinement check.

The aim is for a pattern to ensure consistency between designs so that the refinement or refactoring step preserves meaning. As explained above this has informally been checked for each pattern by comparing the semantic interpretation of the target model with that of the source model and ensuring that the ready sets do not change for every possible

data state and new nondeterminism is not introduced. The discharge of side-conditions will guarantee consistency is maintained by a refinement or a refactoring pattern and the model checker (HST) will provide a mechanism for discharging such side-conditions, should the pattern require it. Consistency of the design process ensures that only designs can be achieved that preserve the original high-level properties of the contract. Consistency of the design process is guaranteed by the fact that it is based solely on pattern application and patterns ensure consistency.

4.2.2 Systematic approach Stage 1

As described above there are three main stages to the systematic approach to identify patterns. The first stage, discussed in this section, aims for completeness of the set of patterns by ensuring operator coverage and consideration of refinement and refactoring rules for statecharts in the literature.

The identification of patterns was based, in the first instance, on an analysis of the operators of the Contractual State Machine language and the applicable refinement and refactoring steps were considered for each. Being systematic over the operators led to identification of patterns covering all of the different possible types of source and target models for a transformation (i.e *contract to contract*, *contract to mixed design* and *mixed design to mixed design*).

The design process moves a design from a contract specification to a state machine model by applying refinement and refactoring patterns that preserve meaning. The aim was to identify model transformations to go from one syntax (for contracts) to another (for state machines) or within a single language (for contracts or state machines) that preserve meaning. One way this was achieved was by considering all of the operators of the contract and state machine languages as possible sources or targets of transformations, to try to establish the smallest permissable transformations that preserve meaning. Basing the systematic discovery of patterns on the language syntax was a starting point for identifying atomic, small granularity patterns. Compound patterns that combine the atomic patterns could be considered later on and may be more meaningful for the engineer.

Basing the discovery of patterns on syntax and operator coverage was one of several ways of ensuring completeness of the patterns (e.g. it is no use having a syntax construction for which no patterns apply). However there were other methods used in the discovery of patterns, discussed in this and the next section. They include the empirical

practice of applying the patterns to a problem in the case study, and the specialisation and generalisation of the patterns. Both strategies helped to decide the optimal set of patterns (e.g. by removing two patterns and replacing them with a more general one, or augmenting a general pattern with specific instances which were used often, easier to use or had simpler side-conditions).

Pattern identification also began with broad research into the numerous formal languages with refinement calculi [17, 22, 23, 58, 70, 107, 112, 126, 180, 224, 255, 256, 260]. There are common types of refinements and refactorings for statecharts described in the literature. Typically, the refinement process begins with an "underspecified" state machine design. Underspecification in this context can mean partiality (missing diagram elements) and abstraction (nondeterminism). Development proceeds by the application of refinement and refactoring laws which enable the elaboration of a design (e.g to add new states and transitions) provided new nondeterminism is not introduced. A list of the most common refinements and refactorings for statecharts described in the literature is presented next. The majority of them are equivalence laws. The refinement and refactoring laws vary for each statechart variant as they are specific to the underlying semantics of the particular language and refinement theory. Therefore the descriptions below are limited to details of the kinds of transformations the rules apply.

Refinements and refactorings for statecharts described in the literature.

Number	Name	Transformation
01	Introduce orthogonality	Creates a set of orthogonal states [253] [94].
02	Remove orthogonality	Removes a set of simple (empty) orthogonal states [253] [94].
03	Create a composite superstate	Groups states into a new composite state. The transformation encloses a group of selected states with a new superstate [242] [94].
04	Remove a composite superstate	Removes a composite state which has no incoming or outgoing transitions that encloses a group of substates [94].
05	Move target down	An incoming transition to a composite state has its target state changed from the composite state to the default/initial state of the composite [253] [166] [94].
06	Move target up	Changes the target of an incoming transition from the initial state of the composite to the composite state [253] [94].
07	Move source up	Replaces a set of identical outgoing transitions, one from every substate of a composite, with a single transition from the composite [253] [166] [242] [94].
08	Move source down	Replaces a single outgoing transition from a composite state with a set of outgoing transitions one from each substate of the composite [253] [242] [94].

Table 4.1

09	Move state into composite	Moves a state into one of its sibling states, (a state at the same hierarchical level). The proviso is that the state to be moved must not be a default/start state. It must have no incoming or outgoing transitions. The sibling state must not have outgoing transitions [242].
10	Move state out of composite	Moves a state out of a composite state, so that it is contained within the same parent state as the composite state. It is no longer a substate of the composite, but at the same hierarchical level as the composite state. The proviso is that it must not be a default/start state. It must have no incoming or outgoing transitions and the composite must have no outgoing transitions [242].
11	Move action, state to transition	This refactoring is applicable to statechart languages that allow states to have entry and exit actions e.g. [248]. It moves a state entry action from a state to its incoming transitions. For example if the state entry action is to increment a variable ($entry : x := x + 1$) this action is removed from the state and added to the actions for each incoming transition to the state. If the state has an existing incoming transition $e1/y := 3$, the transition label becomes $e1/y := 3; x := x + 1$. Similarly this refactoring moves a state exit action from a state to its outgoing transitions. For example if the state exit action is to increment a variable ($exit : x := x + 1$) this action is removed from the state and added to the actions for each outgoing transition from the state. If the state has an existing outgoing transition $e2/y := 4$, the transition label becomes $e2/x := x + 1; y := 4$ [242].

Table 4.1

12	Move action, transition to state	This refactoring is applicable to statechart languages that allow states to have entry and exit actions e.g. [248]. It moves an identical action from every incoming transition to a state to the state's entry action. For example for the incoming transition $e1/y := 3; x := x + 1$, if the identical action is $x := x + 1$ this is removed from the transition which now becomes $e1/y := 3$, and is added as a state entry action $(entry : x := x + 1)$. Similarly this refactoring moves an identical action from every outgoing transition from a state to the state's exit action. For example for the outgoing transition $e2/x := x + 1; y := 4$, if the identical action is $x := x + 1$ this is removed from the transition which now becomes $e2/y := 4$ and added as a state exit action, $(exit : x := x + 1)$ [242].
13	Modify existing transitions	Permits changes to the guard and events of a transition provided no new nondeterminism is introduced [221].
14	Combining and splitting transitions	Combines/splits transitions on guards and events [253] [221] [94].
15	Remove a transition	A transition is removed if an alternative transition exists reducing nondeterminism [216] [166] [221].
16	Add a transition	A transition is added if no corresponding transition exists so that it is not introducing new nondeterminism [216] [166] [221].
17	Remove/add a transition	A transition is added that can never be fired, as it is untriggerable due to a *False* guard [94] [148].
18	Add a new state	A state is added, this usually means a state with no incoming or outgoing transitions [216] [166] [221].
19	Replace a basic state	A basic state is replaced with a composite state (with substates) [216] [166].

Table 4.1

20	Remove an unreachable state	An unreachable state is removed, this usually means a state with no incoming or outgoing transitions [216] [166].
21	Introduce hierarchy	Groups a set of basic states with identical outgoing transitions (to the same target state and with the same labels) into a new composite state. The refactoring replaces the identical outgoing transitions from each substate with a single outgoing transition from the composite state [148].
22	Combine states	Two states are combined, typically the states are basic states and must belong to the same parent state and they are replaced by a single basic state [253] [94].
23	Split state	A state is split, typically the state is a basic state that is replaced by two new basic states. The combined set of incoming and outgoing transitions from the two new states must be equivalent to the set of incoming and outgoing transitions from the original state [253] [94].

Table 4.1: Refinements and refactorings for statecharts described in the literature

Although many of the refinements and refactorings for statecharts from the literature were not applicable to CoSta, such as those relating to entry/exit actions, the study provided a valuable check list on which to base the diagram to diagram patterns for this thesis.

4.2.3 Systematic approach Stage 2

The last section described the first stage of the systematic approach to identify patterns, which aims for completeness of the set of patterns by ensuring operator coverage and consideration of refinement and refactoring rules for statecharts in the literature. The second stage of the approach discussed in this section concerns the specialisation and generalisation of patterns. Specialisation leads to a simpler side-condition and generalisation to better coverage.

The two stages are illustrated together in what follows, starting with a specific pattern to merge two transitions.

Version 1. *If there are two transitions between the same states, with the same triggering event and no guards or assignments, they can be combined and no other constraints need to be validated.*

A specific form of a pattern was considered initially followed by successively more general versions. The specific and general versions were compared to identify any benefits of the more specific version. The aim was to disregard the more specific version unless its presence could be justified, for example, in terms of simplifying the proof burden or providing a methodological benefit, being of practical use to an engineer.

For example, there is a methodological benefit when the more specific form of the pattern is frequently needed during the design process and it requires less in terms of information from the user than a more general version. Simplifying proof burden justifies inclusion of the pattern in the catalogue, even though there is a model checker to automate the evaluation, as it reduces the required levels of resources (time to compute, memory requirements) and in the case of a refinement check the algorithm may not succeed (as it requires too much memory or time to compute). In the software tool when diagram components are analysed to enable and disable the applicable patterns, time constraints apply, and it may not be practical to decide deeper semantic constraints. There is an advantage to having the simpler forms of patterns, because they enable the software to determine which patterns are applicable in real-time.

Version 2 (more general). *If there are two transitions between the same states, with the same events and assignments but possibly different guards, the transitions can be combined by disjoining guards, and no other constraints need to be validated.*

There is no benefit of version 1 over version 2 in terms of proof so it can be replaced by version 2.

Version 3 (more general). *If there are two transitions between the same states, with the same events but possibly different guards and syntactically different assignments shown to be semantically equivalent, the transitions can be combined by disjoining guards and including the assignments from either transition.*

A side-condition is required to prove that the syntactically different assignments are semantically equivalent. This is more general but carries a proof burden so there is a benefit of version 2 over version 3 in terms of proof and thus an argument to justify its presence

in addition to the more general version.

The final stage of the systematic approach to pattern identification is to verify the integrity and utility of the proposed set of patterns through a case study, this is presented in Chapter 7. The systematic approach adopted for pattern discovery and the general aims for the patterns have been discussed. The next section presents a set of patterns for CoSta for the refinement and refactoring steps that are predominately structural in nature with no semantic side-conditions. The more elaborate patterns are presented in the next chapter. Typically these are transformations with side-conditions (e.g relating to conditions/actions on data) that require the model checker (HST) to discharge them.

4.3 Basic patterns

The set of basic refinement and refactoring patterns for Contractual State Machines are presented in this section. The strategy for pattern discovery was to consider the simpler types of patterns first that are predominately structural in nature with no semantic side-conditions. They do not refine the data and are thus potentially more straightforward than the refinement and refactoring patterns that do (these are presented later on in the next chapter) as they do not need to consider the permissible changes to the data that preserve the refinement relation.

Conceptually a refinement or refactoring pattern corresponds to an elaboration of a more abstract model that preserves the refinement relation (and in the case of a refactoring the stronger equivalence relation). The aim is for a pattern to ensure consistency between designs so that the refinement or refactoring step preserves meaning. Described in terms of its implementation and use in practice a pattern expresses a refinement or refactoring step as an update-in-place model transformation. A pattern is effectively a pair of abstract and concrete templates, so that when a particular Contractual State Machine or part of one under investigation matches the abstract template then it may be refined by the concrete template.

Refinement and refactoring steps in the literature are classified as either refinement or equivalence transformations (refactorings) [94, 221, 253]. It is useful to add some further classifications for the refinement and refactoring patterns in this thesis. Patterns can be classified based on the types of source and target models of the transformation, i.e *contract to contract*, *contract to mixed design* and *mixed design to mixed design*. Other classifications are *general* or *specific* and *syntactic* or *semantic*. A *specific* pattern, with respect

to a more *general* pattern, is a pattern constrained for use in fewer situations. Specific patterns may require less information from the user or have simpler side-conditions. *Syntactic* patterns are predominately structural in nature with no semantic side-conditions. Whereas *semantic* patterns are transformations with side-conditions (e.g relating to conditions/actions on data) that require the model checker (HST) to discharge them.

The primary classification for the patterns presented next is based on the types of source and target models of the transformation (i.e *contract to contract*, *contract to mixed design* and *mixed design to mixed design*). Additionally, the patterns are described as being either refinement or refactoring. A *refinement pattern* is one that reduces nondeterminism in a specification and a *refactoring pattern* changes the structure of a specification without affecting its behaviour. It is not always possible to classify some of the patterns in the catalogue as exclusively refinement or refactoring.
The description for each pattern covers the following details:

1. The rationale for the pattern.

2. The pattern constraints restricting the situations in which it can be applied (e.g. syntactic constraints and semantic side-conditions).

3. The parameters of the pattern.

4. The model transformation applied with a diagrammatic illustration.

A presentation of the refinement and refactoring patterns which are predominately structural in nature with no semantic side-conditions follows. All of the patterns minimise proof burden. The refinement and refactoring patterns encompass the necessary steps to get from contracts to everywhere in the design space that implements them. Appendix D.1 has some additional patterns that are not in the core set as they are not critical or strictly necessary to achieve a deterministic solution but may be preferable for stylistic reasons offering the engineer different routes to the same solution.

4.3.1 Contract to contract patterns

Contract to contract refinement and refactoring patterns are presented in this section. These are patterns where both the source and target models of the transformation are contracts.

4.3.1.1 Patterns for adding new parts to a CoSta contract

The pattern theme in this section is *adding new parts to a contract expression.*

Name: *Strengthen contract, add conjunct*

Type: Refinement

Rationale: The Conjunction operator ($\wedge$) specifies that more than one property is required to hold. This pattern permits the state's inner contract to be strengthened by adding another conjunct to the contract expression. The rationale for this pattern is that an engineer may want to strengthen a contract to reduce nondeterminism or manipulate the contract so that it is equivalent to another contract (e.g. to enable the *Reroute* pattern to be applied). The pattern was identified by considering permissible changes to a contract that would result in a refinement under SVRS. Strengthening the inner contract of a state is a refinement under SVRS.

Constraints: An optional side-condition is that the new contract is feasible (is not equivalent to the *False* contract) which could be verified using the model checker. The side-condition is optional because it does not affect the correctness of the pattern application but saves the user from trying to refine a pattern which has no implementation.

Parameters: The parameters are a state with inner contract C_1 and the new contract component C_2 to be conjoined.

Transformation: This pattern refines the contract C_1 to the contract $C_1 \wedge C_2$, where C_2 is provided as a parameter.

Diagram:

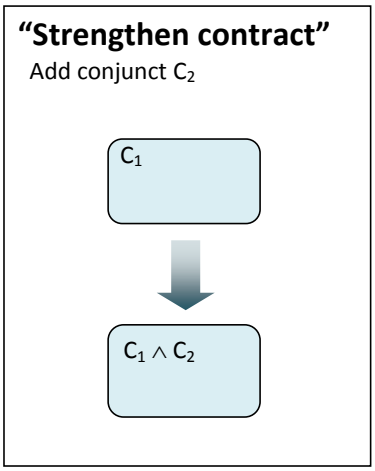

Figure 4.29: Strengthen contract

4.3.1.2 Patterns for removing parts of a CoSta contract

The pattern theme in this section is *removing parts of a contract expression.*

Name: *Strengthen contract, remove disjunct*

Type: Refinement

Rationale: A choice of behaviours is expressed with the Disjunction ($\vee$) operator. This pattern permits the state's inner contract to be strengthened by removing a disjunct from the contract expression. The rationale for this pattern is that an engineer may want to strengthen a contract to reduce nondeterminism or manipulate the contract so that it is equivalent to another contract (e.g. to enable the *Reroute* pattern to be applied). The pattern was identified by considering permissible changes to a contract that would result in a refinement under SVRS.

Constraints: The pattern ensures that the inner contract is a *Disjunction* operator expression of the form $C_1 \vee C_2$.

Parameters: The parameters are a state with an inner contract that is a *Disjunction* operator expression and the contract component to be removed, i.e. the left or right side of the disjunction operator.

Transformation: This pattern refines the contract $C_1 \vee C_2$ to the contract C_1 or the contract C_2.

Name: *Merge conjunct*

Type: Refactoring (a refinement in its most general form).

Rationale: The Conjunction operator ($\wedge$) specifies that more than one property is required to hold. The purpose of the pattern is to simplify the design process by reducing the number of conjuncts to be refined within a contract. The pattern merges conjuncts to remove redundant behaviour (repetition), where the behaviours are the same. This pattern could be accompanied by an inverse, but the inverse transformation (to add a conjunct to a contract that is syntactically equivalent to an existing conjunct) may not be a useful refactoring step when designing Contractual State Machines.

Constraints: The pattern ensures that the inner contract is a *Conjunction* operator expression of the form $C_1 \wedge C_2$. Additionally it verifies that the conjuncts, C_1 and C_2 are syntactically equivalent. The pattern could be generalised to cover the situation where the contracts are syntactically the same apart from the update expressions. The pattern con-

straint could be extended to verify that in addition the corresponding update expressions from both conjuncts are semantically equivalent. To determine the semantic equivalence of two update expressions $[x : P_1]$ and $[x : P_2]$ the side-condition is $\vdash P_1 \Leftrightarrow P_2$. The pattern constraint could be generalised further to check if a refinement relation holds between C_1 and C_2 and the weaker conjunct removed.

Parameters: The parameter is a state whose inner contract is a *Conjunction* operator expression.

Transformation: This pattern refines the contract $C_1 \wedge C_2$ to the contract C_1.

Diagram:

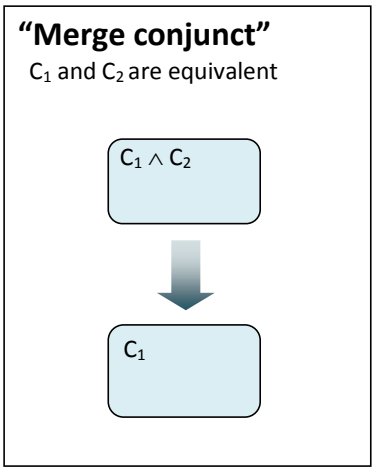

Figure 4.30: Merge conjunct

Name: *Unit*

Type: Refactoring

Rationale: The purpose of the pattern is to simplify a contract by reducing the number of conjuncts. This pattern could be accompanied by an inverse, but the inverse transformation (to add a conjunct *True*) may not be a useful refactoring step when designing Contractual State Machines.

Constraints: The pattern ensures that the inner contract is a *Conjunction* operator expression of the form $C_1 \wedge True$.

Parameters: The parameter is a state whose inner contract is a *Conjunction* operator expression.

Transformation: This pattern refines the contract $C_1 \wedge True$ to the contract C_1.

Name: *If combine actions*

Type: Refactoring

Rationale: The purpose of the pattern is to simplify the design process by reducing the number of conjuncts to be refined within a contract. The pattern merges two *If* operator conjuncts. The inverse of this pattern, (to split an *If* operator expression on its set of actions into the conjunction of two *If* operators) is presented in Appendix D.1. This inverse transformation however may not be a useful refactoring step for designing Contractual State Machines.

Constraints: The pattern ensures that the contract is a *Conjunction* operator expression of the form $C_1 \wedge C_2$ where C_1 and C_2 are *If* operator expressions. The inner contract has the form $[a_1, .., a_n[x : P_1]]L_1 \wedge [b_1, .., b_m[x : P_2]]L_2$. The pattern checks that the update expressions $[x : P_1]$ and $[x : P_2]$ and the continuation behaviours of the *If* expressions L_1 and L_2 are syntactically equal.

The pattern could be generalised to cover the situation where the update expressions $[x : P_1]$ and $[x : P_2]$ are not syntactically equal. A side-condition could verify that the update expressions are semantically equivalent $\vdash P_1 \Leftrightarrow P_2$. The side-condition could be generalised further to cover the situation where the continuation behaviours of the *If* expressions L_1 and L_2 are not syntactically equal. The side-condition check could be extended to verify that L_1 and L_2 are equivalent. Another generalisation is to conjoin L_1 and L_2.

Parameters: The parameter is a state whose inner contract is a *Conjunction* operator expression of the form $C_1 \wedge C_2$ where C_1 and C_2 are *If* operator expressions.

Transformation: This pattern combines two conjoined *If* operator expressions. The transformation replaces the original contract $[a_1, .., a_n[x : P_1]]L_1 \wedge [b_1, .., b_m[x : P_2]]L_2$ with a new contract $[a_1, .., a_n, b_1, .., b_m[x : P_1]]L_1$ that combines the action groups.

Diagram:

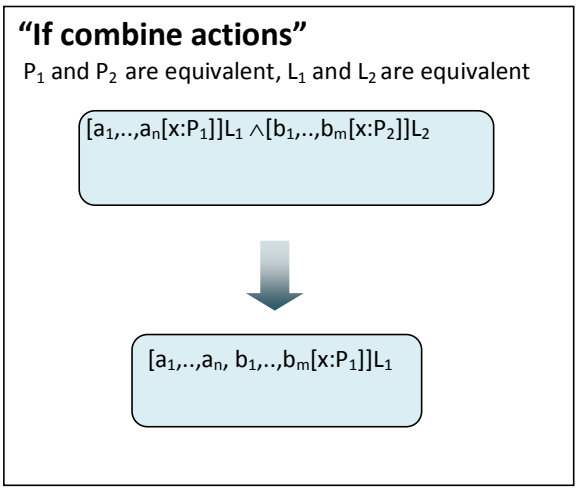

Figure 4.31: If combine actions

4.3.1.3 Patterns for unfolding a CoSta contract

The pattern theme in this section is *unfolding a contract expression.*

Name: *Unfold Always*

Type: Refactoring

Rationale: This pattern enables the engineer to provide more specific details about a design and the required behaviour of the intended process. It *unfolds* the *Always* operator expression to separate out the description of the behaviour for the current step and the ongoing subsequent behaviour of the process. This enables the engineer to be more precise about and elaborate further on the behaviour for the current step separately to considering the subsequent behaviour. This pattern was based on equivalences for the *Always* operator. This pattern could be accompanied by an inverse transformation.

Constraints: The pattern ensures that the inner contract is an *Always* operator expression.

Parameters: The parameter is a state with a contract that is an *Always* operator expression.

Transformation: The *Always* operator specifies a constraint which must be true at all times, whichever events are taken. The pattern *unfolds* an *Always* operator. *Always L* is equivalent to *L* and "*after any event, behave as Always L*".

The inner contract $\Box L$ is refined to $L \wedge [\_]\Box L$

Diagram:

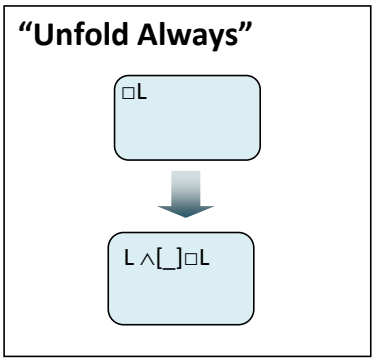

Figure 4.32: Unfold Always

Name: *Unfold Unless*

Type: Refinement.

Rationale: This pattern enables the engineer to elaborate a design and provide more details of the required behaviour. The pattern enables progress to be made with the design as it allows the engineer to make a choice about the behaviour of the process at the current step. It *unfolds* the *Unless* operator expression to separate out the description of the behaviour for the current step and the ongoing subsequent behaviour of the process. This enables the engineer to then elaborate further on these descriptions separately. This pattern was based on equivalences for the *Unless* operator.

Constraints: The pattern ensures that the inner contract is an *Unless* operator expression.

Parameters: The parameter is a state with a contract that is an *Unless* operator expression.

Transformation: The *Unless* operator behaves similarly to the *Always* operator. The left-hand formula must be true at all times. However should the right-hand formula become true, the left hand formula (at this point and thereafter) no longer need hold. The pattern *unfolds* an *Unless* operator. *L Unless M* is equivalent to (*L* and “ *after any event, behave as L Unless M*”) or “*behave as M*”.

1. The contract $L \curvearrowright M$ can be refined to $L \wedge [\_](L \curvearrowright M)$ (see Figure 4.33 diagram 1).

2. The contract $L \curvearrowright M$ can be refined to M (see Figure 4.33 diagram 2).

Diagram:

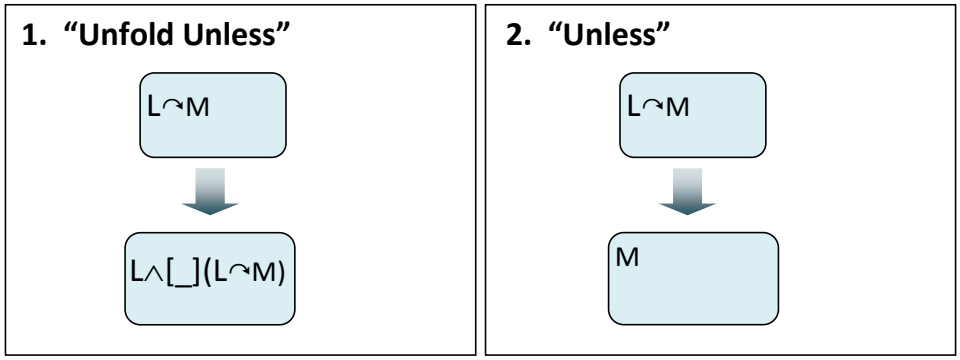

Figure 4.33: Unfold Unless

Name: *Unfold Within k* (BoundedEventually)

Type: Refinement

Rationale: This pattern enables progress to be made with the design as it allows the engineer to make a choice about the behaviour of the process at the current step. The choice is between satisfying the specified constraint at the current step or alternatively not satisfying the constraint at the current step but subsequently satisfying it within the next $k-1$ steps. This pattern was based on equivalences for the *Unfold Within k* operator.

Constraints: The pattern ensures that the inner contract is an *Unfold Within k* operator expression.

Parameters: The parameter is a state with a contract that is an *Unfold Within k* operator expression.

Transformation: The *Unfold Within k* operator specifies a constraint which must be true within k steps. The pattern *unfolds* an *Unfold Within k* operator.

Unfold within k L is equivalent to ("*Any event is enabled*" and "*after any event, behave as Unfold within k-1 L*") or "*behave as L*". If L does not hold, something else must be

enabled and the continuation behaviour specifies that within $(k-1)$, L must hold for every branch.

1. The contract $\Diamond_k L, k > 1$ can be refined to $\langle \_ \rangle \wedge [\_](\Diamond_{k-1} L)$ (see Fig 4.34 diagram 1).

2. The contract $\Diamond_k L, k \geq 1$ can be refined to L (see Fig 4.34 diagram 2).

Diagram:

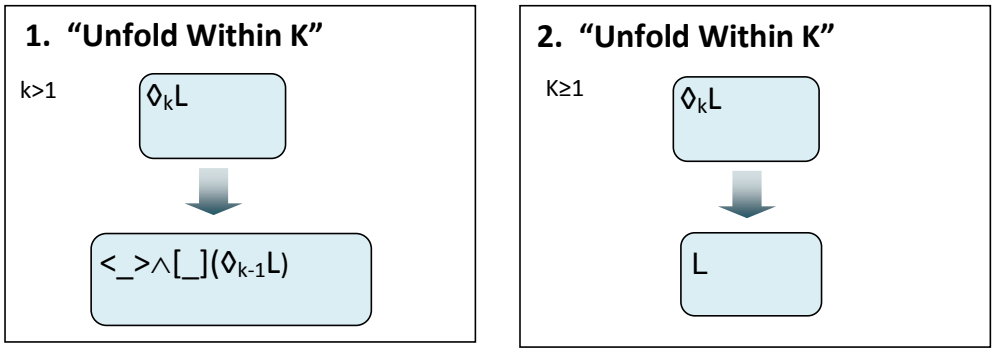

Figure 4.34: Unfold Within k

4.3.1.4 Patterns for rewriting a CoSta contract

The pattern theme in this section is *rewriting a contract expression in terms of an equivalent contract expression*. Each pattern in this section could be accompanied by its inverse transformation.

Name: *Disable*

Type: Refactoring

Rationale: The *Disable* operator specifies that none of the events in a set are available. The purpose of this pattern is to rewrite the *Disable* operator expression in terms of an *If* operator. The approach here is to express a *Disable* operator in terms of an *If* operator rather than introduce separate patterns for the *Disable* operator.

Constraints: The pattern ensures that the inner contract is a *Disable* operator expression.

Parameters: The parameter is a state with a contract that is a *Disable* operator expression.

Transformation: The pattern expresses the *Disable* operator contract as an equivalent *If* operator expression. The inner contract expression $\overline{\langle a_1, .., a_n[x : P] \rangle}$ is replaced by

$[a_1, .., a_n[x : P]]False$

Diagram:

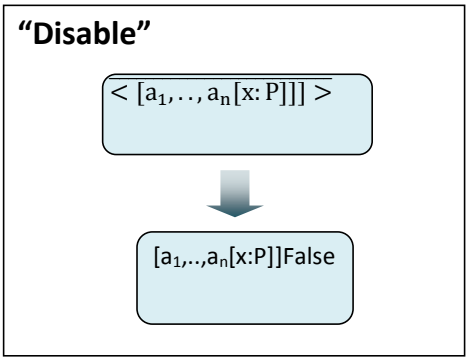

Figure 4.35: Disable

Name: *Commutativity*

Type: Refactoring

Rationale: Other patterns require contracts to be equivalent (e.g in order to eliminate a conjunction state or reroute). This pattern enables a contract to be expressed in an equivalent but different form by swapping conjuncts around.

Constraints: The pattern ensures the inner contract is an $\wedge$ operator expression.

Parameters: The parameter is a state with an inner contract of the form $C_1 \wedge C_2$.

Transformation: The inner contract $C_1 \wedge C_2$ is replaced by contract $C_2 \wedge C_1$.

Name: *Associativity*

Type: Refactoring

Rationale: Other patterns require contracts to be equivalent (e.g in order to eliminate a conjunction state or reroute). This pattern enables a contract to be expressed in an equivalent but different form by changing the bracketing of conjuncts.

Constraints: The pattern ensures the inner contract is an $\wedge$ operator expression of the form $(C_1 \wedge C_2) \wedge C_3$.

Parameters: The parameter is a state with an inner contract that is an $\wedge$ operator expression.

Transformation: The inner contract $(C_1 \wedge C_2) \wedge C_3$ is replaced by contract $C_1 \wedge (C_2 \wedge C_3)$.

4.3.2 Patterns for contracts to mixed designs

This section presents contract to mixed design refinement and refactoring patterns. For these patterns the source model of the transformation is a contract and the target model is a CoSta state machine with contracts.

4.3.2.1 Patterns for adding CoSta state machine constructs

The pattern theme in this section is *adding new constructs to a design.*

Name: *Conjunction introduction*

Type: Refactoring

Rationale: The pattern is required to support the approach to the design process and the introduction of conjunction states to refine contracts with conjuncts. As described above it was concluded that the best approach to refining these contracts would be to separate them into conjunction states and refine the conjunctions independently to two mutually consistent (usually identical) designs. The inverse transformation for this pattern (*Conjunction elimination*) is presented below in Section 4.3.3.1.

Constraints: The pattern checks that the inner contract is a *Conjunction* operator expression.

Parameters: The parameter is a state with an inner contract that is a *Conjunction* operator expression.

Transformation: The contract has the form "$L \wedge M$". The pattern introduces two new substates, one with inner contract "L" and the other with inner contract "M". The new substates are *conjunction* states. As described previously they are an addition to the state machine language and a new form of state decomposition required to support the approach to the design process. Conjoined substates are orthogonal in the same way as parallel states. Conjunction states represent the behaviour *common* to all their substates.

Diagram:

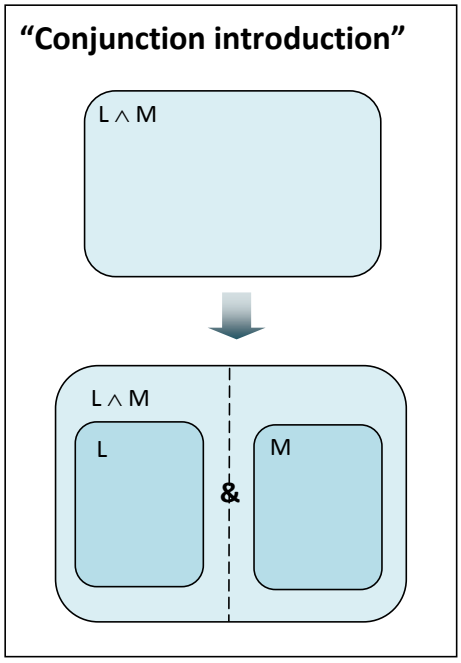

Figure 4.36: Conjunction introduction

Name: *Expand*

Type: Refactoring

Rationale: This pattern constructs the maximal agent for a contract, i.e. a state machine which represents the disjunction (nondeterministic choice) between all the state machines that satisfy the contract. The rationale behind the more general patterns such as this one is that they are available in a supporting role, should a more specific patterns not cover what is required.

The semantic theory of duals, which transforms contracts to LTS/STGA is one-way. Investigating an inverse would be an important theoretical issue, as would investigating the completeness of the inverse. For now, the only possible inverse one could supply is based on a complex side condition and involves additional effort from the user/engineer. For a state machine describing the behaviour of (and encapsulated by) a non-leaf state,

one could ask the user/engineer to supply a contract. If the model-checker confirms the contract is semantically equivalent to the state machine then the state machine could be replaced by an inner contract on the state whose behaviour it defines.

Constraints: This pattern is applicable to a state with an inner contract.

Parameters: The parameter is a state with an inner contract.

Transformation: The pattern introduces a state machine model for an inner contract. It recursively looks at each operator in the contract, expanding its operands and combining the resulting state machines using a construction corresponding to the operator. As contracts are potentially succinct ways of describing highly nondeterministic state machines, applying the *Expand* pattern may produce very complex state machines. Note that the functionality required to support this pattern is not currently supported by the model checker.

4.3.3 Patterns for mixed designs

This section presents refinement and refactoring patterns for mixed designs. These patterns transform a CoSta state machine design (with contracts) into another CoSta state machine design (with contracts). They are important for several reasons, for example, some of them are essential such as the *Reroute* pattern that introduces cycles into the design (and thus satisfies *Always* contracts); and some are important for convenience as a resulting state machine model may not be what is required and rather than the engineer having to backtrack and refine again from a more abstract design, patterns can be applied directly to the Contractual State Machine model to achieve the desired result.

4.3.3.1 Patterns for removing CoSta state machine constructs

The pattern theme in this section is *removing CoSta constructs from a design.*

Name: *Remove a composite superstate/Redundant hierarchy*

Type: Refactoring

Rationale: The pattern eliminates a redundant level of hierarchy. During the design process components are refined independently. Initially behaviour of a component is summarised by a state with a contract. The engineer may then design a state machine model for the behaviour within the state and this may result subsequently in the containing state no longer being required. This pattern provides a means of removing it. The inverse of this pattern (*Create a composite superstate*) is presented below in Section 4.3.3.2.

Constraints: The pattern ensures it is applied to a composite state with substates and the composite state is not the source or target of transitions.

Parameters: The parameter is a composite state.

Transformation: The pattern removes a composite state but retains its substates. If the composite state is itself a substate and a start state, the inner contract of its parent state is assigned to the outer contract of its start state. This transformation is the final step in the sequence of transformations to *flatten hierarchy* which is common in the research literature for refactoring of state machine designs. In figure 4.37 the dotted transition and cloud convey that only the root state is affected by the pattern, there may be further structure reachable from the root state, but it is unaffected by the transformation. The important thing to note about this pattern is that the root state may optionally inherit the inner contract of its new encapsulating state as an outer contract. These are used in rerouting transitions e.g. to introduce a cycle.

Diagram:

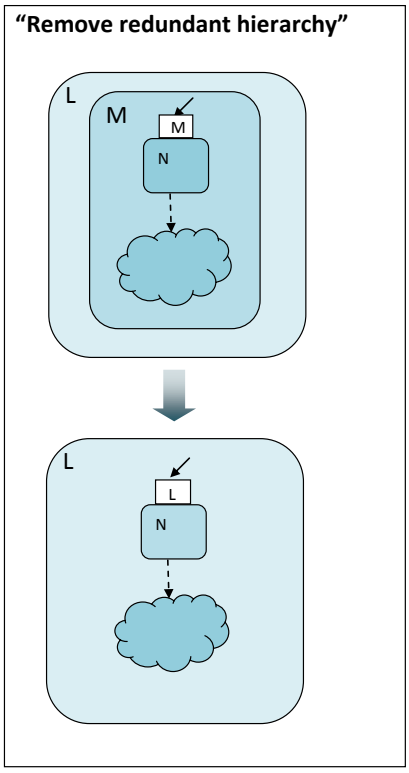

Figure 4.37: Remove a composite superstate

Name: *Combine transitions*

Type: Refactoring

Rationale: This pattern combines two transitions. The purpose of the pattern is to simplify a state machine design. The inverse of this pattern (*Split transition*) is presented in Chapter 5, section 5.2.3.3.

Constraints: The pattern is applicable to two transitions $a[x : P_1]$ and $a[x : P_2]$ between the same source and target states with the same events. The pattern replaces the transitions with a single transition whose update expression is formed from the disjunction of the update expressions of the original transitions.

Parameters: The parameters are two transitions.

Transformation: The pattern combines two transitions $a[x : P_1]$ and $a[x : P_2]$ into a single transition $a[x : P_1 \vee P_2]$ between the same source and target states.

Diagram:

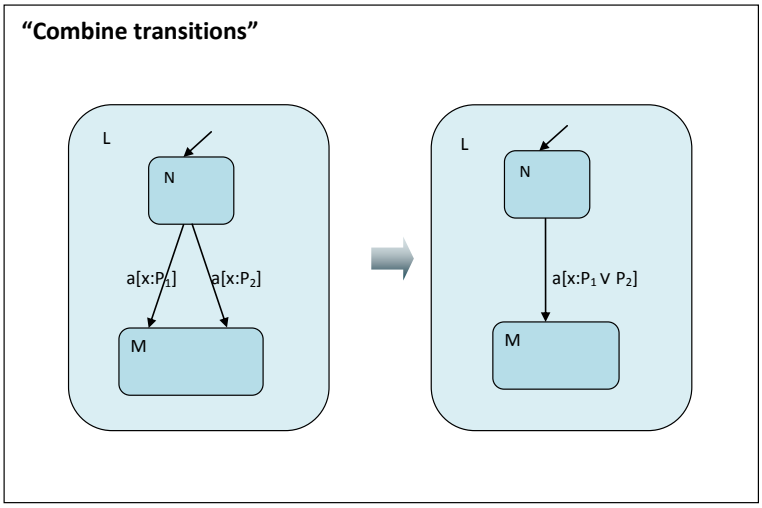

Figure 4.38: Combine transitions

Name: *Combine states*

Type: Refactoring (a refinement in its most general form).

Rationale: This pattern combines two basic states. Combining two states having the same inner contracts which the engineer wishes to refine into the same behaviour will

simplify the design and reduce the number of design steps if the states are combined first before their behaviour is specified in more detail. This pattern was identified when considering ways of reducing the number of design steps during the design process. The inverse of this pattern, *Split state*, is presented in Appendix D.1.

Constraints: The pattern is applicable to two states with inner contracts that have not yet been refined into CoSta state machines. The states to be combined must belong to the same parent state. Additionally the side-condition verifies that their inner contracts are syntactically equivalent. The pattern could be generalised to cover the situation where the contracts are syntactically the same apart from the update expressions. A side-condition could verify that all the corresponding update expressions from both contracts are semantically equivalent. To check if two update expressions, $[x : P_1]$ and $[x : P_2]$ are equivalent, the side-condition is $\vdash P_1 \Leftrightarrow P_2$. The side-condition could be generalised further to check if a refinement relation holds between the two contracts of the states to be combined and the resulting state is assigned the stronger contract.

Parameters: The parameters are two states with unelaborated inner contracts, (not yet refined into CoSta state machines).

Transformation: The two states to be combined are replaced by a new state with incoming transitions consisting of the set of all the incoming transitions to both original states. Either of the original contracts can be selected as the contract for the new combined state.

Diagram:

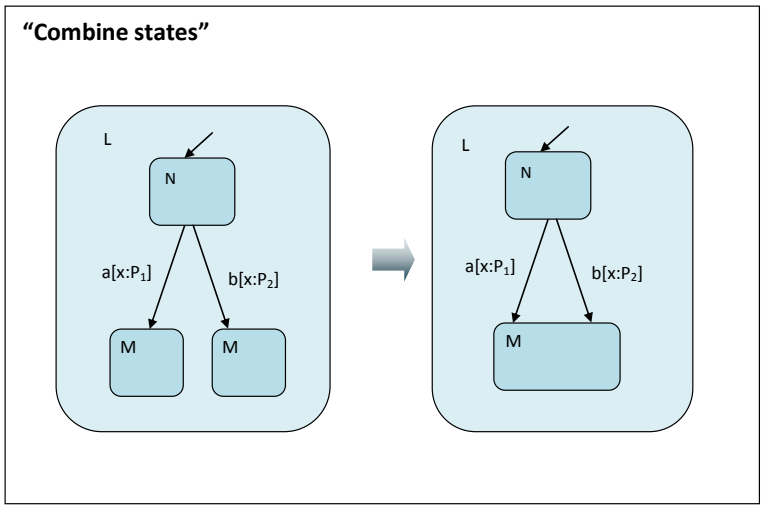

Figure 4.39: Combine states

Name: *Remove an unreachable state*

Type: Refactoring

Rationale: The purpose of the pattern is to simplify a design and remove redundant states. This pattern could be accompanied by an inverse transformation, to add an unreachable state, but it would not be useful during the design process as the state would remain unreachable because it is not a permitted refinement under SVRS to add further connecting transitions to the new state.

Constraints: The test for unreachability is trivially satisfied in this pattern which is applicable to a state that has no incoming transitions to itself or any of its substates from states other than the selected state or any of its substates. Additionally the selected state must not be a start state.

Parameters: The parameter is a state.

Transformation: The transformation removes an unreachable state and all of its outgoing transitions. If it is a composite state its substates and their transitions are removed as well.

Name: *Conjunction elimination/Equal*

Type: Refactoring

Rationale: This pattern is applicable to a conjunction of states and its purpose is to eliminate the conjunction. As described above, this pattern is required to support the approach to the design process based on the separate refinement of conjunction states. It is applied to remove the conjunction operator and syntactically equivalent conjuncts. The inverse of this pattern, (*Conjunction introduction*) is presented above in Section 4.3.2.1.

Constraints: This pattern is applicable to a conjunction of states that are consistent with one another. The processes within every conjunction state must be syntactically the same. The diagrams must have the same structure (state connectivity, transition triggers).

This pattern is based on syntactic equivalence. It could be generalised to cover the situation where the diagrams have the same structure and the transition events are syntactically the same but not the update expressions. A side-condition could verify that the update expressions $[x : P_1]$ and $[x : P_2]$ from the transitions are semantically equivalent. The side-condition is $\vdash P_1 \Leftrightarrow P_2$. There are other ways to check for equivalence and the side-condition could be generalised further to a full SVRS-equivalence check between the conjuncts.

Parameters: The parameter is a composite conjunction state containing syntactically equivalent substates.

Transformation: The pattern replaces the composite conjunction state containing the equivalent substates with one of the conjuncts and the copies are thrown away.

Diagram:

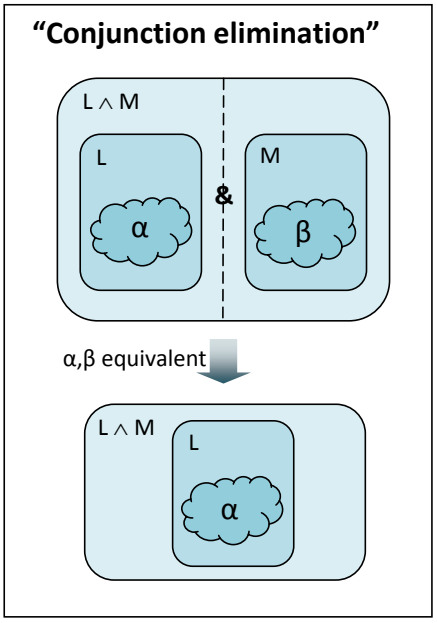

Figure 4.40: Equal

Name: *Conjunction*

Type: Refinement

Rationale: This is a potentially a more general pattern than the *Conjunction Elim/Equal* pattern that is available should it not be possible for the design to be progressed further with the specific pattern that relies on syntactic equivalence.

Constraints: This pattern is applicable to a composite conjunction state and constructs the behavioural intersection of its conjuncts. The pattern is a generalisation of the *Conjunction Elim/Equal* pattern which instead does not rely on the conjuncts being syntactically equivalent. Initially this pattern checks for a refinement relation between the conjuncts and if so selects the strongest conjunct. If a refinement relation does not hold

the conjuncts are refined into nondeterministic Contractual State Machines and their behavioural intersection computed. Note that the functionality required to support this pattern is not currently supported by the model checker.

Parameters: The parameter is a composite conjunction state.

Transformation: The pattern replaces the composite conjunction state with the behavioural intersection of its conjuncts.

Name: *Reroute*

Type: Refactoring (in its most general form it is a refinement).

Rationale: This pattern is important as it permits the introduction of a looping transition into a state machine design by rerouting a state's incoming transitions to a new target state, whose behaviour is a refinement/refactoring of the old target and has already been expanded (specified in terms of a state machine design). The refactoring version of this pattern could be accompanied by an inverse transformation but it may not be useful as a refactoring step when designing Contractual State Machines.

Constraints: This pattern is applicable to two states, one with an inner contract that has not yet been refined to a state machine and another (the new target state for the looping transition) with an associated outer contract that refines the inner contract. The pattern ensures that the unelaborated inner contract and the associated outer contract are syntactically equivalent. The pattern could be generalised to cover the situation where the contracts are syntactically the same apart from the update expressions. In this instance a side-condition could verify that the corresponding update expressions from both contracts are semantically equivalent. To determine the semantic equivalence of two update expressions $[x : P_1]$ and $[x : P_2]$ the side-condition is $\vdash P_1 \Leftrightarrow P_2$. The side-condition could be generalised further to check if a refinement relation holds between the unelaborated inner contract and the associated outer contract.

Parameters: The parameters are two states.

Transformation: The transformation removes the unelaborated state and its incoming transitions are redirected to the state with the associated outer contract.

Diagram:

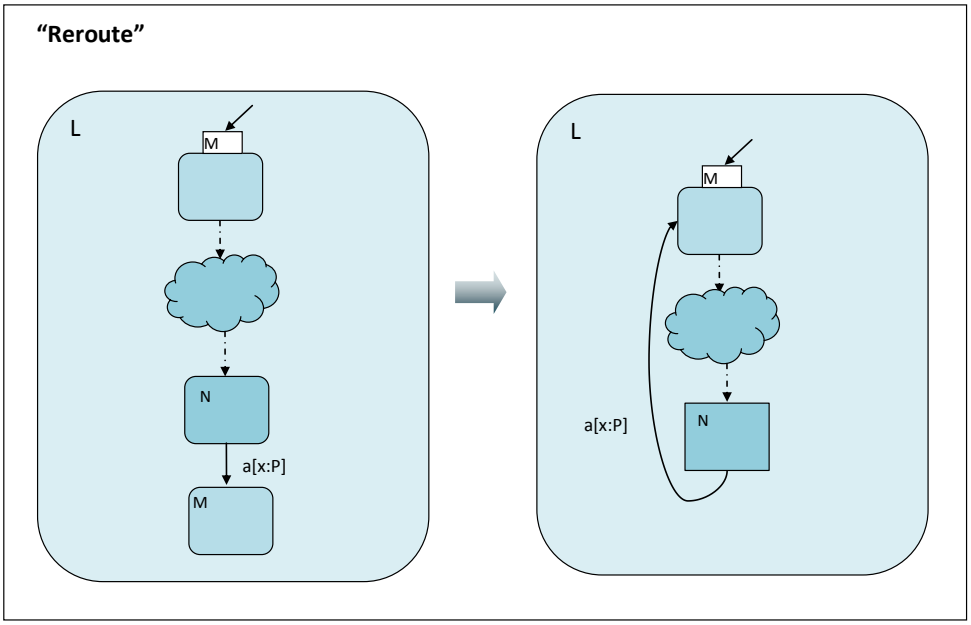

Figure 4.41: Reroute

4.3.3.2 Patterns for adding CoSta state machine constructs

The pattern theme in this section is *adding CoSta design constructs.*

Name: *Create a composite superstate*

Type: Refactoring

Rationale: This transformation is the first step in the sequence of transformations to introduce hierarchy which is common in the research literature for refactoring state machine designs. The inverse of this pattern (*Remove a composite superstate/Redundant hierarchy*) is presented above in Section 4.3.3.1.

Constraints: The pattern is applicable to a set of states that belong to the same parent state.

Parameters: The parameters are a set of states.

Transformation: The pattern introduces a composite state having the selected set of states as substates.

4.3.3.3 Patterns for modifying a CoSta state machine design

The pattern theme in this section is *modifying a CoSta model*.

Name: *Move target down*

Type: Refactoring

Rationale: This transformation is one of a sequence of transformations to flatten hierarchy which is common in the research literature for refactoring state machine designs. The inverse of this pattern (*Move target up*) is presented below in this section.

Constraints: The pattern is applicable to a composite state with incoming transitions.

Parameters: The parameter is a composite state with substates.

Transformation: The transformation shifts the target state of the incoming transitions to the composite state down from the composite state to the default/initial substate of the composite.

Diagram:

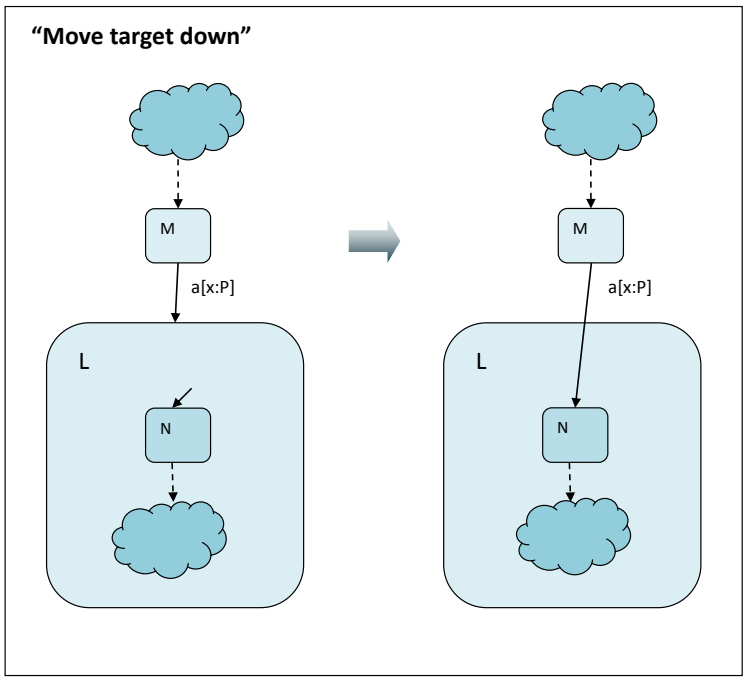

Figure 4.42: Move target down

Name: *Move target up*

Type: Refactoring

Rationale: This transformation is one of a sequence of transformations to introduce hierarchy which is common in the research literature for refactoring state machine designs. The inverse of this pattern (*Move target down*) is presented above in this section.

Constraints: The pattern is applicable to a composite state with incoming inter-level transitions whose target state is the start state within a composite state and whose source state is not a substate of the composite state.

Parameters: The parameter is a composite state with substates.

Transformation: The incoming transitions to the default/initial state of a composite state have their target state shifted up to the parent composite state.

Diagram:

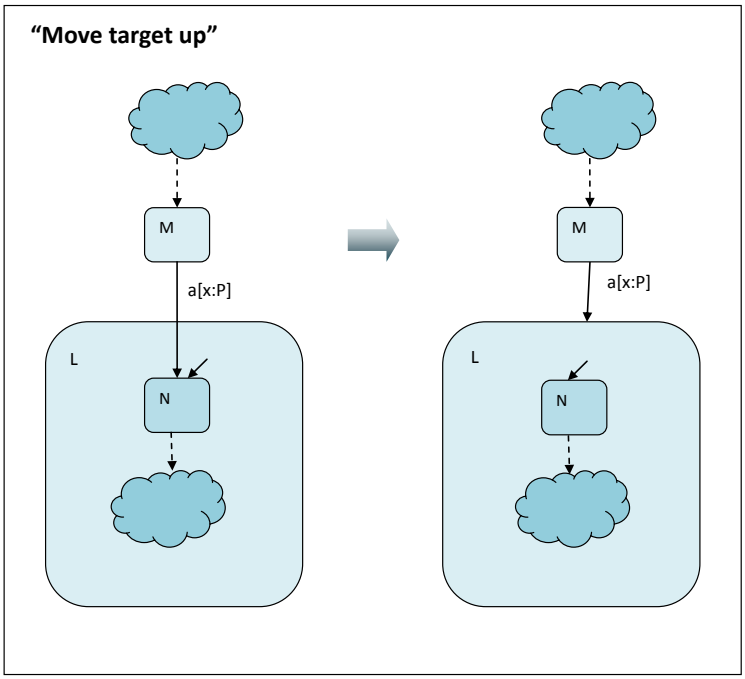

Figure 4.43: Move target up

Name: *Move source down*

Type: Refactoring

Rationale: This transformation is one of a sequence of transformations to flatten hierarchy which is common in the research literature for refactoring state machine designs. The inverse of this transformation (*Move source up*) is presented below in this section.

Constraints: The pattern is applicable to a composite state with outgoing transitions.

Parameters: The parameter is a composite state with substates.

Transformation: The transformation replaces each outgoing transition from the composite state with a set of identical outgoing transitions one from each of the substates of the composite.

Diagram:

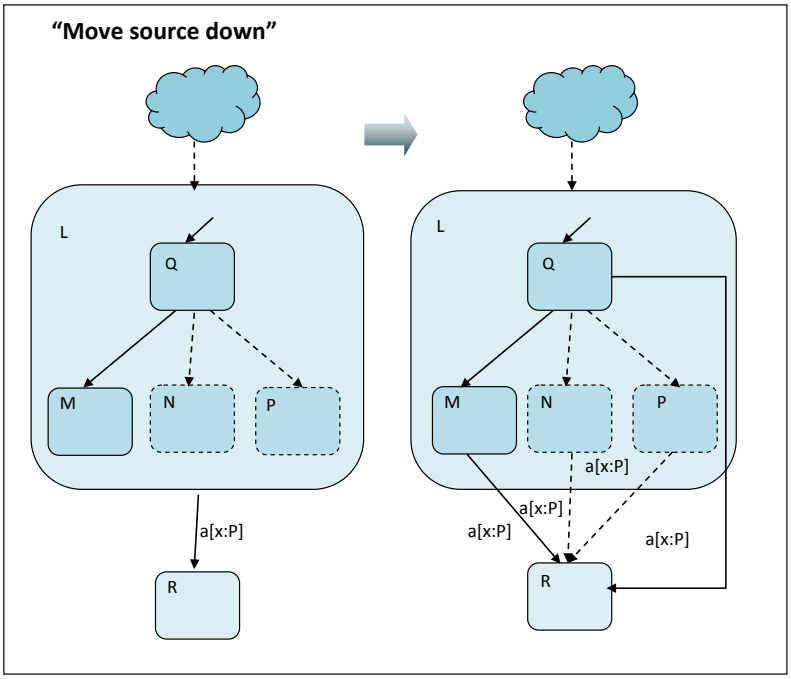

Figure 4.44: Move source down

Name: *Move source up*

Type: Refactoring

Rationale: This transformation is one of a sequence of transformations to introduce hierarchy which is common in the research literature for refactoring state machine designs. The inverse of this transformation (*Move source down*) is presented above in this section.

Constraints: The pattern is applicable to a composite state where each of its substates has an identical outgoing inter-level transition. The set of identical transitions, one for each substate of the composite have the same events, guards, assignments and target states where the target state is outside the composite state. The pattern checks that the transitions have the same events and syntactically equivalent update expressions. The pattern could be generalised to cover the situation where the events are the same but the update expressions are not syntactically equivalent but may be semantically equivalent. In this instance the side-condition verifies that the update expressions $[x : P_1]$ and $[x : P_2]$ are semantically equivalent, the side-condition is $\vdash P_1 \Leftrightarrow P_2$.

Parameters: The parameter is a composite state with substates.

Transformation: The set of identical outgoing transitions one from each substate of the composite are replaced by a single outgoing transition leaving the composite with the same events, update expressions and target state.

Diagram:

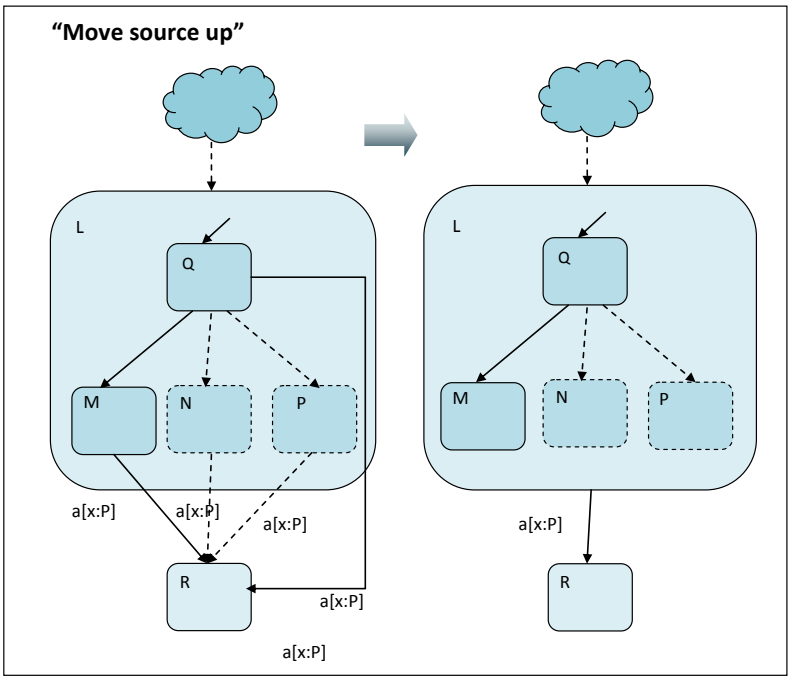

Figure 4.45: Move source up

4.4 Conclusions

The simpler types of refinement and refactoring patterns that in their most specific form are predominantly structural in nature and which do not refine data have been presented in this chapter. The next chapter presents the more elaborate refinement and refactoring patterns for transformations based on semantic side-conditions that require a model checker (or theorem prover) to discharge them.

Chapter 5

Further refinement and refactoring patterns

5.1 Introduction

Chapter 4 presented the refinement and refactoring patterns that are predominantly structural in nature with no semantic side-conditions. This chapter presents the more general refinement and refactoring patterns that focus on transformations with side-conditions (e.g relating to conditions/actions on data) which in the implementation (discussed in Chapter 6) will require a model-checker to discharge them.

5.1.1 Context

As described in Section 2.13.5, the refinement of Contractual State Machines with shared variables may make behaviour more deterministic but must ensure that enabled actions (ready sets) do not change for each data state. Refinement ensures the permissible enabled actions (ready sets) are consistent as well as the behaviours each action in these ready sets leads to. This requires specific and careful attention to how guards and variable values change in a refinement step.

Contracts are usually highly non-deterministic, so the permissible ready-sets are implicitly defined. However at the state machine level, within a shared variable environment, for each data state the enabled actions are explicit and must be maintained subject to the reduction of non-determinism. Changes to data updates must remain consistent (the update still lies within that of the contract) as it is impossible to know what effect the change may have on other processes in the environment by changing the variables in a

new way.

A pattern expects an update expression to be in the form $[x : P]$. The Contractual State Machine language however supports other forms of update expressions in contracts and transitions, therefore the contract update expressions are converted before pattern application. The following describes how update expressions are converted from the concrete forms (e.g. $[G ==> x := e]$) to the $[x : P]$ form. The $[x : P]$ form is not considered part of the syntax for CoSta State Machine designs, therefore there are corresponding transformations (not included here) from the $[x : P]$ form to the concrete forms.
x and x' generalise to vectors of variables and e to a vector of expressions:

1. No update expression specified (allows all variables to change arbitrarily) becomes $[x : true]$.
2. An update expression $[x = e]$ becomes $[x : x' = e]$.
3. An update expression $[skip]$ becomes $[x : x' = x]$.
4. An update expression $[P \Rightarrow x = e]$ becomes $[x : P \wedge x' = e]$.
5. An update expression $[P \Rightarrow skip]$ becomes $[x : P \wedge x' = x]$.

Similarly the transition update expressions are converted before pattern application as follows, where x and x' generalise to vectors of variables and e to a vector of expressions:

1. No update expression specified (no variables allowed to change) becomes $[x : x' = x]$.
2. An update expression $[x = e]$ becomes $[x : x' = e]$.
3. An update expression $[skip]$ becomes $[x : x' = x]$.
4. An update expression $[P]$ becomes $[x : P \wedge x' = x]$. P does not contain any primed variables.
5. An update expression $[P \Rightarrow x = e]$ becomes $[x : P \wedge x' = e]$.
6. An update expression $[P \Rightarrow skip]$ becomes $[x : P \wedge x' = x]$.

5.2 Further patterns

The following sections present the more general refinement and refactoring patterns that focus on transformations with side-conditions. The patterns minimise proof burden as the

side-condition is a simpler proof than a full refinement check on the source and target models. The patterns are presented following the same template that was used in Chapter 4.

5.2.1 Contract to contract patterns

This section presents the contract to contract refinement and refactoring patterns where both the source and target models of the transformation are contracts.

5.2.1.1 Patterns for removing parts of a CoSta contract

The pattern theme in this section is *removing parts of a contract expression.*

Name: *Frame contraction*

Type: Refactoring

Rationale: The purpose of the pattern is to enable the engineer to be more succinct about permitted updates to variables within a contract. If a variable *skips* this can be explicitly specified by removing the variable from the frame. This pattern could be accompanied by an inverse transformation.

Constraints: This pattern is applicable to an update expression within a contract. The frame for the update expression $[x_1, x_2, ..x_n : P]$ can be contracted by removing $x_i, 1 \leq i \leq n$. The side-condition checks that all the updates consistent with P do not change the value of the variable to be excluded, $\vdash P \Rightarrow x_i' = x_i$.

Parameters: The parameters are an update expression from a contract and the variable to be excluded from the frame.

Transformation: The variable to be excluded from the frame is removed from the update expression.

Diagram:

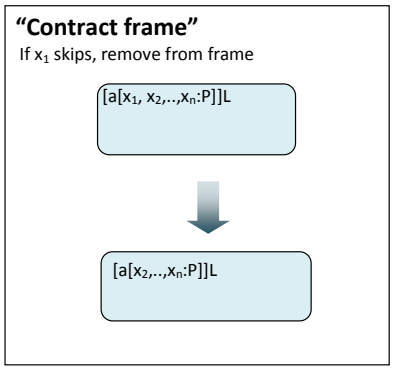

Figure 5.1: Contract frame

5.2.1.2 Patterns for modifying a CoSta contract

The pattern theme in this section is *modifying parts of a contract expression.*

Name: *TotEnable refine update*

Type: Refinement

Rationale: The purpose of the pattern is to allow the update within a *Totalised Enable* operator expression to be refined. The original update expression is $[x : P]$ and $[x : P']$ is the refinement. $\langle[a[x : P]]\rangle$ expresses that a is enabled from everywhere within the (implicit) guard of P, with an update consistent with P (the update still lies within that of P). The guard can be weakened and the update can be strengthened within the guard making it no less deterministic.

Constraints: This pattern is applicable to a *Totalised Enable* operator expression within a contract. For the *Totalised Enable* operator the guard cannot be strengthened or the update weakened as this would weaken the contract and not preserve contractual properties for the *Totalised Enable* operator. Weakening the update may lead to undesired effects on other processes in the shared variable environment.

The pattern permits the following refinements.

(i) The guard can be weakened, introducing new states which the event can be enabled from. Thus enabling the actions in more situations than is required (but keeping the behaviour within the old guard the same).

The side-condition is $((\exists x'.P) \Rightarrow (\exists x'.P')) \wedge ((\exists x'.P) \wedge P' \Leftrightarrow P)$

(ii) P can be strengthened keeping the guard the same, thus strengthening the update

within the guard making it no less deterministic.

The side-condition is $\vdash ((\exists x'.P) \Leftrightarrow (\exists x'.P')) \wedge ((\exists x'.P) \wedge P' \Rightarrow P)$.

The pattern captures both side-conditions within a single rule. The side-condition to be verified is $\vdash ((\exists x'.P) \Rightarrow (\exists x'.P')) \wedge ((\exists x'.P) \wedge P' \Rightarrow P)$

Parameters: The parameters are a *Totalised Enable* operator expression within a contract and the new update expression that is a refinement of the original.

Transformation: The original update expression within the *Totalised Enable* operator expression is replaced by its refinement.

Diagram:

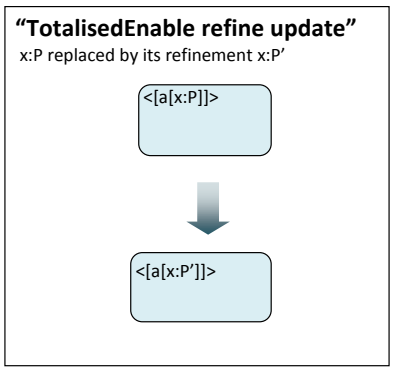

Figure 5.2: TotEnable refine update

Name: *If refine update*

Type: Refinement

Rationale: The purpose of the pattern is to refine the update within an *If* operator expression. $[a[x : P]]L$ expresses that an a action with an update consistent with $x : P$ (i.e from the domain of P and with an effect on the variables permitted by P) must behave like L.

Constraints: This pattern is applicable to an *If* operator expression $[a[x : P]]L$ within a contract. It is not permitted to strengthen P as this weakens the contract and does not preserve contractual properties. P can be weakened to P'. The side-condition is $\vdash P \Rightarrow P'$

Parameters: The parameters are an *If* operator expression within a contract and the new update expression that is a refinement of the original.

Transformation: The transformation replaces the original *If* operator update expression $[x : P]$ with the new one $[x : P']$.

Diagram:

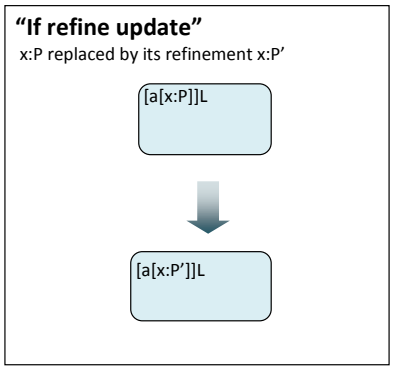

Figure 5.3: If refine update

5.2.2 Patterns for contracts to mixed designs

This section presents refinement and refactoring patterns for contracts to mixed designs (where a mixed design incorporates state machines and contracts). A contract to a mixed design pattern is one where the source model of the transformation is a contract and the target model is a CoSta state machine with contracts.

5.2.2.1 Patterns for adding CoSta state machine constructs

The pattern theme in this section is *adding CoSta design constructs.*

Name: *If*

Type: Refinement

Rationale: This pattern enables the engineer to elaborate on the required operational behaviour for an *If* operator contract. *If* contracts are expanded into designs with mixed content. The *If* operator specifies constraints on the behaviour following a particular set of events (for all events in the set). When accompanied by a variable update, the constraints need only apply from a before state satisfying the guard of the assignment, and following a consistent update. $[a[x : P]]L$ expresses that an a action with an update consistent with $x : P$ must behave like L.

Constraints: The pattern is applicable to a state whose inner contract is an *If* operator expression. The pattern refines the contract $[a[x : P]]L$ to a state machine model consisting of a start state with outgoing transitions (specified as parameters) that each target a new

state with either the inner contract L (if the transition's update lies within that of the contract) or inner contract *True* otherwise. The action a in the contract $[a[x : P]]L$ can be generalised to a set of actions, a negated set of actions or the underscore.

If no update is specified in the contract, transitions with any update from any source data state have a consistent update and must behave like L if the event is in the eventlist. If an update expression $[x : P]$ is specified in the contract, transitions with an update $[x : P']$ consistent with and from a source state satisfying $[x : P]$ (lying within that of the contract) must also behave according to L if the event is in the eventlist.

If the contract specifies a list of events, transitions with some event from the set must behave like L if the updates are consistent or *True* otherwise. If the contract specifies a negated list of events, transitions with some event not in the set must behave like L if the updates are consistent or *True* otherwise. If the contract specifies an underscore for the event, transitions with any event must behave like L if the updates are consistent or *True* otherwise.

The pattern permits a transition to be added with any action and update behaving subsequently as L. This is because a transition which behaves consistently with the contract $[a[x : P]]L$ has to behave like L but a transition that does not behave consistently with the contract can behave in any way as the continuation behaviour is *True* which can be refined to anything including L.

A transition can be added with a different action to those specified in the contract or an update P' with a side-condition $\vdash P' \Rightarrow \neg P$, behaving subsequently as *True*. When introducing transitions that do not behave like L, (with subsequent behaviour *True*) it is necessary to ensure that assuming the event is the same their updates are not consistent with the contract and there must be no overlap. A permitted refinement is to weaken the update expression. If the update expression for the contract is $[x : P]$ and the update expression for the transition is $[x : P']$ the proof obligation $\vdash P \Rightarrow P'$ must hold.

Parameters: The parameters are a state with a contract that is an *If* operator expression. Additionally, this pattern requires the set of new transitions (possibly empty) that are to be introduced by the user.

Transformation: The pattern refines the contract $[a[x : P]]L$ to a state machine model consisting of a start state with outgoing transitions (supplied as parameters) that each target a new state with an inner contract of either L (where L is the constraint associated with the *If* operator) or *True*. When there are no transitions specified *If* is trivially refined

to deadlock (a state with no outgoing transitions).

Diagram:

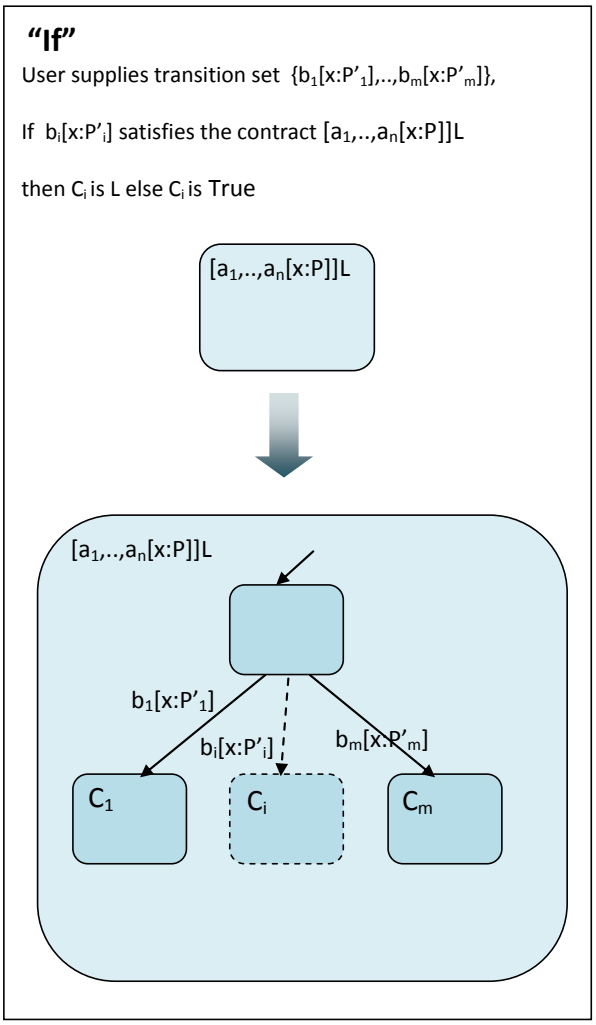

Figure 5.4: If

Name: *TotEnable*

Type: Refinement

Rationale: This pattern enables the engineer to elaborate on the required operational behaviour for a *TotEnable* operator contract. *TotEnable* contracts are expanded into designs with mixed content. The *TotEnable* operator specifies that one or more events in a set must be available. When accompanied by a variable update the event is enabled from

every before-state within the guard of the assignment and must be capable of a consistent update. For $\langle [a[x : P]] \rangle$, a is enabled over the entire guard of P, but only for some change in variables allowed by P.

Constraints: The pattern is applicable to a state whose inner contract is a *TotEnable* operator expression. The set of new transitions (outgoing transitions from the start state) are provided as parameters to the pattern. If the contract specifies a list of events, transitions with some event from the set are possible. If the contract specifies a negated list of events, transitions with some event not in the set are possible. If the contract has an underscore for the event list, transitions with any event are possible.

The set of events and update expressions for the new transitions and the set of enabled events and update expressions in the contract must not be disjoint. The side-condition checks that there is at least one instance where the transition parameters satisfy the *TotEnable* contract. Permitted refinements to the update expression are to weaken the guard or strengthen the update within the guard. To determine whether there is at least one instance where the transition parameters satisfy the *TotEnable* contract, all transitions for the same event are considered together and treated as a single transition with an update expression that is the disjunction of their separate update expressions. This makes it possible to split a transition on its guard or update with this pattern.

If no update expression is specified in the contract, new transitions with some update from some source data state satisfy the contract. If an update expression $[x : P]$ is specified in the contract, transitions with an update $[x : P']$ consistent with and from a source state satisfying $[x : P]$ satisfy the contract. To ensure that the update expressions are consistent the following side-condition is required $\vdash ((\exists x'.P) \Rightarrow (\exists x'.P')) \wedge ((\exists x'.P) \wedge P' \Rightarrow P)$.

Parameters: The parameters are a state with a contract that is a *TotEnable* operator expression. Additionally, this pattern requires the set of new transitions.

Transformation: The pattern refines the *TotEnable* contract to a state machine model consisting of a start state with outgoing transitions (supplied as parameters) that each target a new state with the inner contract *True*.

Diagram:

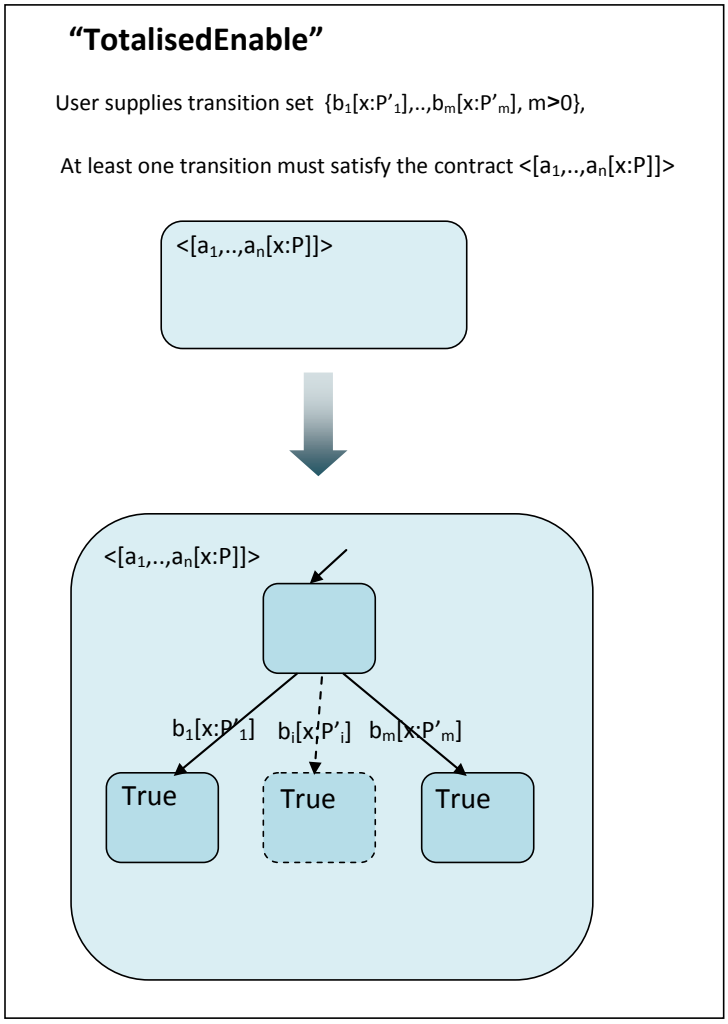

Figure 5.5: Totalised Enable

5.2.3 Patterns for mixed designs

This section presents refinement and refactoring patterns where both the source and target models are mixed designs, where a mixed design incorporates state machines with contracts.

5.2.3.1 Patterns for removing CoSta state machine constructs

The pattern theme in this section is *removing CoSta design constructs.*

Name: *Frame contraction*

Type: Refactoring

Rationale: The purpose of the pattern is to enable the engineer to be more succinct about permitted updates to variables for a transition. If a variable *skips* this can be explicitly specified by excluding the variable from the frame. This pattern could be accompanied by an inverse transformation.

Constraints: This pattern is applicable to a transition and supports frame contraction within the update expression. The frame can be contracted for the update expression $[x_1, x_2, ..x_n : P]$, by removing $x_i, 1 \leq i \leq n$ and the side-condition checks that all the updates consistent with P do not change the value of the variable to be excluded.
$\vdash P \Rightarrow x_i' = x_i$.

Parameters: The parameters are a transition and the variable to be excluded from the frame.

Transformation: The variable to be excluded from the frame is removed from the transition's update expression.

Diagram:

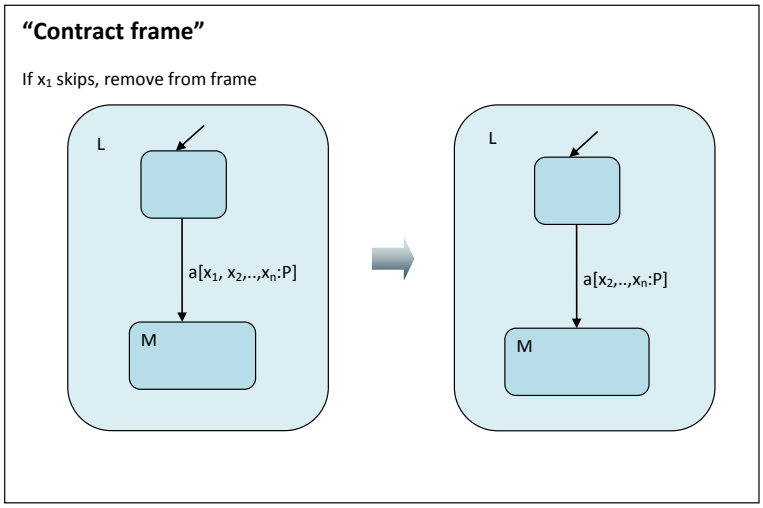

Figure 5.6: Contract frame

Name: *Remove transition with a False guard*

Type: Refactoring

Rationale: The purpose of the pattern is to remove a transition that can never trigger and simplify the design. This pattern could be accompanied by an inverse transformation

but it would not be a useful refactoring step when designing Contractual State Machines.

Constraints: This pattern is applicable to a transition $a[x : P]$ where the guard of the transition is semantically equivalent to *False*. The side-condition verifies $\vdash \neg P$.

Parameters: The parameter is a transition.

Transformation: The pattern removes the transition.

Diagram:

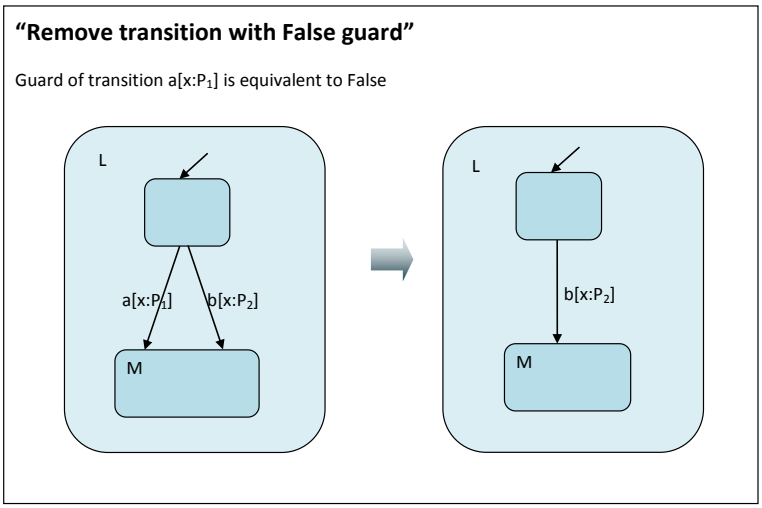

Figure 5.7: Remove transition with a False guard

5.2.3.2 Patterns for modifying a CoSta state machine design

The pattern theme in this section is *modifying CoSta design constructs*.

Name: *Transition refine update*

Type: Refinement

Rationale: The purpose of the pattern is to specify behaviour for a transition more precisely.

Constraints: This pattern is applicable to a transition $a[x : P]$ and supports refinement of the update expression. The update expression can be refined to a stronger expression but the guard must remain the same. P can be strengthened within the guard to P'.
The side-condition is $\vdash ((\exists x'.P) \Leftrightarrow (\exists x'.P')) \wedge ((\exists x'.P) \wedge P' \Rightarrow P)$.

Parameters: The parameters are a transition and a new update expression for the transition that is a refinement of the original.

Transformation: The pattern replaces the original update expression with the new one.

Diagram:

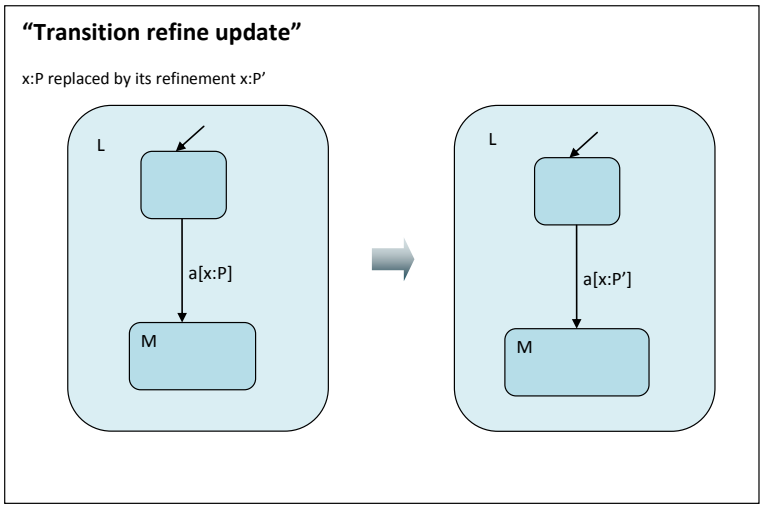

Figure 5.8: Transition refine update

Name: *Strengthen guard*

Type: Refinement

Rationale: The pattern strengthens the guard of a transition. Strengthening or weakening the guard of a transition may change the enabled actions. It is a permitted refinement however to strengthen a guard (to remove/reduce nondeterminism) within an overlap between guards for transitions from the same source state with the same actions.

Constraints: This pattern is applicable to two transitions $a[x : P_1]$ and $a[x : P_2]$ with the same events from the same source state that can have different target states and updates. The guards of the transitions must not be disjoint. The pattern permits the strengthening of the guard of P_1 to P_1' within the overlap of the guards of P_1 and P_2. The side-condition is $\vdash (P_1' \Rightarrow P_1) \wedge (((\exists x'.P_1) \wedge \neg(\exists x'.P_2)) \Leftrightarrow ((\exists x'.P_1') \wedge \neg(\exists x'.P_2)))$.

Parameters: The parameters are two transitions $a[x : P_1]$ and $a[x : P_2]$ and a new update expression $[x : P_1']$ for transition $a[x : P_1]$.

Transformation: The pattern replaces the original update expression $[x : P_1]$ with the new update expression $[x : P_1']$.

Diagram:

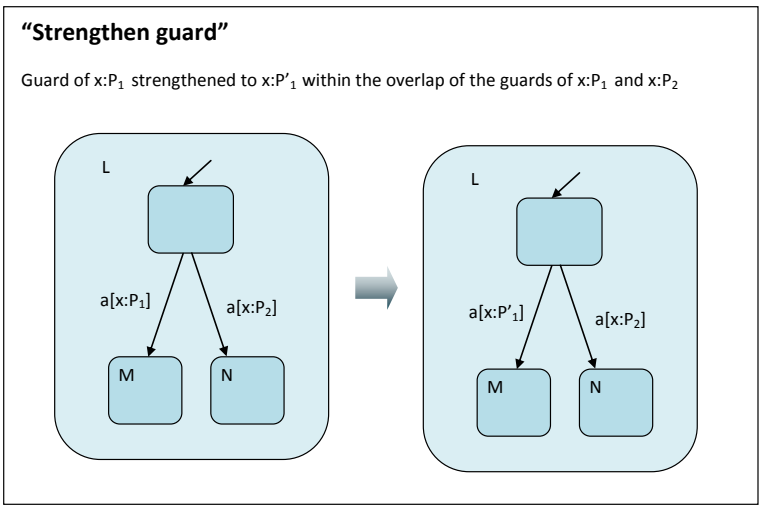

Figure 5.9: Strengthen guard

5.2.3.3 Patterns for adding CoSta state machine constructs

The pattern theme in this section is *adding CoSta design constructs.*

Name: *Split transition*

Type: Refactoring

Rationale: This pattern splits a transition on its update expression into two transitions. Splitting a transition along its guard or update enables different parts of the original update expression (and the continuation behaviour for each of the new transitions if the state is split as well) to be refined in different ways. The inverse of this pattern (*Combine transitions*) is presented in Chapter 4, section 4.3.3.1.

Constraints: This pattern is applicable to a transition $a[x : P]$. The transition is split into two transitions $a[x : P']$ and $a[x : P'']$.

The side-condition verifies $\vdash P \Leftrightarrow (P' \vee P'')$

Parameters: The parameters are the transition to be split on its update expression $a[x : P]$ and the two new update expressions that split the original $[x : P']$, $[x : P'']$.

Transformation: The pattern replaces the original transition $a[x : P]$ by two new transitions $a[x : P']$ and $a[x : P'']$ between the same source and target states.

Diagram:

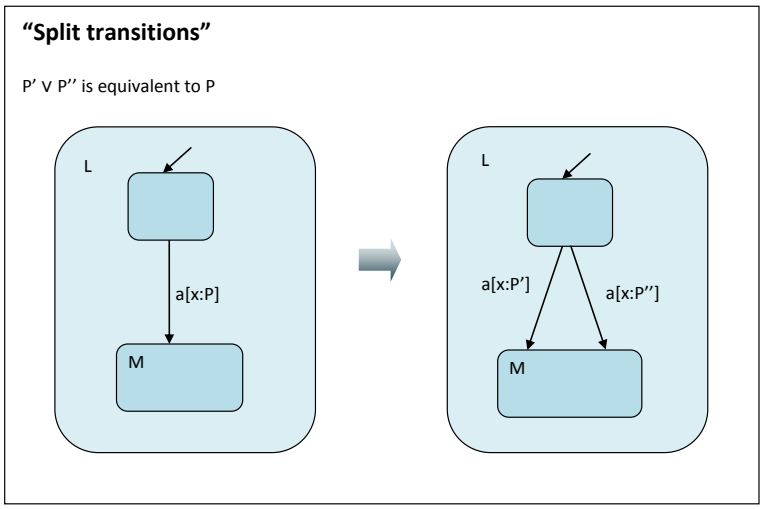

Figure 5.10: Split transition

5.3 General patterns

This section discusses the most general patterns in the catalogue.

Name: *Refinement check*

Type: Refinement

Rationale: If a required refinement is very specific to the application under consideration it may not be predefined. A general refinement pattern permits refinements to be proposed and checked by the engineer. The engineer manually constructs a refinement of a given system, without applying patterns and applies the *Refinement check* pattern to prove that the proposed refinement is correct. The engineer proposes a refinement for a *component* of the overall model. The proof burden is lessened considerably by the compositionality of the underlying theory.

Constraints: The pattern is applicable to models expressed as contracts, state machine diagrams or mixed designs. The source model is a selected component of the current design. The target model is a parameter and the potential component to replace the source component, if the refinement is valid. The side-condition verifies a refinement relation holds between the source and target models.

Parameters: The parameters are the source and target models.

Transformation: The target model replaces the source model.

The general equivalence pattern is an extension of this pattern where the side-condition checks the refinement holds in both directions. This could be implemented as a compound pattern that applies the *Refinement check* pattern twice, the second time swapping the source and target models around. Compound patterns are a research area to be considered for further work and will be discussed later in the last chapter of the thesis.

The general patterns could be applied, for example, to check if removing a transition results in a refinement or an equivalence of the original model. Refinement is the same as equivalence for deterministic models as it is not possible to refine deterministic models any further. If a deterministic model has a transition removed but the new model is equivalent or a refinement of the former model there are two possibilities, either the removed transition is infeasible, its guard (possibly implicit) is equivalent to *False* (i.e. it can never be triggered) or the removed transition is partially feasible (its guard is not *False*) but the source state is unreachable. For a nondeterministic model, equivalence, when a transition is removed, can mean either the branch was redundant and the behaviour was included elsewhere or the branch was unreachable. If the new model is a refinement and not an equivalence, the transition is not unreachable but instead a nondeterministic branch (the actions and guard are included on another branch) has been removed.

5.4 Omitted patterns

This section discusses why certain patterns do not feature in the catalogue. Some patterns common in the literature are not included in the catalogue as although they may not violate Shared Variable Ready Simulation (SVRS) directly, they are not useful during the design process or they may encourage bad designs. For example, although *Introducing a new state* (a common refinement in the literature) does not violate SVRS, patterns for Contractual State Machines permit states to be introduced only when contracts are refined to mixed designs (e.g the *TotalisedEnable* and *If* patterns). It is not useful to introduce states on other occasions as further transitions cannot be added and the states will remain disconnected. Similarly, refactorings to *Introduce a transition with a false guard* or *Remove/add an outgoing transition from an unreachable state* are described in the literature but are not useful for refining Contractual State Machines.

Additionally there were other refinements and refactorings common in the literature that are not applicable to Contractual State Machines as they are based upon syntactic features of statecharts that Contractual State Machines do not support. For example the

refinement to *Strengthen/weaken an event constraint* where the syntax allows multiple events joined with conjunction and disjunction operators; or the refactoring to *Move an action* from a transition to a state entry or exit action where the syntax allows states to have actions associated with them.

Some refinements/refactorings common in the research literature for state machines have not been included as they would violate SVRS refinement rules. For example, common refinement steps in the literature are to *Remove a transition*, *Create a transition* or *Weaken the guard of a transition*. Under SVRS these transformations to a state machine model are not permitted as they change the enabled actions in a way that may no longer satisfy the contract.

Some refinements/refactorings common in the literature are permitted for Contractual State Machines but with different side-conditions to prevent violation of SVRS refinement rules. For example a common refinement in the literature is to *Modify a transition*. Under SVRS, transitions can be modified but in a restricted way. For example, *Strengthening a guard* under SVRS is only permitted when there is an overlap with the guards of other transitions between the same source and target states and *Strengthening the update expression* is only permitted within the guard of the transition. *Removing a state* is a common refinement in the literature, this is not always a permitted refinement under SVRS. If the state has incoming transitions this transformation may change the enabled actions which no longer guarantees that the design satisfies the contract. A state is only permitted to be removed if it is unreachable or redundant.

A pattern for Contractual State Machines that removes transitions to a state with a *False* contract on initial consideration appears to be something the designer should be permitted to do. However this is not the case, for example if a contract $\langle [a] \rangle$ is refined to a design with a start state having an a transition to a target state with the contract *True*, and the contract *True* is then strengthened to *False*. It is not a permitted refinement to remove the a transition whose target state now has a contract *False*, as the original contract $\langle [a] \rangle$ is no longer satisfied. As explained in Chapter 4, it is intended that the CoSta refinement pattern to *Strengthen a contract* will prevent contracts being strengthened to *False* by a side-condition that checks for feasibility of the new contract.

5.5 Summary

A catalogue of refinement and refactoring patterns for Contractual State Machines has been presented in this and the previous chapter. The approach to pattern discovery has been systematic which gives us some confidence in their completeness. Although formal proof is not possible yet as the theoretical research is too immature and a case by case analysis of the patterns cannot be conducted until the model checker (HST) supports a refinement check, the case study, presented in Chapter 7, offers some defense towards the argument that the patterns are valid.

Chapter 6

Tool support and implementation

This chapter is a precursor to evaluation discussed in the next chapter. It will address the implementation of a design tool to support Contractual State Machines (CoSta) and the application of refinement and refactoring patterns. In this chapter we describe summarised examples of the software developed for this thesis. The appendix contains more detailed examples of the code for the design tool and patterns for CoSta.[1] The design tool for CoSta supports formal verification techniques for refinement/refactoring of designs based on the application of patterns from an integrated library of refinement and refactoring patterns. The prototype graphical editor for CoSta has been implemented using the Eclipse Modelling Framework (EMF) [77] and Graphical Modelling Framework (GMF) [75]. The CoSta contract language and the expressions on transition labels were encoded using the EMFText tool [185]. The refinement and refactoring patterns were implemented as a set of wizards, using the Epsilon Wizard Language (EWL) [140], for update-in-place model transformation. The toolset has been integrated with a model checker (HST) to allow a richer set of (semantic) side-conditions for patterns to be checked. After the user has selected the part of the model they wish to refine and the pattern to be applied, some patterns require further input from the user. For example, the *If* or *Enable* patterns that refine a state with an unelaborated inner contract (selected by the user) to a state machine model, additionally require the user to input details for the new transitions to be added. The pattern may call HST to verify a conjecture, [2] for example, to ensure that there is no overlap between the update expression in the contract and the update expression for the

[1] The software for this thesis can be found at `https://svn.cs.york.ac.uk/extsvn/sosym/costa-wizards`, access credentials may be obtained from the author or supervisor.

[2] The refinement check for Contractual State Machines is not yet implemented in HST.

new transition.

6.1 Introduction

A tool suite has been developed that integrates a modelling tool for CoSta with a pattern application tool so that patterns can be applied during the design process. The decision was made not to tightly integrate the new design tool with an existing tool suite such as Mathworks' MATLAB [248]; this approach was taken for example in the PFS/SSA project [250]. We now justify our decision.

An important aim of the research is to achieve a model-driven approach to the engineering process based on the application of refinement and refactoring patterns for CoSta. With this aim in mind, Eclipse [78] was chosen for development over a customisation approach as it is ideally suited for developing graphical editors with support for model transformation, in contrast to the MATLAB environment, for example.

Customising an existing tool set would have required considerably more resources in terms of time and effort because anecdotal evidence from the PFS/SSA project [249] suggests that modifying the MATLAB/Stateflow user interface is really challenging. Customising an existing tool also would not have provided benefits such as being able to directly use the existing editing and animation/simulation tools as they are not suitable. One reason for this is because the semantics of CoSta (based on STGA) is different, being a simplified form of statecharts modified to make them amenable to formal analysis, e.g. permitting nondeterminism and with a restricted form of parallelism. Additionally, contracts are the focus of the approach, and existing tools such as Statemate [120] for example, would have required a great deal of work to accommodate them.

Developing a new system avoids any tie-in to proprietary software and possible problems with maintenance and obsolescence in the future. Therefore there were not many benefits to customising existing tools, but there were potentially a lot of pitfalls. It was felt that a greenfield approach would avoid any negative constraints imposed by prior work and permit experimentation with different possibilities with regards to the semantics of the language and the architectural and design decisions for the software. Direct reuse was rejected as it compromised control of the software architecture and semantics of the language.

Patterns are required to refine an abstract contract into a concrete design. They rely on other model management tasks, primarily model transformation. There are a variety

of approaches to model transformation, (i.e direct manipulation, structure-driven, operational, template-based, relational, graph transformation-based, hybrid) and important differences in their features were compared in [69]. Interactive, in-place, horizontal and vertical endogenous model transformation will support refinement and refactoring patterns.

Methods for applying patterns and implementing the pattern application tool were compared. Pattern application could be implemented with model transformation tools or graph transformation tools [36, 38, 140]. Refinement and refactoring patterns for statecharts could be formalised as graph transformations [39, 132, 167, 168, 246, 252], e.g., as supported by approaches like VIATRA2 [79] and AGG [245]. An advantage of using graph transformation techniques is that they provide a formal basis for model transformation supporting analysis of conflicts and dependencies [106, 213]. Graph transformations are particularly suitable for precisely describing and implementing refactoring transformations on arbitrary graphs. Model transformation tools may be considered more mature and flexible, with the ability to implement some types of pattern that graph transformation tools are not particularly suited to.

The structural transformations for the refinement and refactoring patterns tend to be fairly simple. The patterns already have an underlying formal semantics therefore it makes little sense to recast this as an alternative formalism, (i.e. graphs) in order to express some of the more complex structural rewrites in an arguably nicer way. In addition it is not possible to express or map some transformations into graphs. The pattern transformations heavily involve text with decision procedures based on LTS/STGA and bit-vector arithmetic (SMT) and transformations from temporal logic to state machines. Therefore a very flexible approach to the implementation of the patterns is needed to satisfy all of these requirements.

Graph transformation supports powerful pattern matching, by matching wildcards to subgraphs, complex transformations can be expressed very succinctly. As the CoSta language supports a compositional refinement theory SVRS, the pattern matching for the model transformations does not need to be so elaborate, it can match element for element and the context does not affect the validity of the transformation. Thus the refinement and refactoring patterns do not require complex transformations mapping across global structures and this in turn reduces the need for the use of graph transformation to help to control such complexity. For all of the aforementioned reasons model transformation tools

instead of graph transformation tools are used to implement refinement and refactoring patterns.

The choice of software for implementing the design tool for CoSta was between Eclipse GMF/EMF [75, 77] and the Generic Modelling Environment (GME) [90]. Eclipse is suited to developing "industrial strength" editors. It is multi-platform, open source and Java-based. It is the most popular framework of its kind with a large number of plug-ins available and seamless integration with other modelling and programming tools. GME on the other hand is not in such widespread use and not as well supported. A drawback to GME is that metamodels can be low-cohesive (difficult to maintain, test, reuse and understand) [13]. There are alternative technologies based on different standards with similar approaches, (e.g. Microsoft Domain Specific Languages framework [174], Metacase MetaEdit+ [170], Xactium XMF-Mosaic [263]) but they tend not to be as powerful or flexible as the Eclipse platform. Those that are as powerful are proprietary, leading to the same concerns expressed above for MathWorks MATLAB/Stateflow/Simulink.

It was noted above that update-in-place can support refinement and refactoring patterns. Many languages of different styles have been proposed for mapping transformations. Some mapping transformation languages have been shown to be of use for update transformations in the large. By contrast update-in-place transformations are largely unsupported by model transformation frameworks. Existing languages for mapping transformations such as QVT [189] and ATL [128, 129] cannot be used in their current form for this purpose as they are intended to operate in a batch style where everything is defined before the transformation begins with no user interaction during the model transformation process. This is not sympathetic to the task of model refinement and refactoring which is inherently user-driven with overriding requirements for fine-grained user control.

The Eclipse-based, Epsilon Wizard Language (EWL) [140] is however suitable for user-driven, in-place model transformation. Update-in-place model transformations automatically create, update or delete model elements based on a selection of existing elements in the model and possibly user input in a user-driven fashion. A number of different model management tasks were required by the toolset such as model-to-text transformation and update-in-place model transformation. There would have been challenging issues with reuse, interoperability, integration and consistency to resolve if separate task specific model management languages were used as they are generally inconsistent with each other. Epsilon supports a wider range of model management tasks than comparable platforms

and provides better integration due to the common infrastructure on which the languages have been built.

In summary, Eclipse and Epsilon provide software tailored specifically to the tasks of developing modelling and model management tools (most importantly in-place transformations) and they offer maximum flexibility by being open source and based on the MDA standard. After careful consideration it was concluded that Eclipse meta-tools would be the most suitable choice for prototyping the software. In particular EMF/GMF for the graphical editor and Epsilon EWL for the refinement and refactoring patterns.

6.2 Prototyping the modelling tool

This section discusses how the software for the tool support was prototyped. A general iterative, incremental lifecycle was adopted. The design process began with an initial analysis phase followed by several design, implementation and testing iterations. Following the analysis phase a conceptual architecture was proposed. This was refined into a technical design and then implemented. Throughout the design and implementation iterations, testing has been conducted to assess the correctness, quality and utility of the new features. Errors and omissions identified during testing iterations were fed back to further design and implementation cycles.

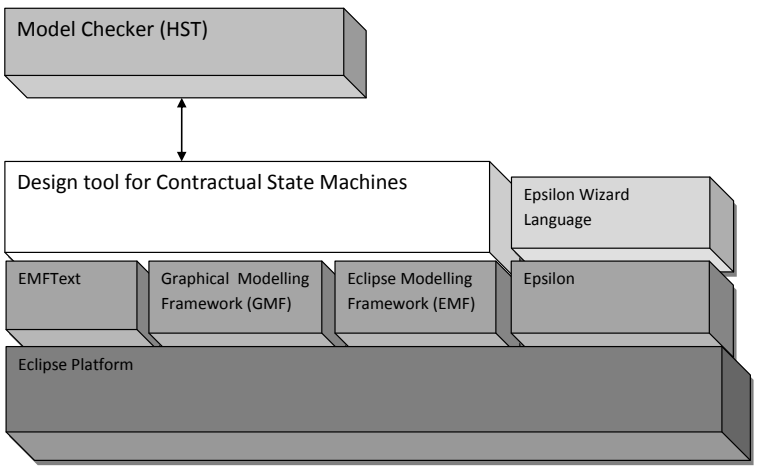

Figure 6.1: Conceptual architecture of tool environment

Figure 6.1 shows the conceptual architecture of the tool environment. The editor for CoSta and the pattern application software are loosely coupled. Patterns (EWL wizards) interface with the model checker (HST) to check side-conditions.

Although a set of refinement and refactoring patterns have been established and implemented for single refinement and refactoring steps it is anticipated that the discovery of new compound/derived refinement and refactoring patterns will be ongoing. With this in mind it was considered that the most important quality attributes for the prototype software were, modifiability, extensibility, usability and performance.

6.2.1 CoSta design tool

This section discusses the design and implementation of the CoSta design tool. The metamodel/abstract syntax for the CoSta graphical state machine modelling language (see Section 2.13.2, figure 2.3) is expressed in Ecore for EMF. Emfatic is a language designed to represent EMF Ecore models in a textual form and was used to describe the metamodels for this toolset [11]. Emfatic provides tools to convert between the Emfatic source and the corresponding Ecore model. It was chosen in preference to the graphical Ecore editor provided by EMF as it represents models in a human-readable syntax rather than metamodelling based on EMF's tree views.

The Emfatic metamodel for the CoSta graphical state machine modelling language was defined and extended with EuGENia annotations [83], (see Appendix A.1). EuGENia was used as a front-end to GMF because it automatically generates all of the files needed to implement a GMF editor, (.gmfgraph, .gmftool and .gmfmap) from a single annotated Ecore file. The annotations relate to the graphical appearance of domain model elements, their concrete syntax, such as shapes and lines and their properties and define the commands for the editor toolbar. An advantage of using EuGENia is that it reduces all of the steps in a typical GMF workflow to a single step and thus saves times and effort.

The syntactic model (figure 2.3) presented in Chapter 2 describes the graphical state machine modelling language. It does not elaborate on the abstract syntax for the text-based CoSta contract language or the text-based language for transition labels. It summarises contracts as being of type *ContractFormula* and transition labels of type *Event-GuardActions*. A full listing of the concrete syntax (described in an EBNF-based language) and abstract syntax (described in Emfatic) for the CoSta contract language is given in Appendix A.2 and for the transition label language in Appendix A.3.

EMFText [185] was used to develop the editors and parsers for the CoSta contract language and the language for transition labels. The editors and parsers for these text-based languages were integrated within the graphical modelling tool for Contractual State Machines to parse a state's inner and outer contract and transition labels. For the Eclipse platform there are two open source tools, Xtext [80] and EMFText for developing textual domain specific languages. Both tools support the definition of a textual language and produce an eclipse-based editor for the language that provides features such as syntax highlighting, parsing with errors/warnings, auto-completion and outlining. They both use EMF for their abstract syntax and rely on ANTLR for parsing the concrete syntax [12].

Both tools can be integrated with any of the transformation frameworks available in Eclipse but Xtext provides automatic configuration for the Xpand/Xtend framework. A drawback to this is that although other transformation frameworks can be used with Xtext, (which is the requirement for this development as the transformation framework is Epsilon), extra effort is required to deconfigure and reconfigure the project. By contrast EMFText does not include a transformation framework by default, but offers the choice of whichever toolset is preferable. This flexibility made it the best option for implementing the parsers as it was not necessary to "deconfigure" the project initially.

A language specification for EMFText consists of an Ecore language metamodel and a concrete syntax specification. The Emfatic description required by EMFText of the Ecore metamodel for the abstract syntax of the CoSta contract language (see Section 2.13.1) is given in Appendix A.2. Additionally EMFText requires a concrete syntax for each abstract syntax describing the textual representation of all metamodel concepts. EMFText provides the cs-language for concrete syntax specification, with syntax specification rules derived from the EBNF language. The concrete syntax definition required by EMFtext for the CoSta contract language is given in Appendix A.2.

Appendix A.3 contains listings of the abstract and concrete syntax descriptions required by EMFtext for transition labels (see Section 2.13.3). EMFtext was used to generate a textual editor, parser and printer for each language from their abstract and concrete syntax definitions. The parser parses language expressions to EMF models and the printer prints EMF models to language expressions. EMFtext generates independent code in the form of plugins.

Customisations of the code generated by EuGENia, EMF and GMF were required for non-standard functionality. Parts of the editor were non-standard, for example, default

transitions are implemented as a node with an image instead of the standard GMF shapes (rectangle, ellipse etc.). This customisation required the creation of a plugin for the custom figure, which was then associated with the node for the default transition.

Customisations were also required to integrate the EMFtext parsers for the inner and outer contracts and transition labels within the CoSta editor. It was necessary to adapt the Emfatic metamodel for the CoSta language to import the Ecore specifications for the language metamodels (representing the abstract syntax) for contracts and transition labels, and include new derived fields for the parsed inner and outer contracts and transition labels. Additionally new Java code was added to instantiate the derived fields by invoking the EMFtext parsers to parse the strings (for the inner and outer contracts and transition labels) and return the Ecore models for the parse trees and assign them to the derived fields. These derived fields were then accessible from the EWL wizards.

Figure 6.2 demonstrates the parser for the contract language parsing an expression (top pane) to the parse tree represented as an EMF model (bottom pane).

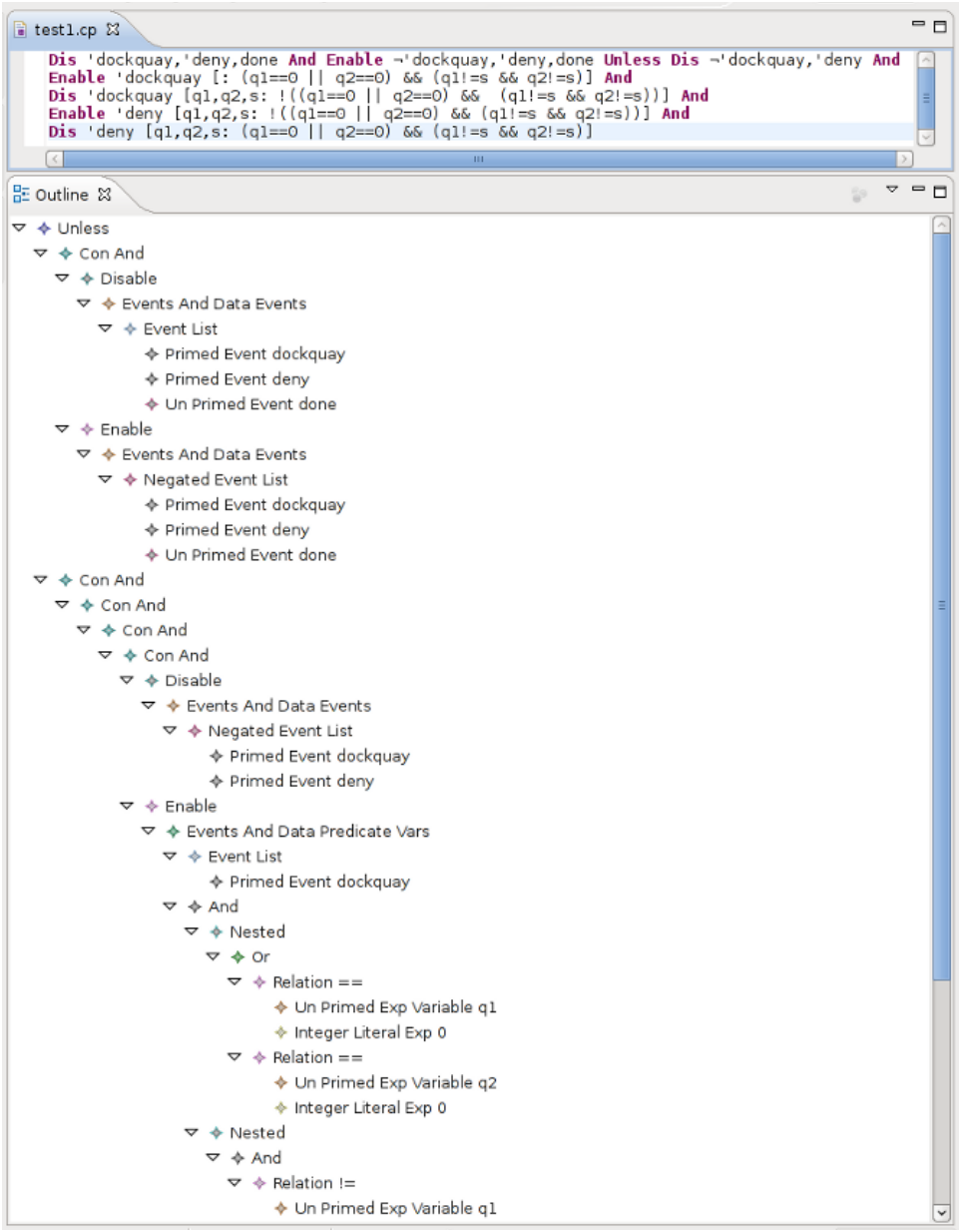

Figure 6.2: Parsed contract

Figure 6.3 demonstrates the parser for the transition syntax parsing an expression (top pane) to the parse tree represented as an EMF model (bottom pane).

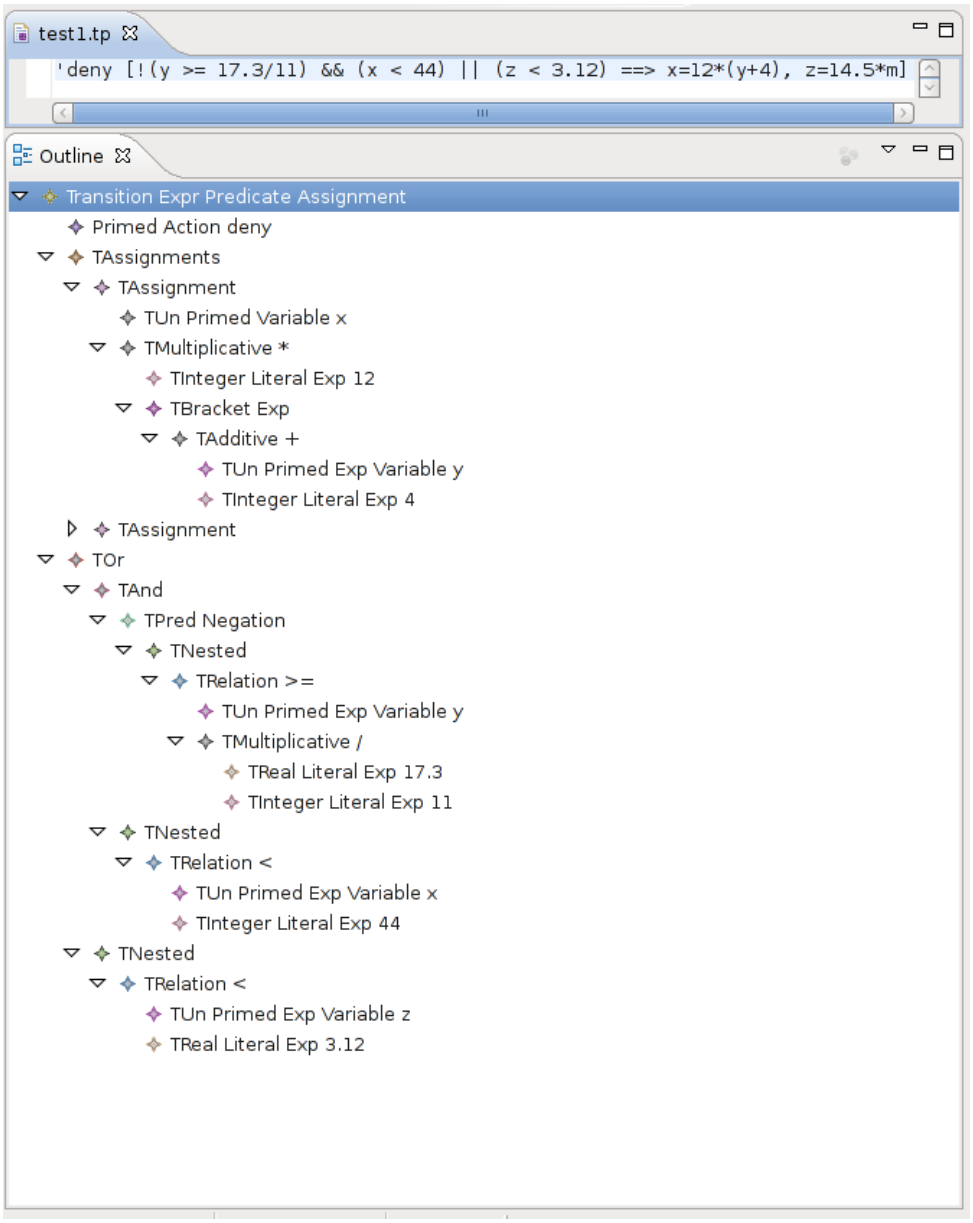

Figure 6.3: Parsed transition expression

Figure 6.4 and figure 6.5 illustrate use of the prototype editor for designing CoSta models. The modelling tool is for the design process once the top-level open contracts have been defined and does not deal with specification of the closed contract. Using the design tool the modelling process begins with the creation of a single state whose inner contract is defined as the initial/top-level open contract for the system. Patterns are repeatedly applied by the user until the required final design is achieved. To apply a pattern within the design tool, the user selects the part of the model to be refined and the tool displays a list of the available/applicable patterns from which the user selects. The tool performs the model transformation for the selected pattern and the user can then continue to apply further patterns to the refined model until the design is complete.

Figure 6.4 shows an example CoSta model and figure 6.5 shows the EMF model representing the parse tree for the example CoSta design.

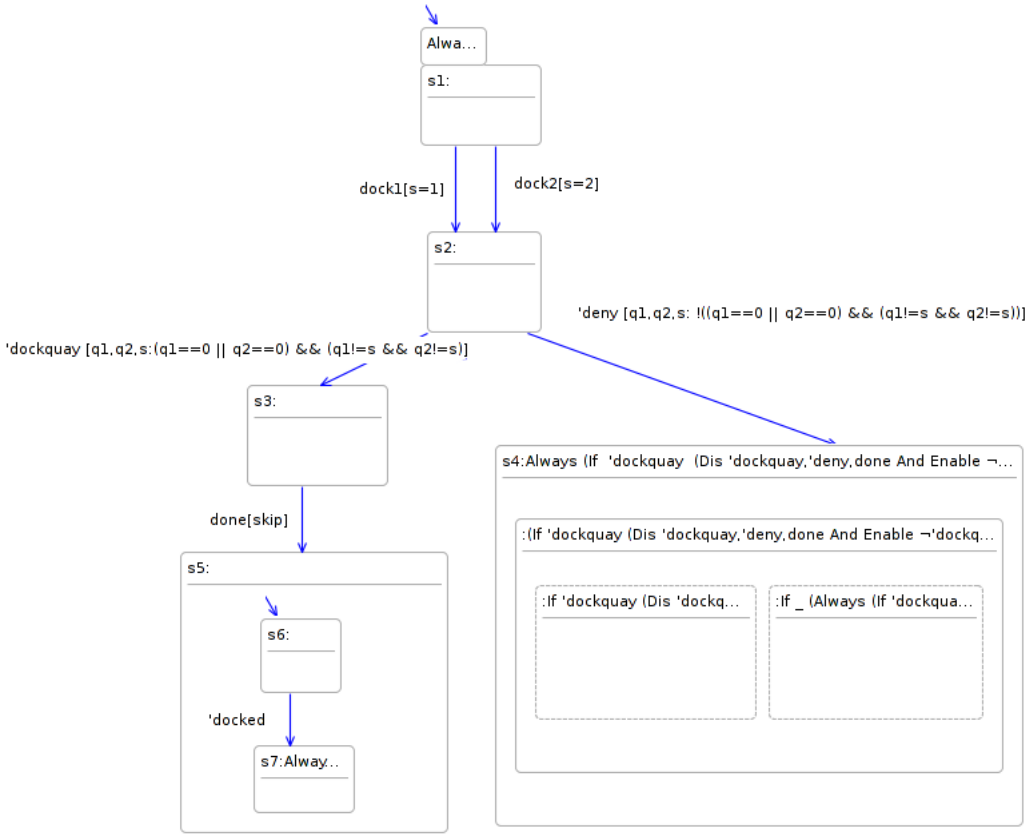

Figure 6.4: Example CoSta model

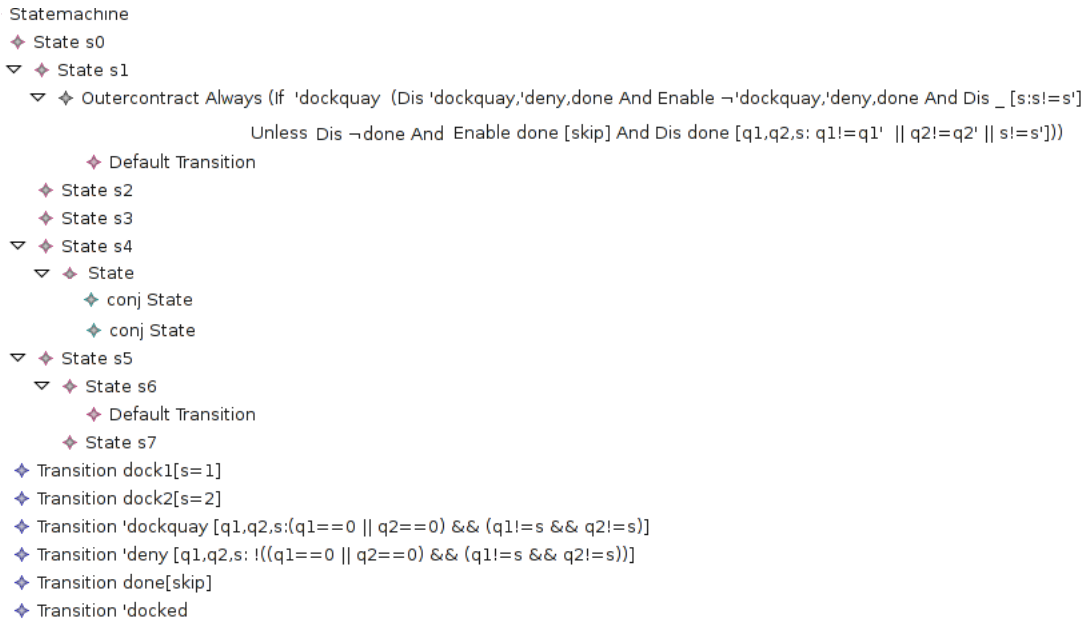

Figure 6.5: EMF model for example CoSta design

6.2.2 Patterns

This section describes the implementation of the pattern application tool to enable patterns to be applied during the design process. A catalogue of patterns were implemented as a set of EWL wizards. The wizards and the EWL interpreter are loosely coupled with the GMF editor for CoSta. The EWL interpreter controls the user-driven selection and application of patterns. Pattern application instantiates a pattern by matching wildcards in the pattern to syntactic and diagrammatic structures in the design.

An example pattern implemented as an EWL wizard is discussed, further EWL wizards can be found in Appendix A.4. Customisations were required to integrate the EMFtext parsers and printers with the EWL wizards. The EMFtext parsers for the contract and transition languages are invoked when the derived fields for a CoSta model (parsedinnercontract, parsedoutercontract, parsedtransition) are accessed within a wizard. They generate EMF models for the parse trees for the contract and transition expressions. In an EWL wizard the EMF model for a parsed expressions can be accessed by declaring a variable and assigning the derived field to it. The example wizard, which is presented later on in this section, demonstrates how an inner contract is parsed and its parse tree accessed.

To implement non-standard functionality, Epsilon supports the Tool concept. A tool is a normal Java class which can be instantiated and accessed from the context of an EWL program. An Epsilon Tool was created to invoke the EMFtext generated printers for the CoSta contract language and transition expressions which are used to derive the corresponding textual expressions from the EMF models representing their parse trees from within a wizard.

A plugin was created with two classes, one for contracts and one for transition expressions. Each class takes the Ecore model for the parsed expression, invokes the relevant EMFtext printer and returns the corresponding textual expression. The plugin was declared as an Epsilon tool to enable any routine within the class to be invoked within an EWL wizard. The example wizard (presented later on in this section), demonstrates how the classes from the Epsilon tool, printlanguageTool, are called from within a wizard to generate the textual representation of a parsed inner contract. Customisations were also required to enable the wizards to interface with the model checker (HST) and this will be discussed in the next subsection.

Several patterns were presented in Chapters 4 and 5 which had complex side-conditions such as checks for feasibility, refinement and behavioural intersection. However, the functionality required to implement these patterns fully was not available via the model checker. The model checker (HST) supports evaluation of quantifier-free conjectures on data. For more complex side-conditions inspection is used as a temporary alternative, where the user is prompted to confirm side-conditions hold. The use of these patterns is not advisable without the checks being conducted by the model checker (HST).

A description of the implementation of a pattern as an EWL wizard follows. A more detailed, precise description of the refinement/refactoring patterns as opposed to how they have been implemented is given in Chapters 4 and 5. The *If* pattern refines a contract to a CoSta model. The wizard is applicable if the user selected component is a state. The side-condition checks that the state's inner contract is an *If* operator expression, (i.e. of the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ"). The *If* operator specifies constraints on the behaviour following a particular set of events (whatever the values of the variables, and for all events in the set). When accompanied by a variable update, the constraints need only apply from a *before* state satisfying the guard (or feasibility) of the assignment, and following a consistent update. $[a[x : P]]\phi$ expresses that an a action with an update consistent with $x : P$, (i.e. from the domain of

P) must behave like ϕ. The action a in the contract $[a[x : P]]\phi$ can be generalised to a set of actions, a negated set of actions or the underscore. If no update is specified in the contract, transitions with any update from any source data state have a consistent update and must behave like ϕ if the transition event is in the contract *eventlist*. If an update expression $[x : P]$ is specified in the contract, transitions with an update $[x : P']$ consistent with and from a source state satisfying $[x : P]$ (i.e. lies within that of the contract) must also behave according to ϕ if the transition's event is in the contract's *eventlist*.

If the contract specifies a list of events, transitions with some event from the set must behave like ϕ if the updates are consistent or *True* otherwise. If the contract specifies a negated list of events, transitions with some event not in the set must behave like ϕ if the updates are consistent or *True* otherwise. If the contract specifies an underscore for the event, transitions with any event must behave like ϕ if the updates are consistent or *True* otherwise.

A transition can be added with any action and update behaving subsequently as ϕ. A transition can be added with a different action or an update Q where $Q \Rightarrow \neg P$, behaving subsequently as *True*. It is possible to refine $a[x : P]$ to any event and update to a state that behaves like ϕ. A transition which behaves consistently with $a[x : P]$ has to behave like ϕ but a transition that does not behave consistently with $a[x : P]$ can behave in any way as the continuation behaviour is *True* which can then be refined to anything including ϕ. When introducing transitions that do not behave like ϕ, (with subsequent behaviour *True*) it is necessary to ensure that assuming the event is the same their updates are not consistent with $[x : P]$ and there must be no overlap.

The state's inner contract is parsed to an EMF model by the EMFtext parser for the contract language. Similarly the transition expressions that are input by the user are parsed to EMF models by the EMFtext parser for the transition language. The parse trees permit easier manipulation and comparison of the contract and transition labels. To check if transition parameters input by the user are consistent with the contract, the model checker (HST) may be invoked to decide conjectures for side-conditions. These are described below.

The wizard refines the contract *If* to a state machine model. The wizard prompts the user for the new transitions. The wizard introduces a set of substates for the selected component consisting of a start state with outgoing transitions which are input by the user. The transitions each target a new state whose inner contract is either set to ϕ or

otherwise *True*. A transition targets a new state with an inner contract ϕ if it has an event which is one of the events specified in the contract as well as an update consistent with the update expression in the contract. A permitted refinement is to weaken the update expression.

The wizard prompts the user, firstly, for the transitions with subsequent behaviour *True*. For each of them the side-condition checks that either the event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or the update expression for the transition input by the user is not consistent with the contract update expression, (the update expressions are disjoint and there is no overlap). The model checker (HST) may be called to verify this. Then the user is prompted for the transitions with subsequent behaviour ϕ. It is not necessary to perform any checks for these transitions as if the conditions for the subsequent behaviour ϕ do not hold, then the subsequent behaviour is *True* and ϕ is a refinement of *True*.

A pattern is instantiated by user-guided matching against the abstract syntax of the model. The pattern application process involves the following steps: The EWL interpreter offers a list of possible patterns where the pattern constraints are satisfied, (the selected model component is well-formed and an instance of the abstract template of the pattern). The user selects the required pattern and supplies parameters. EWL then performs the transformation. The body of the wizard is executed and the highlighted component is replaced with an instance of the concrete template.

Figures 6.6, 6.7, 6.8 illustrate the steps in this process.

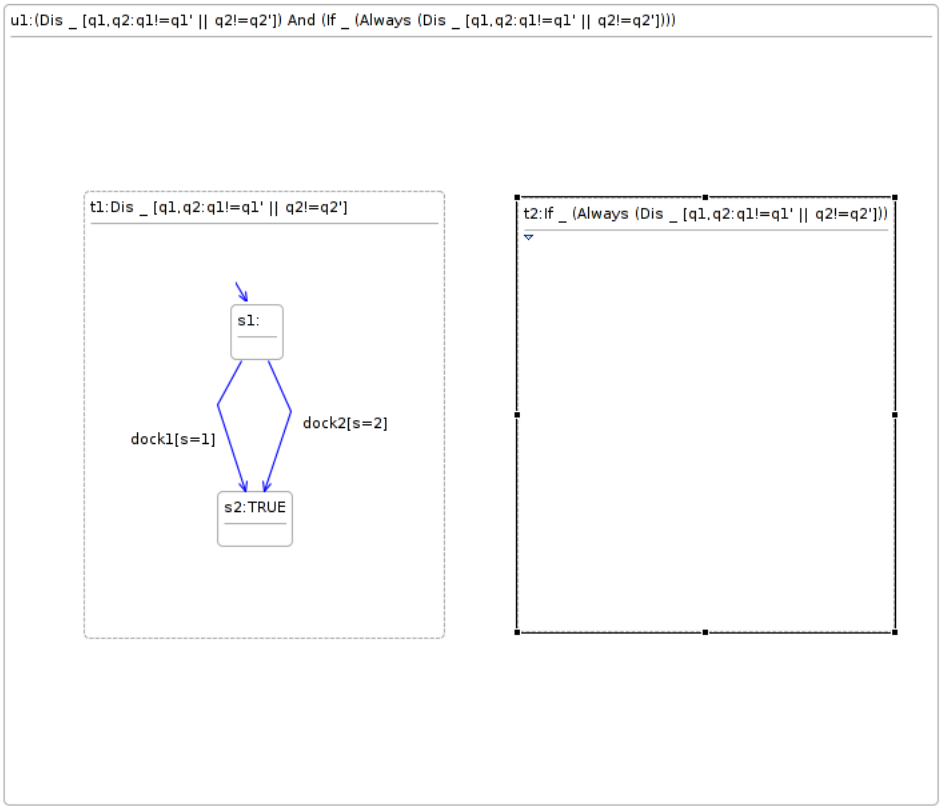

Figure 6.6: Selected state with If contract

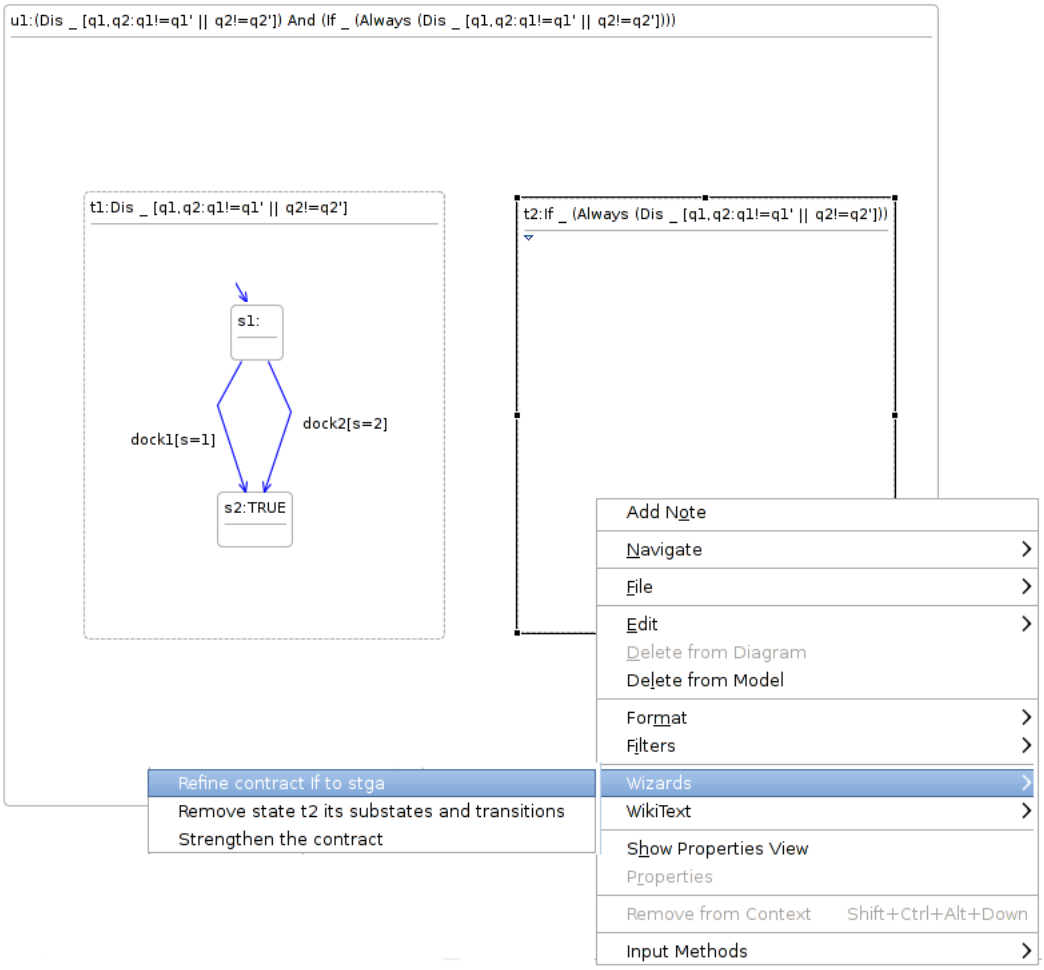

Figure 6.7: Selected wizard - Refine contract If to a CoSta state machine

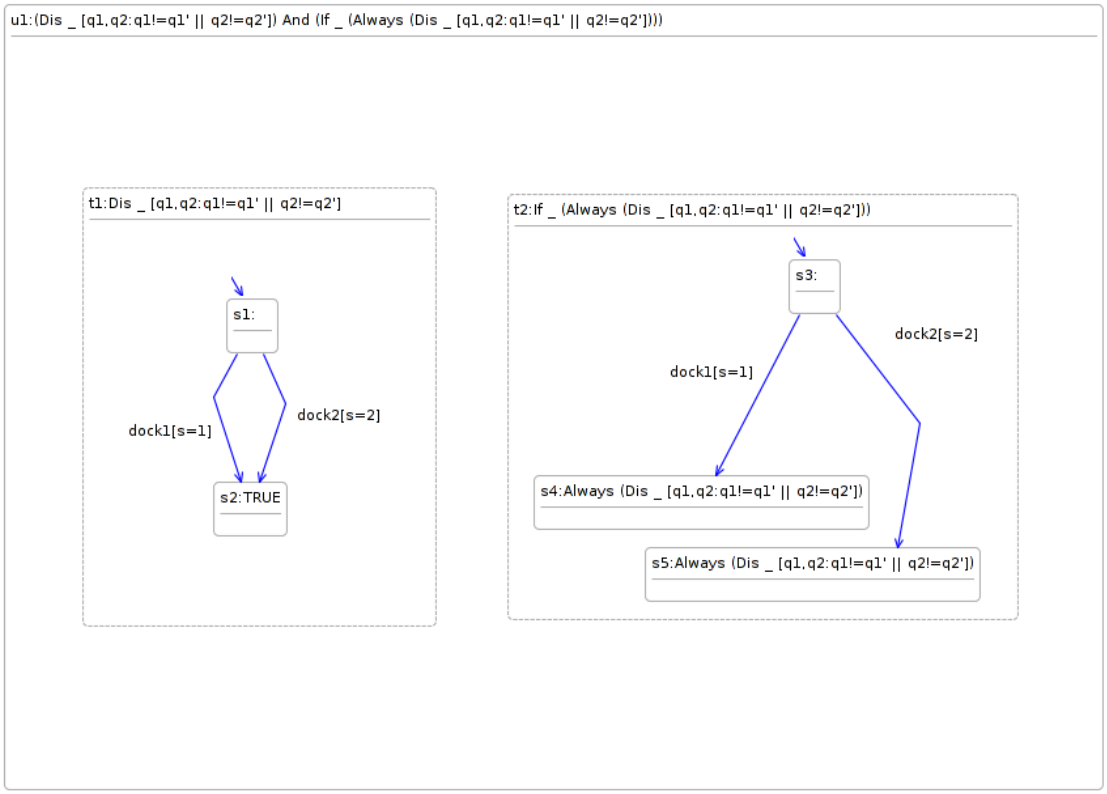

Figure 6.8: The If contract refined to a CoSta State Machine

Excerpts of the EWL code for the *If* wizard follow. The wizard refines the *If* operator to a state machine. The code excerpt declares variables which include a reference to the Epsilon tool, *printlanguageTool*, with routines that invoke an EMFtext printer to derive the corresponding textual expression from an EMF model representing the parse tree. This part of the wizard also prompts the user to confirm that they wish to refine the inner *If* contract of the selected state to a CoSta state machine.

```
1  wizard wizIf {
2  guard : self.isKindOf(State)
3  and (self.parsedinnercontract.isTypeOf(If))
4  title : "Refine contract If to a CoSta State Machine"
5  do {
6  --declare a reference to the Epsilon tool printlanguageTool
7    var printlanguageTool : new Native("printlanguage.printlanguageTool");
8    ...
9    ...
10 --Variables for contract expressions parsed to EMF models
11   var pc1:Formula;
12   var pc2:Formula;
13 --Variable for transition expression parsed to an EMF model
14   var te:TransitionExpression;
```

```
15   ....
16   ....
17   if (UserInput.confirm('Refine contract If to a CoSta State Machine' + ' ?', true))
18   {
```

Listing 6.1: Declare variables

The next code excerpt from the *If* wizard invokes the EMFtext parser to parse the inner contract for the selected state. The code inspects the EMF model (parse tree for the contract expression) to determine if there is an *eventlist* and its type. The code extracts the set of events and prompts the user to enter the new transitions with subsequent behaviour *True*. For each of the transitions it is checked that either the event is not in the contract eventlist or it is in the negated contract eventlist or the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1). The model checker (HST) is invoked to determine if the update expressions are disjoint. Finally if the checks are successful the new transitions having subsequent behaviour *True* are added to the CoSta model that will replace the contract.

```
1  --Invokes the EMFtext parser to parse contract expression
2     pc1 := self.parsedinnercontract;
3     while (pc1.isTypeOf(ConNested))
4          pc1 := pc1.body;
5     if (pc1.eventsanddata.eventformula.isTypeOf(EventList) or
6     pc1.eventsanddata.eventformula.isTypeOf(NegatedEventList))
7     {
8  --get the list of events
9     }
10 --prompts user for transitions with subsequent behaviour True
11     s1 := UserInput.prompt('Enter required transitions with subsequent behaviour
           True');
12 --Create bag of parameters
13      ....
14 --calls acceptProofObligation to invoke HST to check the
15 --update expressions are disjoint.
16
17 if ((pc1.eventsanddata.eventformula.isTypeOf(EventList) and
18           (events.excludes(e1.strip().ev()) or
19             (events.includes(e1.strip().ev()) and acceptProofObligation(updateExpr2
                   ,updateExpr1)))) or
20            (pc1.eventsanddata.eventformula.isTypeOf(NegatedEventList) and
21             (events.includes(e1.strip().ev()) or
22              (events.excludes(e1.strip().ev()) and acceptProofObligation(
                    updateExpr2,updateExpr1)))) or
23            (pc1.eventsanddata.eventformula.isTypeOf(Underscore) and
                  acceptProofObligation(updateExpr2,updateExpr1)))
24     {
25        ....
26 --add a new start state.
27         self.substates.add(substate1);
28        for (t in transitionParameters)
```

```
29          {
30          ....
31  --transition expression is parsed to an EMF model
32          te := t.parseTransitionExpression();
33          ....
34  --add the new transition from the start state to a new target
35  --state with contract True
36          self.substates.add(substate2);
37          substate2.innercontract := "TRUE";
38          transition := Transition.createInstance();
39          transition.source := substate1;
40          transition.target := substate2;
41          transition.label := t.toString();
42          sm.transitions.add(transition);
43        }
44     }
```

Listing 6.2: Add transitions to CSM design

The next code excerpt from the *If* wizard calls the EMFtext printer for contracts to generate a textual representation of the parsed inner contract from its parse tree (EMF model). It prompts the user for transitions with subsequent behaviour ϕ where the *If* contract is of the form $[a]\phi$. The new transitions are added to the CoSta model that will replace the contract.

```
1       ....
2         pc2 := pc1.body;
3       ....
4   --calls the EMFtext printer for contracts
5       s3 := printlanguageTool.printcp(pc2);
6       ....
7   --prompts the user for transitions with
8   --subsequent behaviour Phi
9       s1 := UserInput.prompt('Enter required transitions with subsequent behaviour '
              + s3);
10        --Create bag of parameters
11      ....
12      if (transitionParameters.size() > 0)
13      {
14        ....
15        for (t in transitionParameters)
16        {
17        ....
18  --transition expression is parsed to EMF model.
19        te := t.parseTransitionExpression();
20        ....
21  --Create a new transition for the parameter from the start
22  --state to a new target state with contract Phi
23        ....
24          substate2 := State.createInstance();
25        self.substates.add(substate2);
26        substate2.innercontract := s3;
27        transition := Transition.createInstance();
28        transition.source := substate1;
```

```
29          transition.target := substate2;
30          transition.label := t.toString();
31          sm.transitions.add(transition);
32          }
33        }
34      forceRefresh();
35    ....
36    }
37  }
```

Listing 6.3: Add transitions to CSM design

The final code excerpt from the *If* wizard below shows the routine that invokes the EMFtext parser to parse a transition expression.

```
1  operation String parseTransitionExpression():TransitionExpression
2  {
3    var transition:Transition;
4    var tp:TransitionExpression;
5    transition := Transition.createInstance();
6    transition.label := self;
7  --Invokes the EMFtext parser to parse the transition expression.
8    tp := transition.parsedtransition;
9    delete transition;
10   return tp;
11 }
```

Listing 6.4: Routine that invokes the EMFtext parser

6.2.2.1 Integration with model checking technology

This section describes how the pattern application tool integrates model transformation and model checking technology (HST). HST is used to check the validity of application of refinement and refactoring patterns with complex side-conditions. The Heterogeneous Specification Tool (HST) is currently under development in the research group and can be used for simulating/animating an STGA-like language and deciding quantifier free conjectures on data. Conjectures are decided by an SMT solver (Boolector). Preorder/refinement checking algorithms are not yet implemented. HST is in the process of being extended to support refinement and interpretation of temporal contracts by their characteristic LTS/STGAs.

Patterns use HST to compare arithmetic and Boolean expressions, to determine, for example, if two arbitrary expressions are disjoint or semantically equivalent. HST can determine tautologies. For example to check whether P and Q are disjoint HST can be invoked to decide if $(P \wedge Q \Rightarrow False)$ is a tautology.

EWL does not inherently support the ability to interface with HST, therefore customisation is required to implement the non-standard functionality using the Epsilon Tool concept (in the same manner as described above for integrating the EMFtext printers with the EWL wizards). A plugin is created to start and stop HST, pass commands from the wizard to HST for execution and return the results from HST to the wizard. The plugin is declared as an Epsilon tool to enable any routine within the class to be invoked within an EWL wizard. Here is an example of the routines from the Epsilon tool, (hstTool) being invoked from an operation (acceptProofObligation) within a wizard to evaluate the side-condition for a pattern.

```
1  operation acceptProofObligation(Q,P):Boolean
2  {
3  --HST will check whether P and Q are disjoint,
4  --declare a reference to the Epsilon tool, hstTool,
5   var hstTool : new Native("hst.hstTool");
6   ....
7  --calls the routine from the Epsilon tool to start HST
8   hstTool.startHst();
9   ....
10 --calls the routine from the Epsilon tool to pass a
11 --command to HST for execution,
12 --the command sets HST in STGA mode.
13  hstTool.writeBuffer(hstTool.getBw(),"c stga\n");
14  ....
15 --HST will decide if !(Q && P) is tautological
16  line := vars + ":int16|!(" + Q + " && " + P + ")\n";
17 --calls the routine from the Epsilon tool to pass a
18 --command to HST for execution, the conjecture to
19 --be evaluated (line).
20  hstTool.writeBuffer(hstTool.getBw(),line);
21  ....
22  while (not end and not error)
23  {
24    ....
25 --calls the routine from the Epsilon tool to retrieve
26 --the output from HST
27       line:= hstTool.readBuffer(hstTool.getBr());
28     if ("Added Declaration".isSubstringOf(line)) end:=true;
29     else if ("Command not recognised".isSubstringOf(line)) error:=true;
30    ....
31  }
32  if (error) UserInput.Inform("Invalid HST command -  unable to determine tautology
        ");
33  else
34  {
35 --If a valid conjecture has been input, calls the routine
36 --from the Epsilon tool to pass a command to HST for
37 --execution, the command is to decide the tautology.
38       hstTool.writeBuffer(hstTool.getBw(),"d\n");
39    ....
40    while (not end)
```

```
41     {
42       line:= hstTool.readBuffer(hstTool.getBr());
43       ....
44          if (("theorem".isSubstringOf(line)) or ("Theorem".isSubstringOf(line))) end
                :=true;
45          if ("Not a theorem".isSubstringOf(line)) theorem:=false;
46       ....
47         }
48     ....
49        if (theorem) UserInput.Inform("HST has determined that the tautology is a
              theorem");
50        else UserInput.Inform("HST has determined that the tautology is not a theorem
              ");
51     ....
52    }
53    ....
54  --calls the routine from the Epsilon tool to stop HST
55    hstTool.stopHst();
56    ....
57    accept := UserInput.confirm("Do you agree the proof obligation holds : " + Q + "
          => ~" + P, true);
58  }
```

Listing 6.5: Routine that calls HST to evaluate the side-condition for a pattern

An Epsilon Generation Language (EGL) template [212] has been defined to translate a state machine design into the STGA-like language required by HST. This transformation could be part of a sequence of steps that allows the CoSta design tool to interface with the simulation capability of HST to determine the next set of possible transitions which could be used to provide model simulation facilities within the CoSta design tool.

6.3 Summary

This chapter presented a prototype implementation of the software that enables the design of CoSta models supported by a rigorous design process through the application of refinement and refactoring patterns. The software is developed with Eclipse and Epsilon which are based on open standards. This enables it to be integrated with other software used in the engineering process with minimal effort or easily extended to support additional constructs and concepts.

A drawback with Epsilon EWL is that it does not permit patterns to be combined to form more complex transformations [247]. One way to achieve this is copy and adapt the code of existing wizards. EWL however does support reusability of operations which can be grouped into external libraries, as it is built on top of the Epsilon Object Language (EOL) layer. The same effect of combining wizards to form more complex transformations could

be achieved by capturing the model transformation performed by a wizard as a reusable operation in a library and then combining the operations within a wizard.

The next chapter evaluates the validity of the hypothesis through a case study realised using the prototype software implementation described in this chapter.

Chapter 7

Validation and evaluation

7.1 Introduction

The research hypothesis in Chapter 3 proposed a tool supported, model-based approach to a systematic and stepwise design process that supports refinement of Contractual State Machine designs from abstract specification to concrete model through the application of refinement and refactoring patterns. This chapter evaluates the research hypothesis by applying the proposals to a case study. Sections 7.2 to 7.5 present the case study, Section 7.6 evaluates the results and Section 7.7 describes the overall conclusions. The purpose of the case study is to demonstrate the utility of the refinement and refactoring patterns, (presented in Chapters 4 and 5) and their supporting tools, (described in Chapter 6), for the design of Contractual State Machines. The aim is to evaluate the practicality of the proposed design process and its engineering qualities and identify the main strengths and weaknesses of the approach.

7.2 The case study

This section presents the case study, which demonstrates the refinement of a design from an abstract specification of the system to a parallel concrete one. The refinement process is based on a disciplined and systematic application of patterns: the patterns provide a systematic basis for gradually introducing detail as design decisions are made.

The design process starts with a closed contract. This top-level contract expresses high-level properties such as deadlock freedom and invariants. From the closed contract further design decisions are made and abstract open contracts are specified. The open contracts

express high-level design decisions, for example relating to how actions synchronise or how to sequence the synchronising actions and permitted updates to shared variables. From the open contract the design process proceeds with the application of refinement and refactoring patterns until the completed state machine design is achieved. Patterns to support formal refinement and refactoring for Contractual State Machines have been catalogued and implemented as described in Chapters 4, 5 and 6; as such, we demonstrate use of a number of patterns and as such illustrate the engineering potential of the approach. An aim of the case study is to provide further proof of concept and illustrate the approach which is supported by automated tools.

7.3 The docking system

This section discusses the docking system example presented in the paper [220] that the case study is loosely based on. The example described in the paper focuses on a system for docking ships in a port. The system tracks the entering of ships into the port and maintains a queue of ships waiting to dock. It docks ships at quays and allows ships to leave a quay and the port. A record of which ships are docked at which quays is maintained. Only one ship can be docked at each quay. The system prevents the double allocation of ships to quays. The same ship cannot be docked at more than one quay and a ship cannot be both waiting in the queue and be docked at the same time.

7.3.1 Differences between this case study and the CSP||B development

The case study in this thesis does not replicate the original case study in [220] in every detail but is based loosely upon it, because the aims of Contractual State Machines (CoSta) are different to those of CSP||B. CoSta is temporal logic and statechart specific with less emphasis on data abstraction. The CSP||B example uses *divergence-freedom* as a healthiness criterion, where computation preconditions are used to police invariant properties. Here, a symmetric approach is taken with the use of guarding conditions to police invariants, with deadlock-freedom as the corresponding system-wide healthiness criterion. In addition the CSP||B example does not have shared variables, but instead uses parameter passing. Data abstraction is not a primary driver for CoSta. Furthermore, the model checkers under development for CoSta will restrict data to finite concrete representations. The case study in the thesis therefore works with a simplified abstraction of the data.

For example, a single sequence is used to represent quays in the docking system and the sequence is modelled as a set of variables.

There are numerous reasons for the differences between the CSP||B example and this case study. An aim of the CSP||B example was integration of existing formal methods. CoSta and the research for this thesis focuses on support for a rigorous design process from temporal logic contracts to statecharts. There are differences in the languages; for example, CoSta's chief means of abstraction is through the use of temporal contracts. With CSP||B the focus is more towards abstract processes operating on abstract data representations. They both support communication and concurrency but different styles of process algebra synchronisations. CSP's is multiway whereas CoSta's parallel operator is a shared-variable version of CCS's two-way synchronising operator. The (shared-variable) CCS parallel operator was used in this case study, but at the top level, before the patterns apply. The refinement relation (SVRS) was designed to be compositional with respect to it.

One of the goals of the CSP||B study is to show that the global behaviour of the system upholds the "preconditions" stated within the concurrent components of the system. In [220] these preconditions were given as conditions outside which the component *diverges*. In comparison, CoSta contracts were not designed with divergence foremost in mind. They were designed to allow the dual interpretation of logical formula as labelled transition systems and it is not possible to describe divergence scenarios in the same way.

However, one can express the conditions under which the system *deadlocks*, and it is therefore possible to perform a related form of analysis based on deadlock freedom. This is achieved by stipulating deadlock freedom as a global property of the system, and then providing deadlock scenarios, rather than divergence scenarios, in the concurrent components of the system. Showing the system as a whole refines the deadlock freedom requirement is not quite the same as showing that the individual components do not deadlock, since one component might deadlock while the other continues to communicate with the environment. This may be perceived as a weakness in the contract language and a clear justification for completing the theory in respect of divergence. However, for the purposes of this case study, the weaker property will be sufficient.

7.4 Design of the docking system version 1

Two main versions of the case study were conducted; the first version does not consider the data (there are no variable or data updates), whereas the second version does. The first version (without data) is less complex and its main purpose is to illustrate the approach. The first version refines contracts into a CoSta state machine design using patterns all of which do not refer to data. This version is thus not contingent on patterns with complex side-condition checks. This is an advantage as some of the side-condition checks are not implemented in the model checker (e.g. STGA refinement or predicates involving quantifiers). This means it is possible with this version to road test the theory without relying on human inspection of side-conditions. This is important to the validation process.

This section presents the simplest version of the case study to illustrate the approach which refines an abstract open contract to a state machine design through pattern application. The example is based on the docking system which implements a service, notionally to allocate quays to ships as they enter a harbour. The ship requests a quay and the system responds with an available (i.e. empty) quay then the ship proceeds to dock. Other services include the vacating of a quay etc, although they are omitted in the following example. There is a single top-level requirement that each request must be met with a response. The following example concentrates solely on deadlock freedom and defers consideration of the data.

The case study follows the design strategy described in Chapter 2. The initial stage of the process is to devise the top-level contract which is based on closed reasoning, (the variables of the model are impervious to outside interference) and expresses model-wide properties. The main stage of the design is based on open reasoning and expresses properties of the sub-components of the model, where variables are subject to interference from the other sub-components. Closed contracts are expressed in a classical temporal logic style whereas open contracts are based on the operators described in Chapter 2. The basis for the development was the closed contract $\Box\langle\_\rangle$, which is included here for completeness. The closed contract says that an observable action must be enabled so it is not possible for the system to deadlock (as only internal actions can lead to deadlock).

Figure 7.1 shows the architecture of the system which is based on two parallel components. The first *ShipReq* acts as an interface component, communicating with the ships; the second *Quay* handles requests for quays, relayed from the first component and maintains the state of the system. The components are shown along with the direction in which

events are produced and consumed. The *dock* (respectively *docked*) event is produced (respectively consumed) outside the system.

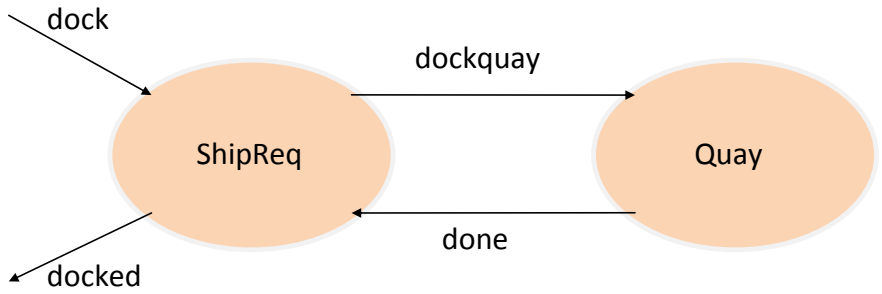

Figure 7.1: Architecture of system version 1

The open contracts for each component specify a minimal condition needed for the system as a whole to avoid deadlock and they describe the correct sequencing of the internally synchronising events.

For *ShipReq* we have three conjuncts:

$$A \quad (\overline{\langle' dockquay, done\rangle} \wedge \langle[-' dockquay, done]\rangle) \curvearrowright (\overline{\langle -' dockquay\rangle} \wedge \langle[' dockquay]\rangle)$$

$$B \quad \Box[' dockquay]((\overline{\langle' dockquay, done\rangle} \wedge \langle[-' dockquay, done]\rangle) \curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done]\rangle))$$

$$C \quad \Box[done]((\overline{\langle' dockquay, done\rangle} \wedge \langle[-' dockquay, done]\rangle) \curvearrowright (\overline{\langle -' dockquay\rangle} \wedge \langle[' dockquay]\rangle))$$

There are three ways deadlock can occur. The first is if both components deadlock. The second is if the components offer incompatible internal actions exclusively so that the internal events *dockquay* and *done* are not synchronised and thirdly if one component deadlocks and the other offers an internal action that can never proceed. The open contracts must prevent these scenarios from occurring. They need to ensure the correct sequencing of internal actions and make sure that in between those actions there is always an external action enabled so neither component can deadlock.

The first conjunct for the *ShipReq* component says that (from the start of operation) both internal synchronising events (*dockquay* and *done*) must remain disabled, and some other event (e.g *dock, docked*) must remain enabled, until the *dockquay* event is ready to be produced. The condition stipulates this must be an exclusive event, i.e all other events are

disabled. There is a complementary conjunct in the contract for *Quay* that says internal events (*dockquay* and *done*) are disabled until *dockquay* is ready to be consumed (which we omit here). The second conjunct says that after a *dockquay* event has been produced, the internal events are disabled with some external event enabled (e.g *docked*) until the *done* event is (exclusively) enabled. Again there is a complementary conjunct in the *Quay* open contract. The final conjunct stipulates what happens after the *done* event, i.e. that internal events (*dockquay* and *done*) are disabled and external events (e.g *docked*) are enabled until the *dockquay* event becomes (exclusively) enabled again.

The design of the *ShipReq* component is developed through successive refinement via the application of patterns to refine the open contract to a CoSta state machine and transform the component into a (scaled down but) functionally correct design which is guaranteed to satisfy its open contract. Contracts are refined by keeping the same patterns of interactions but reducing nondeterminism. The design steps detailed below are for part of the *ShipReq* component that handles requests for the docking of a ship in a quay. For this fragment of the design there are over one hundred refinement/refactoring steps each one requiring a pattern application. Therefore a summary only of the larger refinement process is described.

A high-level overview of the refinement process can be given. The first design step transforms the open contract above into a hierarchy of conjoined contracts; we call the conjuncts of the contract A, B and C, respectively. In Step 2 the contracts A and C are transformed into CoSta state machines, with the introduction of *dock* and *dockquay* transitions. The conjunction is eliminated in Step 3 for the state with the contract $A \wedge C$ and a redundant level of hierarchy is removed. These first three steps are depicted in figure 7.2, this diagram summarises the first part of the larger refinement process. Figures 7.2 and 7.3 show a high-level summary of the steps and do not give a complete picture of the whole refinement decomposition.

Step 4 extends the CoSta state machine with contract $A \wedge C$ by adding the *done* and *docked* transitions. We require the *docked* transition to loop back to the start state, the *Reroute* pattern can be applied as the *docked* transition's target state has inner contract $A \wedge C$, equal to the outer contract on the start state. The *Reroute* pattern removes the *docked* transition's target state and loops the transition back up to the start state. Step 5 transforms the contract B to a CoSta state machine with *dock*, *dockquay*, *done* and *docked* transitions. The *docked* transition is rerouted to the start state to implement a loop; this

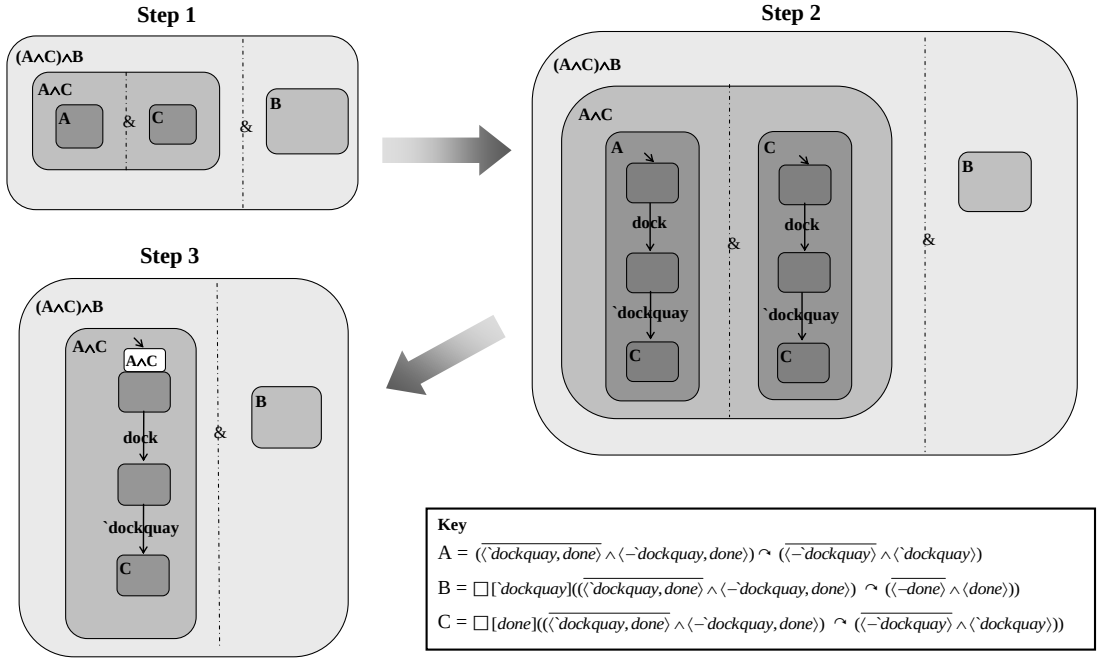

Figure 7.2: Summary of steps in the first stage of the refinement of *ShipReq*

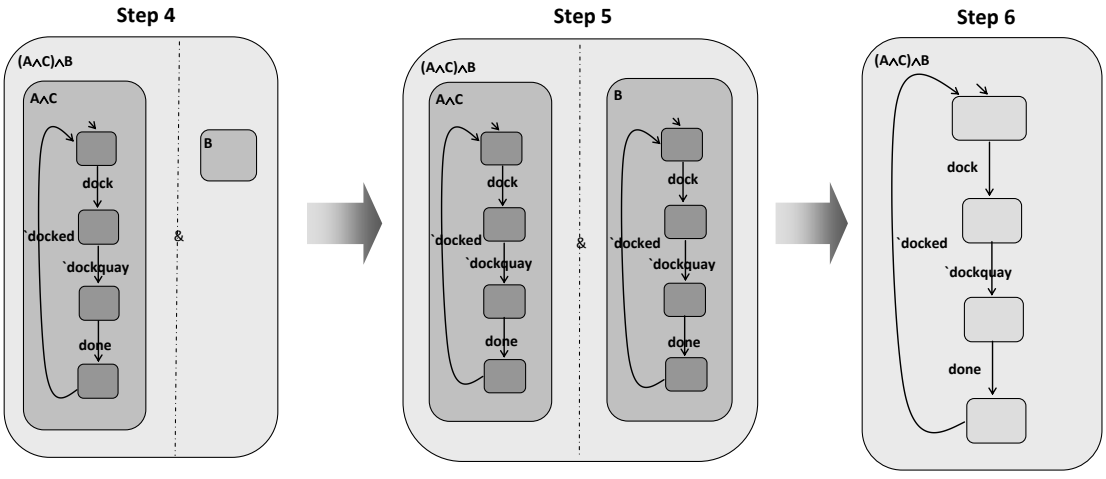

Figure 7.3: Summary of steps in the second stage of the refinement of *ShipReq*

step is complicated, and involves application of several patterns, i.e. *Unfold Always* and *Conjunction introduction*, as well as *If*. In Step 6 the conjunction is eliminated for the state with the contract $A \wedge C \wedge B$ and two levels of redundant hierarchy are removed. These last four steps are depicted in figure 7.3 and are a summary of the last part of the refinement process.

Step 5, which creates the CoSta state machine for the contract B, is now described in more detail. A summary of the refinement steps are depicted in figure 7.4.

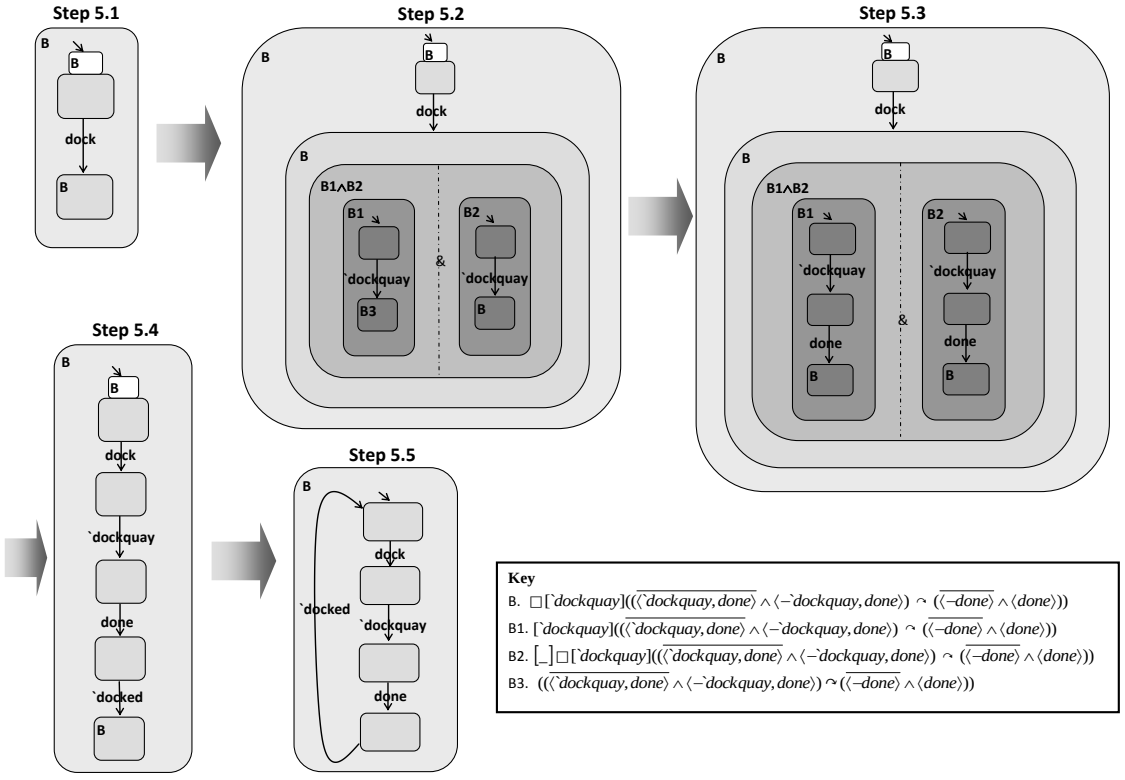

Figure 7.4: Summary of the refinement process for step 5

Step 5.1 shows the introduction of the *dock* transition and the patterns required are *Unfold Always* and *Conjunction introduction* followed by *If* applied to both of the new conjuncts. Then the application of the *Conjunction elimination* pattern to eliminate the conjunction of states is followed by applying the *Redundant hierarchy* pattern. The *Redundant hierarchy* pattern assigns the inner contract from the parent state to the outer contract of the start substate.

Step 5.2 is a refinement of the leaf state with contract B from Step 5.1. It introduces

the *dockquay* transition. The patterns applied are *Unfold Always* to refine contract B to $B1 \wedge B2$, then *Conjunction introduction* followed by *If* applied to both conjuncts. Before eliminating the conjunction, the leaf states with contracts $B3$ and B are further refined and *done* transitions are introduced, this is achieved in Step 5.3.

Step 5.3 refines the contract $B3$ to introduce the *done* transition. The patterns *Unless* and *Conjunction introduction* are applied first then the *TotEnable* pattern is applied to both conjuncts, followed by *Conjunction elimination* and *Redundant hierarchy*. To refine the leaf state with contract B for the conjunct with contract $B2$ in order to introduce the *done* transition, the patterns *Unfold Always* and *Conjunction introduction* are applied first, then the *If* pattern is applied to both of the new conjuncts. The application of the *Conjunction elimination* pattern is followed by *Redundant hierarchy*.

Step 5.4 applies *Conjunction elimination* and *Redundant hierarchy* to remove the conjunction. This step then introduces the *docked* transition. To achieve this *Unfold Always* and *Conjunction introduction* are applied, followed by the *If* pattern to both new conjuncts. The conjunction is eliminated and redundant hierarchy flattened with the application of patterns *Conjunction elimination* and *Redundant hierarchy*.

Step 5.5 applies the *Reroute* pattern to the *docked* transition to loop it back to the start state. The *Reroute* pattern can be applied to a leaf state with an inner contract where there exists another state with an associated outer contract that is syntactically equal to[1] the inner contract of the leaf state. The transformation removes the leaf state and its incoming transitions are redirected to the state with the associated outer contract.

7.5 Design of the docking system version 2

This section presents the second and more complex version of the case study for the docking system. The purpose of the second case study is to apply some of the data-extended refinement and refactoring patterns (described in Chapter 5) to a design that incorporates shared variables and data updates.

For the second version, the main challenges were to firstly extend the top-level contract with new requirements relating to the data. For example, an invariant is needed to prevent the double allocation of quays. The invariant ensures that a ship can only be docked at

[1] more generally it can be a check for semantic equivalence or a refinement check and requires a model-checking side-condition

a quay if it has not been docked already and the quay is empty. The next challenge was to specify new open contracts that respect the top-level contract. They are based on open reasoning and express properties of the sub-components of the model, where variables are subject to interference from the other sub-components. Once specification of the closed contract has been achieved, the overall aim is to express the open contracts, that still respect the closed contract and support the required behaviour of the docking system which implements a service to allocate quays to ships as they enter a harbour. If a ship requests a quay, the system responds with an available berth where the ship will be docked. The system has two parallel components, *ShipReq* acts as an interface component, communicating with the ships; and *Quay* that handles requests for quays, relayed from the first component and maintains the state of the system. The open contracts express properties of these parallel subcomponents and describe a sequential pattern of requesting a *dock* and then performing the internal events, *dockquay* and *done*. The scope is limited to docking a ship in a quay. No discussion of a ship leaving a quay or the port is given, and neither is the interleaving of more than one ship considered.

Once the open contracts had been devised, the final challenge was to then apply refinement and refactoring patterns to refine the open contracts to a Contractual State Machine design. There are two top-level requirements, one is that an invariant on the ships and quays is maintained (so as not to double allocate a ship to more than one quay), another is that each request must be met with a response. The model checkers under development for CoSta will restrict data to finite concrete representations. The case study therefore works with a simplified abstraction of the data. The variables of the system are s representing the ship to be docked and two quays q_1 and q_2.

The contracts for the second version of the case study are not scalable for large numbers of quays as they have been written specifically for a system with two quays only. For example, the invariant is expressed as a guard on the *dockquay* event, the quays are limited to two and they are specifically referred to as q_1 and q_2 in the contracts to enforce the invariant. A more general version of the invariant would be scalable. Being succinct for larger structures requires either a quantifier or since we are restricted to finite data structures, an indexed conjunction/disjunction. The SMT-solver used by the model checker does not support quantifiers however, so using quantifiers in the modelling language would produce predicates that the model checker could not deal with. In the case of the simpler case study (version 1) this is not a problem. Longer term it is expected that

the model checker will provide support for indexed conjunctions/disjunctions as a finite alternative to quantifiers. Once in place the modelling language could be augmented to take advantage of the additional constructs.

The top-level contract is:

$$\Box\langle\_\rangle \wedge \Box\overline{\langle\_[q_1, q_2 : (q_1 = q_2 \Rightarrow q_1 = 0) \Rightarrow \neg(q_1' = q_2' \Rightarrow q_1' = 0)]\rangle}$$

The closed contract says that an observable action must be enabled so it is not possible for the system to deadlock (as only internal actions can lead to deadlock) and that any action must be disabled that would result in a ship being docked in more than one quay at the same time.

Figure 7.5 shows the top-level architecture of the system updated to include variables. The architecture is based on two parallel components. The events shown are either those produced/consumed outside the system or the internal synchronising actions between the two parallel components, (for example the *click* event is of no significance to any other component and is purely part of the Quay component's internal processing so it does not appear in figure 7.5 for the top-level architecture). The *dock* events update the variable s to contain the ship that needs to be docked. The *docked* event is consumed outside the system and is produced by the *ShipReq* component once a ship has been docked in a quay. The *dockquay* event signals to the *Quay* component that a ship needs to be docked (which is not currently docked) and there is space available in the quays. The *done* event is produced by the *Quay* component once a ship has been docked in a quay. CoSta's shared variable environment allows all variables, including the variable s for example, that represents the ship to be docked, to be accessible by both the *ShipReq* and *Quay* components and it does not need to be passed as a parameter on any of their actions.

In order to give the parallel operator a simple semantics, updates on synchronising actions were syntactically restricted to *skip*. The open contracts are defined so as to abstract the design and guarantee the closed contract. For each component, (*ShipReq, Quay*) the open contracts must sequence the synchronising actions to prevent deadlock, with minimal restrictions on how the data is updated, ensuring that synchronising actions *skip* and the invariant is maintained. It is necessary to have state in order to express the open contract, this point was initially made in Section 2.13.

When the full system (with data) was considered, one of the challenges for the data-extended version of the case study was to specify open contracts that preserve the invariant and prevent deadlock. The design had to satisfy additional requirements on the data and

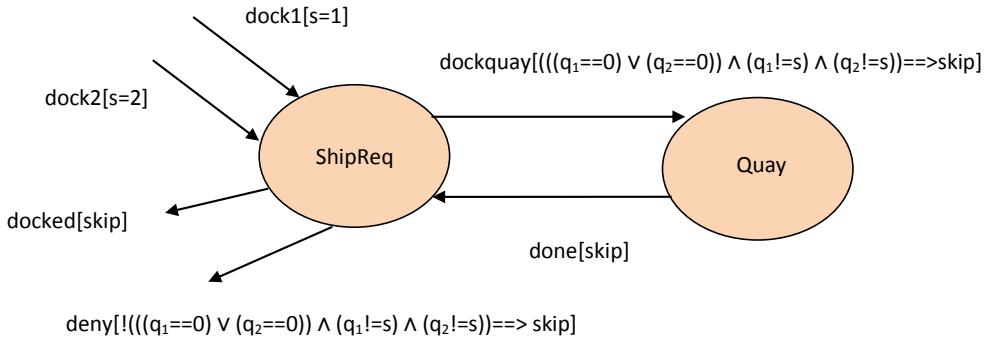

Figure 7.5: Architecture of system version 2

these requirements needed to be specified as properties in the contracts and preserved by the development [2]. Uncontrolled updates to the data will not satisfy the top-level properties. The solution is to specify more precisely in the open contracts which actions update data and what update is required. Simply adding a conjunct to the open contracts of both components (*ShipReq, Quay*) to disable any action which breaks the invariant does not guarantee deadlock freedom when the components are composed. For example, an update to the quays could be allowed to happen more than once, the first update occurring at the wrong time which nevertheless does not break the invariant but creates the problem that when the update to the quays occurs at the right time, a ship is already docked and thus breaks the invariant and so the system deadlocks. This problem is solved by sequencing the update to the quays at the correct time and insisting that at other times all actions *skip*.

7.5.1 Open contract for the ShipReq component

This section discusses the open contract for the *ShipReq* component. The *ShipReq* process accepts a request from an approaching ship to dock, *dock* and sends back a response, *docked* (indicating success when the precondition is *True*) or *deny* (if the precondition of *dock* is *False*). The precondition is that there is a vacant quay and the ship is not already

[2] It would be interesting future work to determine whether the two versions of the case study could be linked by refinement, i.e. whether it would be possible to start with the dataless contract and strengthen it to include data during the design process.

docked. If the precondition is *True*, the *ShipReq* process synchronises with *Quay*.

The approach adopted for specifying the open contracts is to separate out the behavioural conjuncts and the data conjuncts where possible, i.e where actions behave consistently with respect to the data throughout their behaviour. For example in the *ShipReq* contract all conjuncts apart from the final one are quite loose about updating the data this allows them to focus on the properties of interest. The final conjunct however constrains the contract further to prevent the quays being updated.

The invariant for the *ShipReq* component is: $(q_1 = q_2) \Rightarrow (q_1 = 0)$.

The open contract for *ShipReq* has five conjuncts, these will be described in turn.

$R \stackrel{def}{=} (q_1 = 0 \vee q_2 = 0) \wedge (q_1! = s \wedge q_2! = s)$

R ensures there is an empty quay available and prevents the double allocation of ships to quays (preserves the invariant). In order to maintain the invariant stipulated in the open contracts, any update from any action has to preserve the invariant so that all actions are disabled where the invariant is not respected.

The system is based on two parallel components, *ShipReq* and *Quay*. *ShipReq* is the interface component, communicating with the ships. *Quay* handles requests for quays relayed from *ShipReq* and maintains the state of the system. The *ShipReq* component has the following actions:

- *dock* is a request from an approaching ship to dock in a quay. It updates the ship variable s with the identifier of the ship requesting to dock. In the open contract presented below it is enabled from the start of operation (it could occur multiple times at this point to request for different ships to dock and each time the ship variable s would be overwritten with the ship's identifier for the latest request) until the internal synchronising event *'dockquay* and the action to indicate failure *'deny* are enabled. If the same ship tries to dock twice the value of the variable s representing the ship to dock will remain unchanged. The *dock* event is enabled again after a *'deny* or a *done* action has occurred. It cannot occur between *'dockquay* and *done* in the form where it updates the ship variable s with the identifier of a new ship to be docked, as it is not permitted to update the ship variable s with a different value until the *done* event has occurred and the original ship has been docked. The *dock* action could occur between *'dockquay* and *done* to dock the same ship twice but the value of the variable s representing the ship to dock would remain unchanged so there would be no adverse effect on the behaviour of the system. The aim for

the open contracts is to specify the minimal restrictions needed to maintain the invariant and prevent the system as a whole from deadlocking. The open contract could be strengthened further however, to reduce the availability of the *dock* event, and outlaw the same ship being able to request a dock twice.

- *'dockquay* is the request to dock a ship relayed to the *Quay* component. It is an internal synchronising action to enable *ShipReq* to synchronise with the *Quay* component.

- *'deny* is the response from the *ShipReq* component indicating failure.

- *done* is an internal synchronising event to enable *Quay* to synchronise with the *ShipReq* component. It indicates that a ship has been docked by the *Quay* component.

- *'docked* is the response from the *ShipReq* component indicating success.

The first conjunct is:

$$((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \\ \curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[:R]]\rangle \wedge \\ \overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge \\ \overline{\langle' deny[:R]\rangle}) \wedge$$

The first conjunct for the *ShipReq* component says that (from the start of operation) both internal synchronising events (*dockquay* and *done*) and the *deny* action must remain disabled, and some other event (e.g *dock, docked*) must remain enabled, until the *dockquay* or *deny* events are ready to be produced. The condition stipulates this must be an exclusive event, i.e all other events are disabled. Throughout the *ShipReq* open contract whenever the *dockquay* or *deny* actions appear, the synchronising action *dockquay* must *skip* and is enabled from every before state within the guard of R, and disabled from every before state within the guard of $\neg R$. The *deny* event is the complement of this and enabled from every before state within the guard of $\neg R$ and disabled from every before state within the guard of R.

The open contract for the *ShipReq* component ensures that it only synchs with *Quay* in the situation where the invariant will be preserved. There is a complementary conjunct in the contract for *Quay*, (which is presented later) with respect to the synchronising actions

that says internal events (*dockquay* and *done*) are disabled until *dockquay* is ready to be consumed.

The second conjunct is:

$$\Box[{}'dockquay](\overline{\langle {}'dockquay,{}' deny, done\rangle}\wedge\langle[-{}'dockquay,{}' deny, done]\rangle)\wedge\overline{\langle \_[q_1,q_2,s : s'! = s]\rangle}$$
$$\curvearrowright (\overline{\langle -done\rangle}\wedge\langle[done[skip]]\rangle\wedge\overline{\langle done[\neg skip]\rangle}))\wedge$$

The second conjunct says that after a *dockquay* event has been produced, the internal synchronising events (*dockquay* and *done*) and the *deny* action are disabled with some external event enabled (e.g *docked*). Additionally all events are disabled that update s until the *done* event is (exclusively) enabled. The *done* event is enabled with an update consistent with *skip* and disabled otherwise. Again there is a complementary conjunct for the synchronising actions in the *Quay* open contract.

The open contract for the *ShipReq* component must ensure that the synchronising actions *skip* and the *dockquay* event is only enabled if one of the quays is empty and s is not already docked. The *ShipReq* component must also ensure that actions preserve the invariant between the *dockquay* and *done* events and do not alter the quays or s in a way that will force the system as a whole not to satisfy the top-level properties.

If s is put in the quays or the quays are filled this will not necessarily break the invariant directly but will force the update in the *Quay* component to behave in a way that does not satisfy the top-level property (either because it deadlocks as there is no space in the quays or it breaks the invariant if s is already in a quay). This problem is solved by insisting that all actions *skip* between *dockquay* and *done*.

The third conjunct is:

$$\Box[done](\overline{\langle {}'dockquay,{}' deny, done\rangle}\wedge\langle[-{}'dockquay,{}' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -{}'dockquay,{}' deny\rangle}\wedge\langle[{}'dockquay[:R]]\rangle\wedge$$
$$\overline{\langle {}'dockquay[q_1,q_2,s : \neg(R\wedge skip)]\rangle}\wedge\langle[{}'deny[q_1,q_2,s : \neg R]]\rangle\wedge$$
$$\overline{\langle {}'deny[:R]\rangle}))\wedge$$

The third conjunct stipulates what happens after the *done* event, i.e. that internal events (*dockquay* and *done*) and the *deny* action are disabled and external events (e.g *docked*) are enabled until the *dockquay* and *deny* events become (exclusively) enabled again.

The fourth conjunct is:

$$\Box[{}'deny](\overline{\langle {}'dockquay,{}' deny, done\rangle}\wedge\langle[-{}'dockquay,{}' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -{}'dockquay,{}' deny\rangle}\wedge\langle[{}'dockquay[:R]]\rangle\wedge$$
$$\overline{\langle {}'dockquay[q_1,q_2,s : \neg(R\wedge skip)]\rangle}\wedge\langle[{}'deny[q_1,q_2,s : \neg R]]\rangle\wedge$$

$$\overline{\langle' deny[:R]\rangle}))\wedge$$

The fourth conjunct stipulates what happens after the *deny* event, i.e. that internal events (*dockquay* and *done*) and the *deny* action are disabled and external events (e.g *docked*) are enabled until the *dockquay* and *deny* events become (exclusively) enabled again.

The fifth conjunct is:

$$\Box\overline{\langle_{-}[q_1, q_2, s : (q_1'! = q_1) \vee (q_2'! = q_2)]\rangle}$$

The fifth conjunct stipulates that any action is disabled that updates the quays. It is important to ensure either that conjuncts are mutually exclusive and at no time will more than one of them hold; or if it is possible for more than one conjunct to hold at the same time, to ensure their combined subsequent behaviour is what is required. For example if the fourth and second conjuncts were permitted to hold simultaneously the subsequent behaviour could only be refined into something which never again offered a synchronising action (*dockquay, done*) since to do so would either contradict the second conjunct or contradict the fourth conjunct because they offer the possibility of different internal actions.

The *dockquay* and *done* actions are carefully sequenced and the conjuncts of the *ShipReq* contract are mutually exclusive so that not more than one of them will hold at a time given that a *dockquay* or *done* event is enabled.

The complete open contract for the *ShipReq* component is:

$$R \stackrel{def}{=} (q_1 = 0 \vee q_2 = 0) \wedge (q_1! = s \wedge q_2! = s)$$

$$((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[:R]]\rangle\wedge$$
$$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle\wedge$$
$$\overline{\langle' deny[:R]\rangle}))\wedge$$

$$\Box[' dockquay]((\overline{\langle' dockquay,' deny, done\rangle}\wedge\langle[-' dockquay,' deny, done]\rangle)\wedge\overline{\langle_{-}[q_1, q_2, s : s'! = s]\rangle}$$
$$\curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))\wedge$$

$$\Box[done]((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[:R]]\rangle\wedge$$
$$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle\wedge$$
$$\overline{\langle' deny[:R]\rangle}))\wedge$$

$$\Box[' deny]((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[:R]]\rangle\wedge$$

$$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle' deny[: R]\rangle})) \wedge$$
$$\Box\overline{\langle\_[q_1, q_2, s : (q_1' ! = q_1) \vee (q_2' ! = q_2)]\rangle}$$

7.5.2 Open contract for the Quay component

This section discusses the open contract for the *Quay* component. The *Quay* process assigns a ship to a quay. The precondition is that there is a vacant quay and the ship is not already docked.[3] It waits for synchs from *ShipReq* (*dockquay*); when received the *click* event docks a ship, and the ship is stored in the quay variable. There are no hidden actions in the modelling language, so a "dummy" visible event (*click*) is used to accompany the internal change of state. It could be hidden (renamed *tau*) at the architectural level[4]. Finally the *Quay* process synchronises with *ShipReq* on the *done* event. Another service is to vacate a quay; the *Quay* component accepts a request from a docked ship to vacate the quay. The precondition is that the ship is assigned to the quay. If the precondition holds the quay is set to *zero* (meaning empty). If the precondition is *False*, "error" is returned. This service is omitted in the following example to simplify it further.

As described above the system is based on two parallel components, *ShipReq* and *Quay*. *ShipReq* is the interface component, communicating with the ships. *Quay* handles requests for quays relayed from *ShipReq* and maintains the state of the system. The *Quay* component has the following actions:

- *dockquay* is the request to dock a ship relayed from the *ShipReq* component. It is an internal synchronising action to enable *ShipReq* to synchronise with the *Quay* component.
- *'done* is an internal synchronising event to enable *Quay* to synchronise with the *ShipReq* component. It indicates that a ship has been docked by the *Quay* component.
- *click* docks a ship, the ship is stored in the quay variable. In the open contracts

[3]Preconditions are specified as guards. The process deadlocks if the precondition is not *True*, and the aim is to show that the system as a whole does not deadlock.

[4]In CoSta, *tau* serves a special purpose in the theory of duals and therefore the refinement ordering (it acts as disjunction). The refinement ordering does not at present allow *taus* to carry data updates as this would interfere with its special status.

presented below there are several occassions when some other action apart from *dockquay*, *'done* or *click* is enabled. For example, at the start of operation until the *dockquay* event is enabled or between the *dockquay* and *click* events. The *click* event however is the only action that is permitted to update the data.

Two versions of the open contracts for the *Quay* component are presented. The first version for *Quay* rules out updates of any kind by any action other than *click*. Although this achieves the objective to carefully control when the quays can be updated and the kinds of data updates permitted, it could be made less restrictive. The second version of the open contracts for the *Quay* component improves on the first by relaxing the restrictions on updates made by actions other than *click*. It permits updates weaker than *skip* on actions other than *click*. The reason for presenting the two versions is to exemplify the kind of validatory reasoning involved in constructing the open contracts.

The invariant for the *Quay* component is: $(q_1 = q_2) \Rightarrow (q_1 = 0)$.

$M \stackrel{def}{=} (q_1 = 0) \vee (q_2 = 0)$, M ensures there is an empty quay.

$N \stackrel{def}{=} ((q_1 = 0) \wedge (q_1' = s)) \vee ((q_2 = 0 \wedge (q_2' = s)) \wedge (q_1'! = q_2')$, N allocates s to the empty quay.

The open contract for *Quay* has six conjuncts, these will be described in turn.

The first conjunct:

$$((\overline{\langle dockquay,' done, click\rangle} \wedge \langle[-dockquay,' done, click]\rangle) \curvearrowright (\overline{\langle -dockquay\rangle} \wedge \langle[dockquay]\rangle))\wedge$$

The first conjunct for the *Quay* component says that (from the start of operation) both internal synchronising events (*dockquay* and *done*) and the *click* action must remain disabled, and some other event (e.g *dock*) must remain enabled, until the *dockquay* event is ready to be produced. The condition stipulates this must be an exclusive event, i.e all other events are disabled.

The second conjunct:

$$\Box[dockquay]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle[-'done, dockquay, click]\rangle) \curvearrowright (\overline{\langle -click\rangle} \wedge \langle[click[q_1, q_2, s : M]]\rangle))\wedge$$

The second conjunct says that after a *dockquay* event has been produced, the internal events (*dockquay* and *done*) and the *click* action are disabled with some external event enabled (e.g *docked*) until the *click* event is (exclusively) enabled. The *click* event is enabled from every before state within the guard of M which ensures there is an empty

quay.

Before the *Quay* component allocates a ship to a quay there must be an available quay and no ship equal to s already docked as the first situation would cause deadlock and the second would break the invariant. The open contract for the *Quay* component must introduce an additional event *click* to carry out the state update as the synchronising actions must *skip*. The *click* event is sequenced between *dockquay* and *done* and disabled at any other time. All other actions *skip* or weaker (*skip* on s and preserve the invariant).

The *Quay* design on its own can deadlock, the way it interacts with the *ShipReq* component however prevents this occurring. *ShipReq's* sequencing ensures that it only synchronises with *Quay* in the situation where the invariant is preserved and a ship is not already docked. Therefore it is only necessary in the *Quay* component to check that there is an available berth before the update to allocate a ship to a quay, between the synchronising actions.

The updates need to be sequenced so that until the ship has been allocated to a quay any other updates to the quays or s must wait. If the updates are not sequenced the contract would allow a ship to be docked or the quays filled, which may not necessarily violate the invariant directly but force the *click* update to behave in a way that does not satisfy the top level property either because it deadlocks if M is *False* or because it breaks the invariant because s is already in a quay. The open contract for *Quay* must ensure that it cannot update q_1 or q_2 during the sequencing of the actions except in one place in between *dockquay* and *done*.

The third conjunct:

$$\Box[click]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle[-'done, dockquay, click]\rangle) \curvearrowright (\overline{\langle -'done\rangle} \wedge \langle['done]\rangle))\wedge$$

The third conjunct for the *Quay* component says that after a *click* event has been produced, both internal synchronising events (*dockquay* and *done*) and the *click* action must remain disabled, and some other event (e.g *dock*) must remain enabled, until the *done* event is ready to be produced. The condition stipulates this must be an exclusive event, i.e all other events are disabled.

The fourth conjunct:

$$\Box['done]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle[-'done, dockquay, click]\rangle) \curvearrowright (\overline{\langle -dockquay\rangle} \wedge \langle[dockquay]\rangle))\wedge$$

The fourth conjunct stipulates what happens after the *done* event, i.e. that internal events

(*dockquay* and *done*) and the *click* action are disabled and external events (e.g *docked*) are enabled until the *dockquay* event becomes (exclusively) enabled again.

The fifth conjunct:

$\Box\overline{\langle -click[q_1, q_2, s : \neg skip]\rangle}\wedge$

The fifth conjunct, in addition, stipulates that actions other than *click* must *skip*.

The sixth conjunct:

$\Box\overline{\langle click[q_1, q_2, s : \neg N]\rangle}$

The sixth conjunct ensures that at all times the *click* action is disabled from every before state within the guard of $\neg N$.

The complete open contract for the *Quay* component is:

$((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-dockquay,' done, click]\rangle)$

$\curvearrowright (\overline{\langle -dockquay\rangle} \wedge \langle [dockquay]\rangle))\wedge$

$\Box[dockquay]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$\curvearrowright (\overline{\langle -click\rangle} \wedge \langle [click[q_1, q_2, s : M]]\rangle))\wedge$

$\Box[click]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$\curvearrowright (\overline{\langle -'done\rangle} \wedge \langle ['done]\rangle))\wedge$

$\Box['done]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$\curvearrowright (\overline{\langle -dockquay\rangle} \wedge \langle [dockquay]\rangle))\wedge$

$\Box\overline{\langle -click[q_1, q_2, s : \neg skip]\rangle}\wedge$

$\Box\overline{\langle click[q_1, q_2, s : \neg N]\rangle}$

The next version of the contract for the *Quay* component permits updates weaker than *skip* on actions other than *click* but insists that actions other than *click* do not update s, do not insert s into a quay, preserve the invariant and do not fill the quays.

$((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-dockquay,' done, click]\rangle)$

$\curvearrowright (\overline{\langle -dockquay\rangle} \wedge \langle [dockquay[skip]]\rangle\wedge$

$\overline{\langle dockquay[\neg skip]\rangle}))\wedge$

$\Box[dockquay]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$\curvearrowright (\overline{\langle -click\rangle} \wedge \langle [click[q_1, q_2, s : M]]\rangle))\wedge$

$\Box[click]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$\curvearrowright (\overline{\langle -'done\rangle} \wedge \langle ['done[skip]]\rangle \wedge \overline{\langle 'done[\neg skip]\rangle}))\wedge$

$\Box['done]((\overline{\langle dockquay,' done, click\rangle} \wedge \langle [-'done, dockquay, click]\rangle)$

$$\curvearrowright (\overline{\langle -dockquay \rangle} \wedge \langle [dockquay[skip]] \rangle \wedge \overline{\langle dockquay[\neg skip] \rangle}))\wedge$$

$\Box\overline{\langle -click[q_1, q_2, s : (q_1^{'}! = 0) \wedge (q_2^{'}! = 0) \vee}$

$\overline{(q_1^{'} = s) \vee (q_2^{'} = s) \vee (s'! = s) \vee (q_1^{'} = q_2^{'}) \wedge (q_2^{'}! = 0)]\rangle}\wedge$

$\Box\overline{\langle click[q_1, q_2, s : \neg N] \rangle}$

The N update could be strengthened to check that a ship is not already docked.

$N_2 \stackrel{def}{=} ((q_1 = 0) \wedge (q_2! = s) \wedge (q_1^{'} = s)) \vee ((q_2 = 0) \wedge (q_1! = s) \wedge (q_2^{'} = s))\wedge$

$(q_1^{'}! = q_2^{'}) \wedge (s' = s)]$

This is probably unnecessarily strong as this constraint is checked by the *ShipReq* component just before it synchronises with *Quay*.

7.5.3 Refinement of the ShipReq component

This design of the *ShipReq* component is developed through successive refinement via the application of patterns to refine the open contract to a CoSta state machine which is guaranteed to satisfy it. The design steps detailed below are for the *ShipReq* component that handles requests for the docking of a ship in a quay. The following patterns from Chapters 4 and 5 are applied:

- Conjunction Introduction.
- Unfold Unless.
- Disable.
- Combine States.
- Totalised Enable.
- Conjunction Elimination.
- Redundant Hierarchy.
- Move Target Down.
- Unfold Always.
- Commutativity.
- If.
- Strengthen Contract.

- Reroute.

This section demonstrates the first stage of the refinement process, the full refinement is given in Appendix C.

The contract for the *ShipReq* component has five conjuncts $A \wedge B \wedge C \wedge D \wedge E$ (Conjunction is left-associative).

$R \stackrel{def}{=} (q_1 = 0 \vee q_2 = 0) \wedge (q_1! = s \wedge q_2! = s)$

$$A = ((\overline{\langle\, 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle ['dockquay[:R]]\rangle \wedge \overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle ['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle 'deny[:R]\rangle}))$$

$$B = \Box[done]((\overline{\langle\, 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle ['dockquay[:R]]\rangle \wedge \overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle ['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle 'deny[:R]\rangle}))$$

$$C = \Box['deny]((\overline{\langle\, 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle ['dockquay[:R]]\rangle \wedge \overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle ['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle 'deny[:R]\rangle}))$$

$$D = \Box['dockquay]((\overline{\langle\, 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[q_1, q_2, s : s'! = s]\rangle} \curvearrowright (\overline{\langle -done\rangle} \wedge \langle [done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$$

$$E = \Box\overline{\langle \_[q_1, q_2, s : (q_1'! = q_1) \vee (q_2'! = q_2)]\rangle}$$

The model checker (HST) supports evaluation of quantifier-free conjectures on data. Patterns with more complex side-conditions (e.g *TotEnable's* side-condition may require evaluation of a conjecture with quantifiers) rely on inspection for verification as the functionality required to implement these patterns fully was not available via the model checker. Though when the model checker (HST) is extended, the approach in this thesis will naturally extend.

7.5.3.1 Refinement stage 1

The overall high-level process of stage 1 refines the conjunct with contract A to introduce the *dock1* and *dock2* transitions.

1. Apply the *Conjunction introduction* pattern four times.

The *Conjunction introduction* pattern is applicable if the selected component is a state whose inner contract is an *And* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *And* ψ". The pattern introduces two new substates, one with inner contract "ϕ" and the other with inner contract "ψ". The contract is: $A \wedge B \wedge C \wedge D \wedge E$, *Conjunction introduction* is applied for the first time, two new states are created one with contract $\phi = A \wedge B \wedge C \wedge D$ the other with contract $\psi = E$. *Conjunction introduction* is then applied for the second time to the state with contract $A \wedge B \wedge C \wedge D$. Two new states are introduced one with contract $\phi = A \wedge B \wedge C$ and the other with contract $\psi = D$.

Conjunction introduction is applied for the third time to the state with contract $A \wedge B \wedge C$. Two new states are introduced one with contract $\phi = A \wedge B$ the other with contract $\psi = C$. *Conjunction introduction* is finally applied for the fourth time to the state with contract $A \wedge B$. Two new states are introduced one with contract $\phi = A$ the other with contract $\psi = B$ (see Figure 7.6).

Figure 7.6: Conjunction introduction pattern applied four times

2. Refine the state with contract A.

$R \stackrel{def}{=} (q_1 = 0 \vee q_2 = 0) \wedge (q_1! = s \wedge q_2! = s)$

$A = ((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$
$\curvearrowright (\overline{\langle-' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$
$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle' deny[: R]\rangle}))$

Apply the *Unfold Unless* pattern. The *Unfold Unless* pattern is applicable if the selected component is a state whose inner contract is an *Unless* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *Unless* ψ" and the pattern unfolds the contract to "ϕ *And (If* $\_$ $(\phi$ *Unless* $\psi))$".

ϕ is $(\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$

ψ is $(\overline{\langle-' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$
$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle' deny[: R]\rangle})$

$A1 = \phi$

$A2 =$ *(If* $\_$ $(\phi$ *Unless* $\psi))$.

The pattern unfolds the contract A to $A1 \wedge A2$

$A1 = \overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle$

$A2 = [\_]((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$
$\curvearrowright (\overline{\langle-' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$
$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle' deny[: R]\rangle}))$

3. Apply the *Conjunction introduction* pattern to the state with contract $A1 \wedge A2$. Two new states are introduced, one with contract $A1$ and the other with contract $A2$.

4. Apply the *Conjunction introduction* pattern to state with contract $A1$ (see Figure 7.7.

The contract $A1 = A9 \wedge A10$. The pattern creates two new substates one with contract $A9$ and the other with contract $A10$.

$A1 = \overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle$

$A9 = \overline{\langle' dockquay,' deny, done\rangle}$

$A10 = \langle[-' dockquay,' deny, done]\rangle$

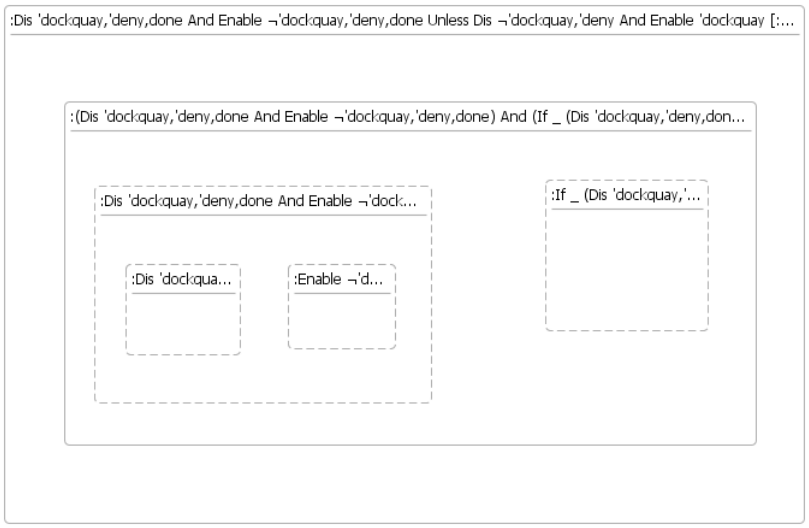

Figure 7.7: Conjunction introduction pattern applied to contract A1

5. Apply the *Disable* pattern to the state with contract $A9$ to introduce two new transitions *dock1* and *dock2*. The *Disable* operator specifies that none of the events in a set are available. When accompanied by a variable update the action is disabled for all the updates consistent with the expression. It may still be enabled within its guard as long as its update is not consistent. The *Disable* operator contract $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed as an equivalent *If* operator expression $[a_1, .., a_n[x : P]]False$, i.e *If* one of the events $a_1, .., a_n[x : P]$ is enabled then subsequently behave as *False*.

The *If* pattern is applied it does not permit the introduction of transitions with subsequent behaviour *False* so in this instance transitions are introduced with subsequent behaviour *True*. The pattern checks that the action is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, the model checker (HST) is used to verify that the update expression for a transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. the update expressions are disjoint and there is no overlap. The model checker (HST) is invoked to decide if $\neg(updateExpr2 \wedge updateExpr1)$ is tautological.

$A9 = \overline{\langle' dockquay,' deny, done\rangle}$

The *Disable* pattern is applied, $A9$ is rewritten as an *If* operator expression

$A9 = [' dockquay,' deny, done]False$.

The two new transitions required are $dock1[s = 1]$ and $dock2[s = 2]$.

The pattern checks firstly whether the transition event is in the contract *eventlist* and if it

is then a further check is conducted invoking the model checker (HST) to ensure that the transition update expression and the contract update expression are disjoint and there is no overlap. The first transition is $dock1[s = 1]$ whose event is not included in the contract *eventlist* so the transition is added with subsequent behaviour *True* and no further check is required. Similarly the second transition is $dock2[s = 2]$ whose event is not included in the contract *eventlist* so the transition is added with subsequent behaviour *True* and no further check is required (see Figure 7.8).

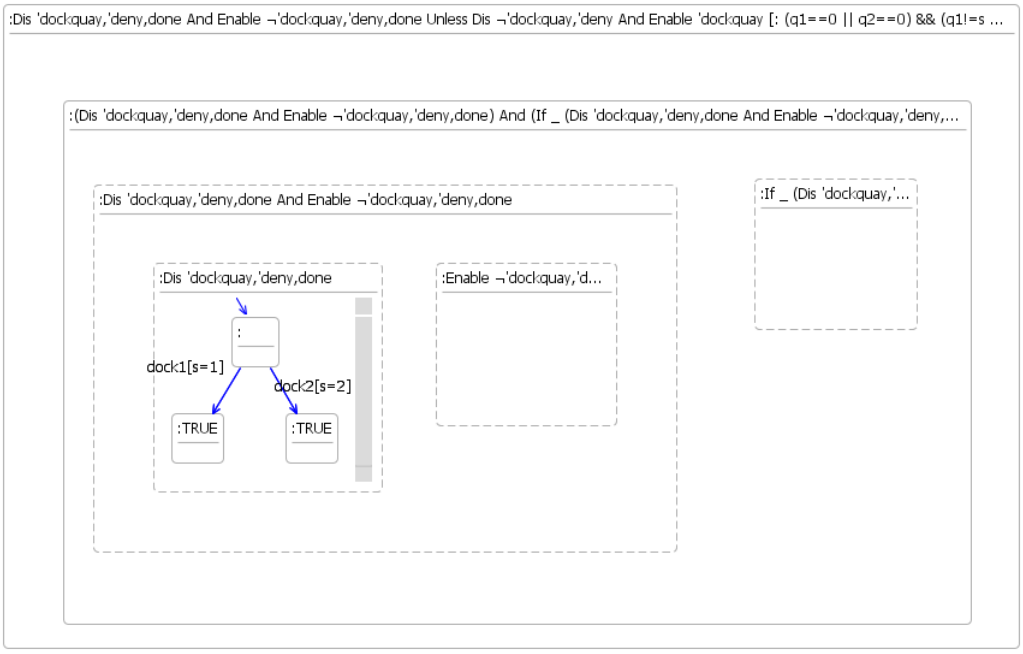

Figure 7.8: Disable pattern applied to contract A9

6. Apply the *Combine states* pattern to combine the substates of $A9$ with the inner contract *True*. The *Combine states* pattern is applicable to two states. The pattern checks that the states to be combined have the same parent state and no substates or outgoing transitions. It also checks that their inner contracts are syntactically equivalent. The inner contract is *True* for both states (see Figure 7.9).

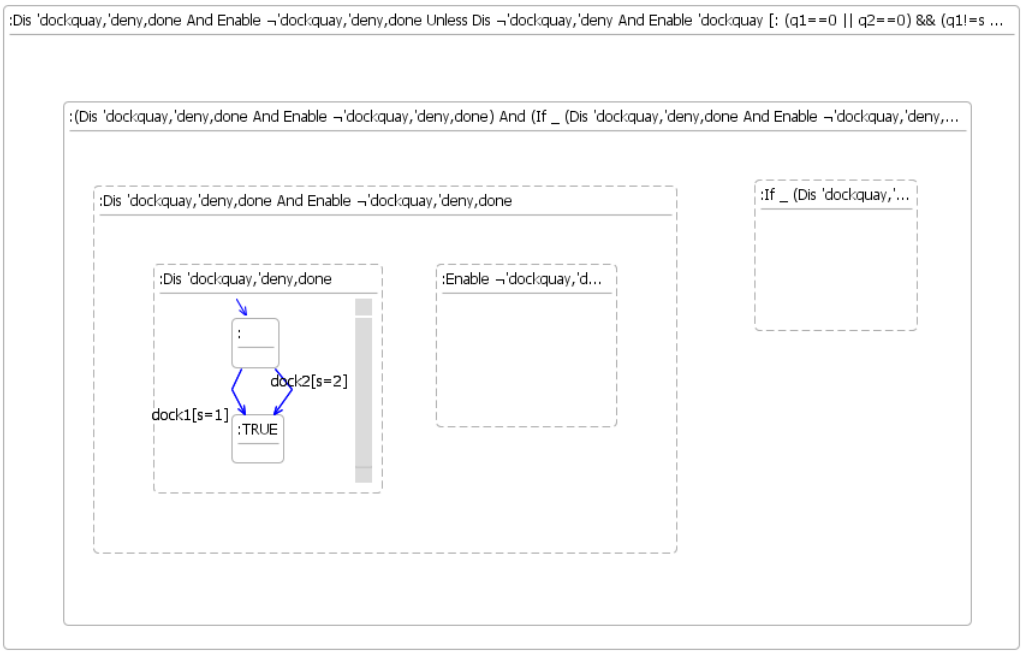

Figure 7.9: Combine states pattern applied to state with contract A9

7. Apply the *TotEnable* pattern to the state with contract $A10$ to introduce two new transitions *dock1* and *dock2*. The *TotEnable* pattern refines a *TotEnable* contract to a state machine model. The *TotEnable* pattern is applicable if the selected component is a state. The pattern checks that the state's inner contract has the form *TotEnable eventFormula* or *TotEnable eventFormula [updateExpression]*. *EventFormula* can be a set of events, a negated set of events or the underscore character. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract.

Checks based on syntax are performed initially. To determine if the transition satisfies the contract the transition event must be included in the contract's *eventlist*. If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore transitions with any event are possible.

If the contract is of the form $\langle[eventlist]\rangle$ it is satisfied by transitions of the form *event*, *event[assignmentlist]*, *event[skip]*. If the contract is of the form $\langle[eventlist[G]]\rangle$, where G is a guard and refers only to *before* values for the variables, it is satisfied by transitions of the form $event[G]$, $event[G \Longrightarrow assignmentlist]$, $event[G \Longrightarrow skip]$ (it is a permitted refinement to strengthen the update within the guard). If the contract has

the form $\langle[eventlist[skip]]\rangle$ or $\langle[eventlist[G \Longrightarrow skip]]\rangle$ it is satisfied by transitions of the form *event*, $event[skip]$. Otherwise a more detailed check is performed to determine if the transition satisfies the contract. To ensure that the update expressions are consistent the following check is required.

The side-condition is $\vdash ((\exists x' : P) \Rightarrow (\exists x' : P')) \wedge ((\exists x' : P) \wedge P' \Rightarrow P)$.

It permits the guard to be weakened and the update strengthened within the guard. Currently this check is verified by inspection as it requires the evaluation of a conjecture with quantifiers on data and this functionality is not yet supported by the model checker (HST). The current refinement step (as well as all other refinement steps in this case study that apply the *TotEnable* pattern) does not need the more detailed check to be performed as the simpler checks are sufficient to determine that the transition satisfies the contract (as shown below).

The contract is $A10 = \langle[-'dockquay,' deny, done]\rangle$

The two new transitions required are $dock1[s = 1]$ and $dock2[s = 2]$

The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. Checks based on syntax are performed initially. If the transition event is not in the contract's negated list of events and if the contract is of the form $\langle[eventlist]\rangle$ it is satisfied by a transition of the form *event[assignmentlist]*. For the first transition $dock1[s = 1]$, *dock1* is not in the negated list of events for the contract $\langle[-'dockquay,' deny, done]\rangle$ also the contract is of the form $\langle[eventlist]\rangle$ and the transition is of the form, *event[assignmentlist]*. Therefore it is concluded that the transition satisfies the contract and the contract is replaced by a state machine with both new transitions $dock1[s = 1]$ and $dock2[s = 2]$ with subsequent behaviour *True* (see Figure 7.10).

Although for this refinement step the simpler checks based on syntax suffice to determine that the new transition satisfies the contract, the instantiated form of the above side-condition, that needs to be verified by inspection for the more detailed check would be :

$\vdash ((\exists s, q_1, q_2 : true) \Rightarrow (\exists s, q_1, q_2 : (s' == 1) \wedge (q_1' == q_1) \wedge (q_2' == q_2))) \wedge$

$((\exists s, q_1, q_2 : true) \wedge (s : s' == 1) \Rightarrow (s, q_1, q_2 : true))$

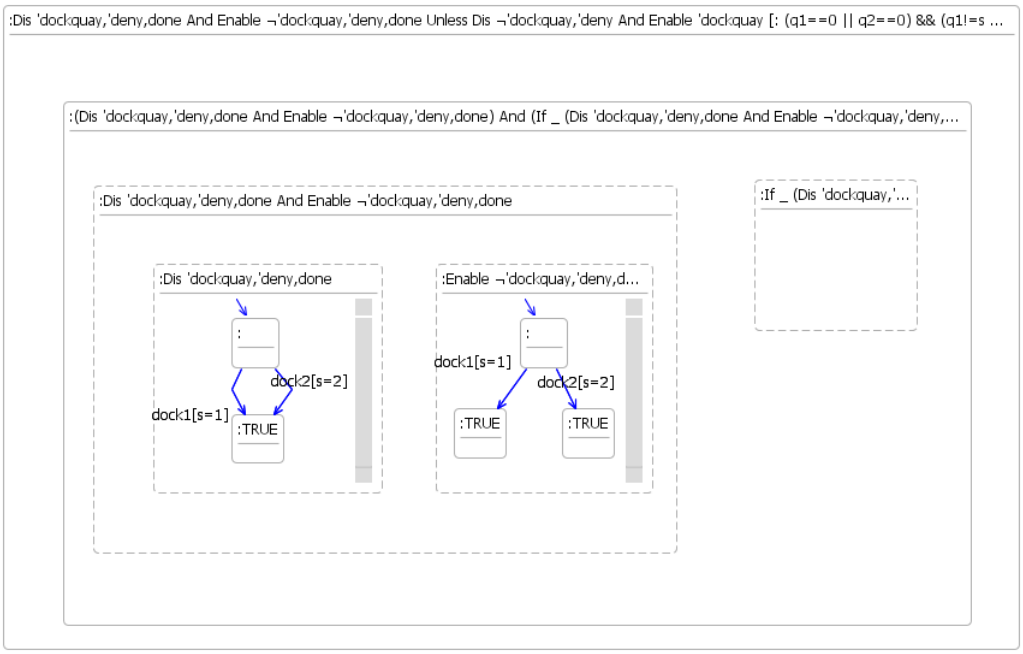

Figure 7.10: TotEnable pattern applied to contract A10

8. Apply the *Combine states* pattern to combine the substates of $A10$ with contract *True*.

9. Apply the *Conjunction elimination* pattern to the state with contract $A1$. The *Conjunction elimination* pattern is applicable if the selected component is a state of type *conjunction*. This pattern can be applied to the conjunction of like diagrams and throws away the copies. The pattern relies on inspection to determine equivalence (see Figure 7.11).

10. Apply pattern *Redundant hierarchy*; the inner contract $A1$ is assigned to the outer contract of the start state. The *Redundant hierarchy* pattern is applicable if the selected component is a state. The pattern checks that the state has substates and no outgoing/incoming transitions. The pattern flattens the level of hierarchy by removing the selected component but retaining its substates. If the selected component is a substate and a start state, the pattern assigns the inner contract of its parent state to the outer contract of its start state. The pattern verifies that the state has substates and no outgoing or incoming transitions, it is itself a substate and a start state so the pattern removes the selected component, (the redundant level of hierarchy), retains its substates and assigns the inner contract of its parent state to the outer contract of its start state (see Figure 7.12).

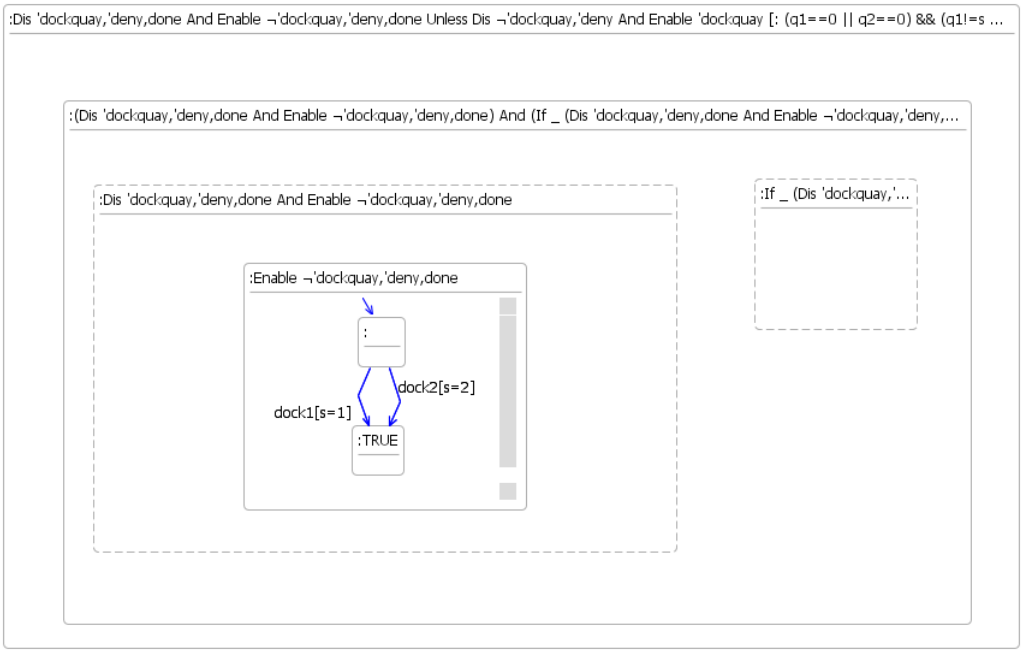

Figure 7.11: Conjunction elimination pattern applied to state with contract A1

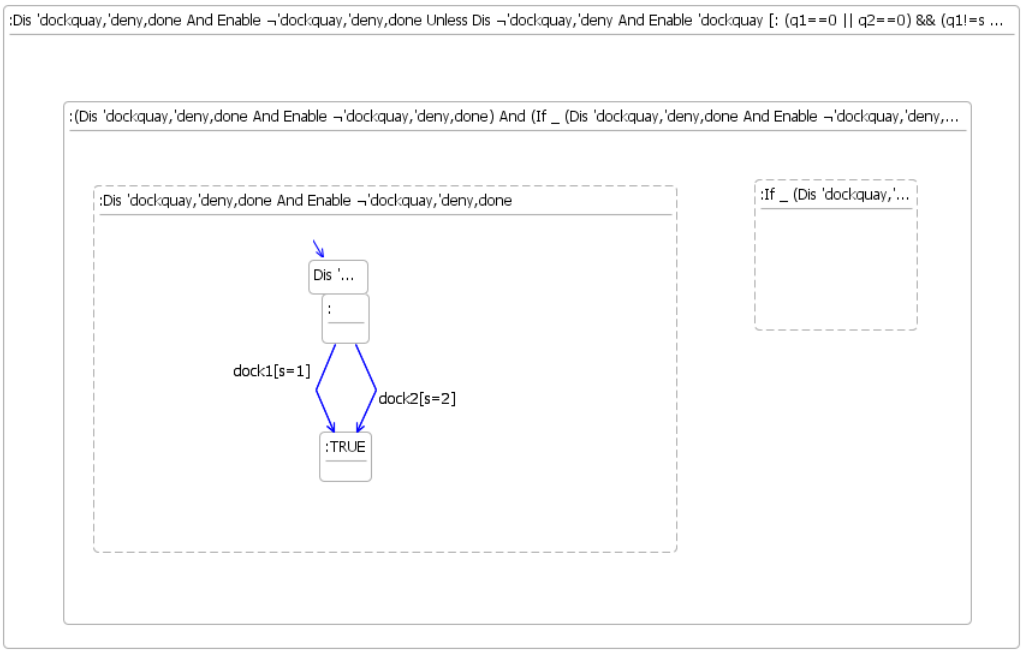

Figure 7.12: Redundant hierarchy pattern applied to state with contract A10

11. Apply the *If* pattern to state with contract $A2$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form *If eventFormula* ϕ or *If eventFormula [updateExpression]* ϕ. *EventFormula* can be a set of events, a negated set of events or the underscore character.

For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist* the side-condition verifies that the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. The model checker (HST) is invoked to decide if $\neg(updateExpr2 \wedge updateExpr1)$ is tautological.

It is not necessary to perform any checks for transitions with subsequent behaviour "ϕ" as if the conditions for the subsequent behaviour "ϕ" do not hold, then the subsequent behaviour is *True* which can then be refined to "ϕ". The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*. A permitted refinement is to weaken the update expression. If the contract is $[a[x : P]]\phi$, and the transition is $a[x : P']$ the proof obligation $\vdash P \Rightarrow P'$ must hold for the transition to subsequently behave as ϕ. We do not need to add this to the side-condition check for transitions with subsequent behaviour of ϕ, as explained above, it is not necessary to perform any checks for these transitions.

The contract is: $A2 = [\_]((\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$

$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$

$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$

$\overline{\langle 'deny[: R]\rangle}))$

The two new transitions required are $dock1[s = 1]$ and $dock2[s = 2]$

The contract $A2$ has the form $[\_]\phi$, which states that *all* enabled transitions subsequently behave like ϕ.

$\phi = (\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$

$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$

$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$

$\overline{\langle 'deny[: R]\rangle})$

The contract is replaced by a state machine with the new transitions $dock1[s = 1]$ and $dock2[s = 2]$ each with subsequent behaviour ϕ (see Figure 7.13).

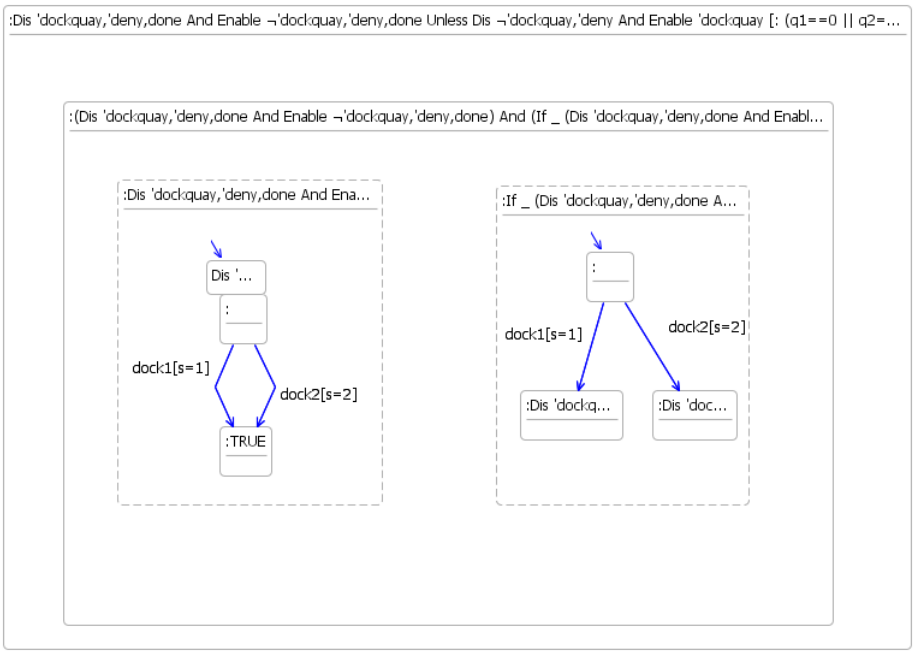

Figure 7.13: If pattern applied to contract A2

12. Apply the *Combine states* pattern to the new substates which both have contract A.

13. Apply *Conjunction elimination* pattern to the state with contract $A1 \wedge A2$.

14. Apply *Redundant hierarchy* pattern, the inner contract is assigned to the outer contract of the start state.

15. Apply *Redundant hierarchy* pattern, inner contract A is assigned to the outer contract of the start state.

7.5.3.2 Refinement stage 2

The overall high-level process of stage 2 further refines the conjunct with contract A, in particular the target state of the *dock1* and *dock2* transitions. The refinement process introduces the *'dockquay* and *'deny* transitions.

16. Further refine the substate with contract A. Apply the *Unfold Unless* pattern. The *Unfold Unless* pattern is applicable if the selected component is a state whose inner contract is an *Unless* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *Unless* ψ" and the pattern refines the contract to "ψ" in this instance (see Figure 7.14).

$A = ((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$

$$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge$$
$$\overline{\langle 'dockquay[q_1,q_2,s:\neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1,q_2,s:\neg R]]\rangle \wedge$$
$$\overline{\langle 'deny[:R]\rangle}))$$

$\phi = ((\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$

$\psi = \overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge$
$\overline{\langle 'dockquay[q_1,q_2,s:\neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1,q_2,s:\neg R]]\rangle \wedge$
$\overline{\langle 'deny[:R]\rangle}$

Apply the *Unfold Unless* pattern, contract A becomes $A3$.

$A3 = \overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge$
$\overline{\langle 'dockquay[q_1,q_2,s:\neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1,q_2,s:\neg R]]\rangle \wedge$
$\overline{\langle 'deny[:R]\rangle}$

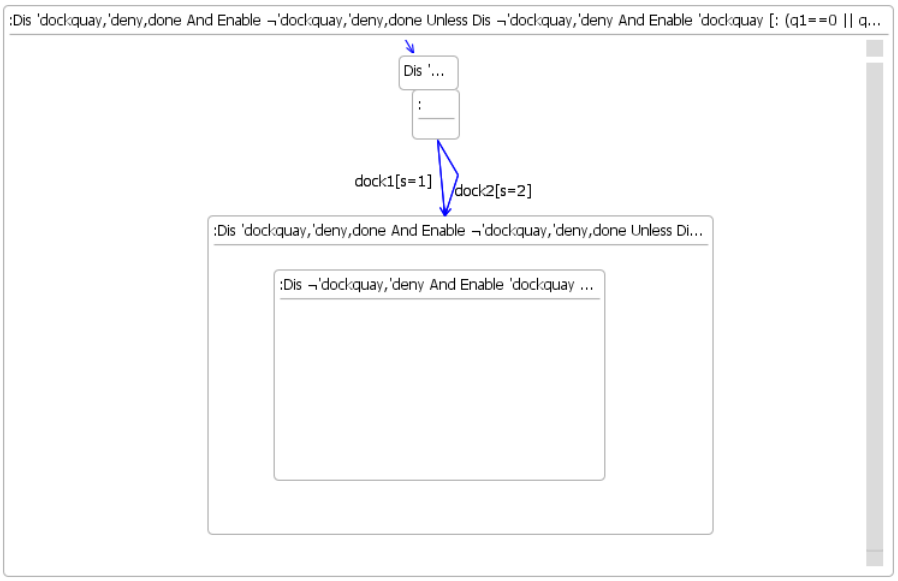

Figure 7.14: Unfold Unless pattern applied to contract A

17. Apply the *Conjunction introduction* pattern to the state with contract $A3$. This introduces two new states one with contract $A4 \wedge A5 \wedge A6 \wedge A7$ and the other with contract $A8$.

$A4 = \overline{\langle -'dockquay,' deny\rangle}$

$A5 = \langle['dockquay[:R]]\rangle$

$A6 = \overline{\langle 'dockquay[q_1,q_2,s:\neg(R \wedge skip)]\rangle}$

$A7 = \langle['deny[q_1,q_2,s:\neg R]]\rangle$

$A8 = \overline{\langle 'deny[: R]\rangle}$

18. Apply the *Conjunction introduction* pattern to the state with contract $A4 \wedge A5 \wedge A6 \wedge A7$. This introduces two new states one with contract $A4 \wedge A5 \wedge A6$ and the other with contract $A7$.

19. Apply the *Conjunction introduction* pattern to the state with contract $A4 \wedge A5 \wedge A6$. This introduces two new states one with contracts $A4 \wedge A5$ and the other with contract $A6$.

20. Apply the *Conjunction introduction* pattern to the state with contract $A4 \wedge A5$. This introduces two new states one with contract $A4$ and the other with contract $A5$.

21. Refine state with contract $A4$, apply the *Disable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

$A4 = \overline{\langle -'dockquay,' deny\rangle}$

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False* so in this instance transitions are only introduced with subsequent behaviour *True*. The pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, the model checker (HST) is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1). The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$. The pattern confirms that the transition events are in the negated contract *eventlist*. The contract is replaced by a state machine with the new transitions each with subsequent behaviour *True* (see Figure 7.15).

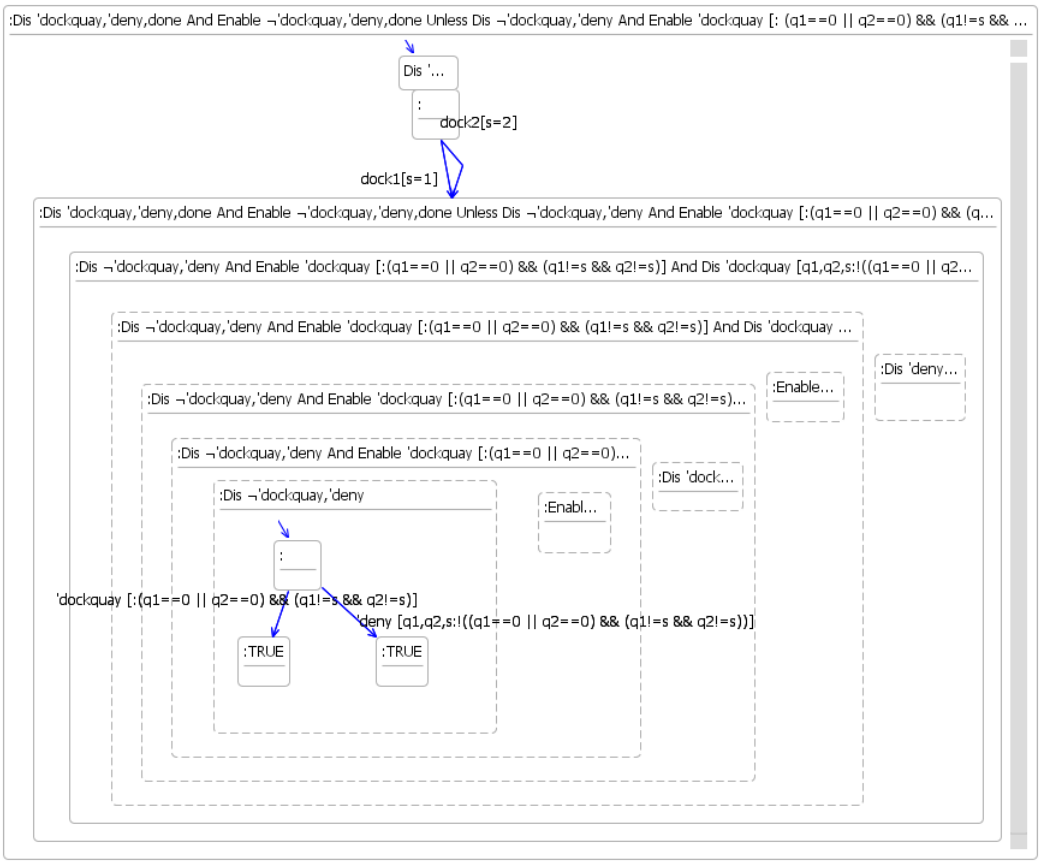

Figure 7.15: Disable pattern applied to contract A4

22. Refine state with contract $A5$, apply the *TotEnable* pattern to introduce two new transitions *'dockquay* and *'deny*. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. Checks based on syntax are performed initially. To determine if the transition satisfies the contract the transition event must be an event included in the contract's *eventlist*. If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore, transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the transition's update expression is equivalent to the contract's update expression then the transition satisfies the contract.

The contract is $A5 = \langle[' dockquay[: R]]\rangle$.

The required new transitions are: $' dockquay[R \Longrightarrow skip]$ and $' deny[!R \Longrightarrow skip]$.

The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. The first transition $' dockquay[R \Longrightarrow skip]$ satisfies the *TotEnable* contract as

'dockquay is in the contract *eventlist* and the transition update expression is equivalent to the contract update expression. The inner contract is replaced by a state machine with both new transitions $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$ with subsequent behaviour *True* (see Figure 7.16).

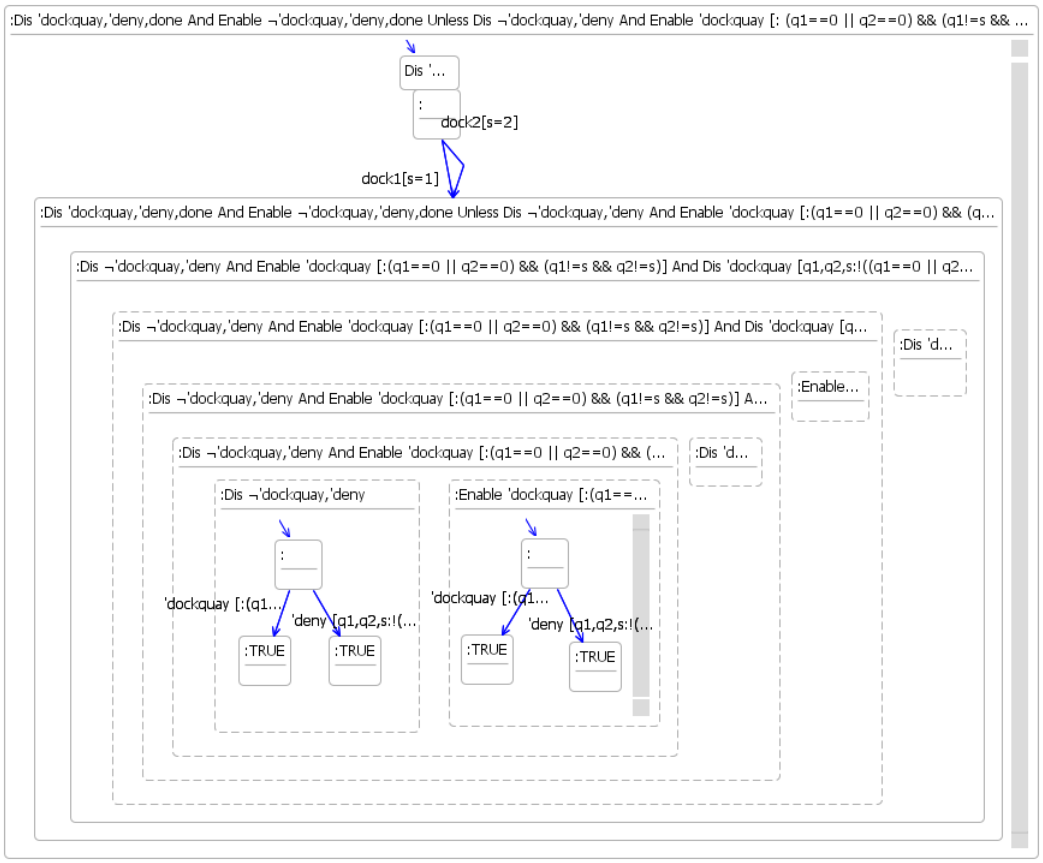

Figure 7.16: TotEnable pattern applied to contract A5

23. Apply the *Conjunction elimination* pattern to the state with contract $A4 \wedge A5$.

24. Apply the *Redundant hierarchy* pattern to the state with contract $A5$. Inner contract $A4 \wedge A5$ is assigned to the outer contract of the start state.

25. Refine state with contract $A6$, apply the *Disable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

$A6 = \overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle}$

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False* so in this instance transitions are only introduced with subsequent behaviour *True*. The pattern checks that the transition event is not in the contract *eventlist* (or it is in

the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, the model checker (HST) is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. the expressions are disjoint and there is no overlap. The model checker (HST) is invoked to decide if $\neg(updateExpr2 \wedge updateExpr1)$ is tautological.

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

For the first transition $'dockquay[R \Longrightarrow skip]$ the event is in the contract eventlist so the model checker (HST) is called to verify that the update expression for the transition $[R \Longrightarrow skip]$ is not consistent with the contract update expression $[q_1, q_2, s : \neg(R \wedge skip)]$. The model checker (HST) confirms that $\neg((R \wedge skip) \wedge (\neg(R \wedge skip)))$ is tautological. The transition is added with subsequent behaviour *True*.

For the second transition $'deny[!R \Longrightarrow skip]$ the pattern checks and confirms that the transition event *'deny* is not in the contract *eventlist* so the transition is added with subsequent behaviour *True* and no further checks are required (see Figure 7.17).

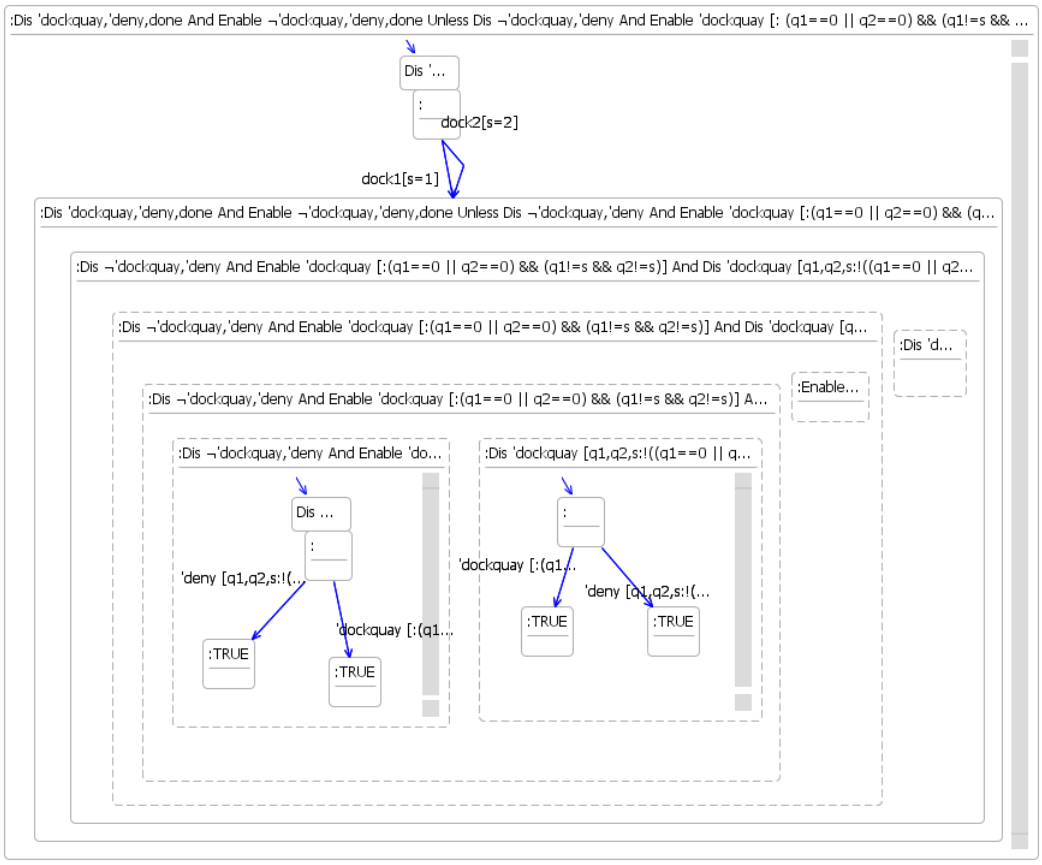

Figure 7.17: Disable pattern applied to contract A6

26. Apply the *Conjunction elimination* pattern to the state with contract $A4 \wedge A5 \wedge A6$.
27. Apply the *Redundant hierarchy* pattern to the state with contract $A6$. Inner contract $A4 \wedge A5 \wedge A6$ is assigned to the outer contract of the start state.
28. Refine state with contract $A7$, apply the *TotEnable* pattern to introduce two new transitions *'dockquay* and *'deny*. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. Checks based on syntax are performed initially. To determine if the transition satisfies the contract the event must be an event included in the contract's *eventlist*.

If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore, transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the transition's update expression is consistent with the contract's update expression then the transition satisfies the contract.

It is a permitted refinement when introducing a transition to strengthen the contract's update expression within the guard. If the update expression in the contract places no restrictions on the possible after values of variables so that they could be updated in any way and the contract is of the form $\langle[eventlist[G]]\rangle$ where G is a guard and refers only to *before* values for variables, it is satisfied by transitions of the form $event[G]$, $event[G \Longrightarrow assignmentlist]$, $event[G \Longrightarrow skip]$.

$A7 = \langle['deny[q_1, q_2, s : \neg R]]\rangle$

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. For the second transition $'deny[!R \Longrightarrow skip]$, *'deny* is in the contract *eventlist*. The contract $A7$ says $'deny$ is enabled when $\neg R$ holds and the variables can be updated in any way. The transition says that a $'deny$ action is enabled when $\neg R$ holds and the variable values remain the same. The transition's update expression strengthens the contract's update expression within the guard which is a permitted refinement. The transition satisfies the contract and the contract is replaced by a state machine with both new transitions (see Figure 7.18).

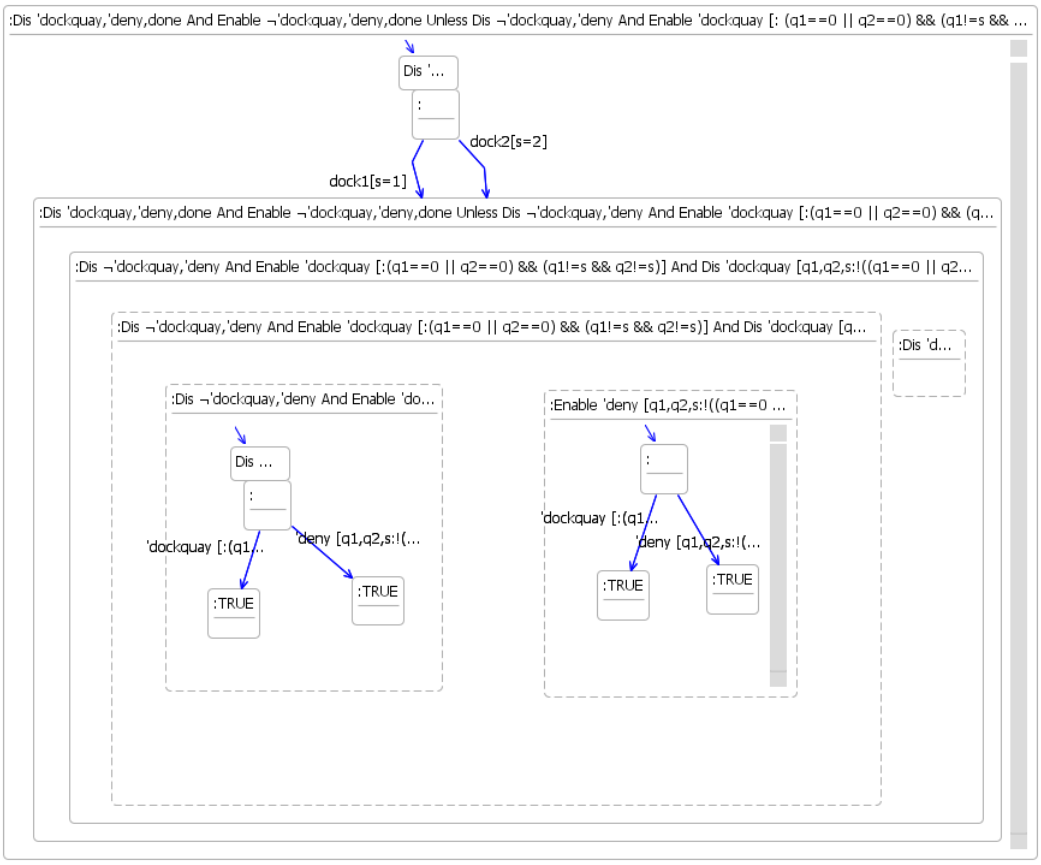

Figure 7.18: TotEnable pattern applied to contract A7

29. Apply the *Conjunction elimination* pattern to the state with contract $A4 \wedge A5 \wedge A6 \wedge A7$.

30. Apply the *Redundant hierarchy* pattern to the state with contract $A7$. Inner contract $A4 \wedge A5 \wedge A6 \wedge A7$ is assigned to the outer contract of the start state.

31. Refine state with contract $A8$, apply the *Disable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

$A8 = \overline{\langle 'deny[: R]\rangle}$ rewritten in terms of the *If* operator $A8 = ['deny[: R]]False$.

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False* so in this instance transitions are only introduced with subsequent behaviour *True*. The pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, the model checker (HST) is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. the update expressions are disjoint and there is no overlap. The model checker (HST)

is invoked to decide if $\neg(updateExpr2 \wedge updateExpr1)$ is tautological.

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

For the first transition $'dockquay[R \Longrightarrow skip]$ the pattern checks and confirms that the transition event *'dockquay* is not in the contract *eventlist* so no further checks are required and the transition to a state with contract *True* is added.

For the second transition $'deny[!R \Longrightarrow skip]$, the event is in the contract *eventlist* so the model checker (HST) is called to verify that the update expression for the transition is not consistent with the contract update expression. The model checker (HST) confirms that $\neg((R \wedge skip) \wedge (\neg R \wedge skip))$ is tautological. The transition $'deny[!R \Longrightarrow skip]$ to a state with contract *True* is added (see Figure 7.19).

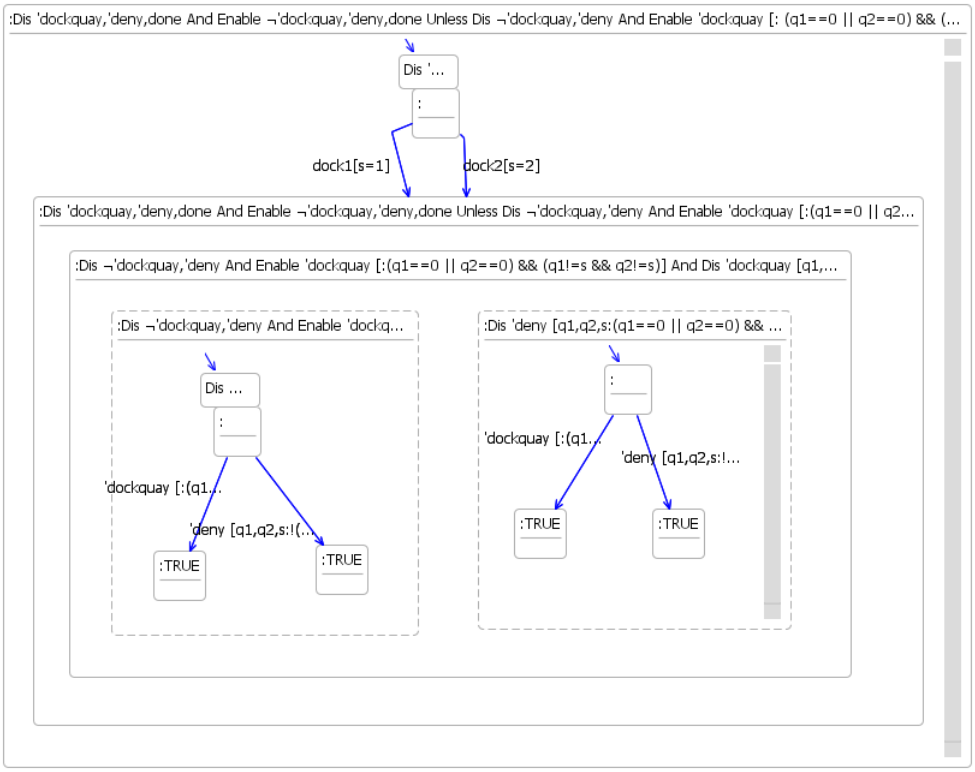

Figure 7.19: Disable pattern applied to contract A8

32. Apply the *Conjunction elimination* pattern to the state with contract A.

33. Apply the *Redundant hierarchy* pattern. The inner contract is assigned to the outer contract of the start state.

34. Apply the *Redundant hierarchy* pattern. Inner contract A is assigned to the outer contract of the start state (see Figure 7.20).

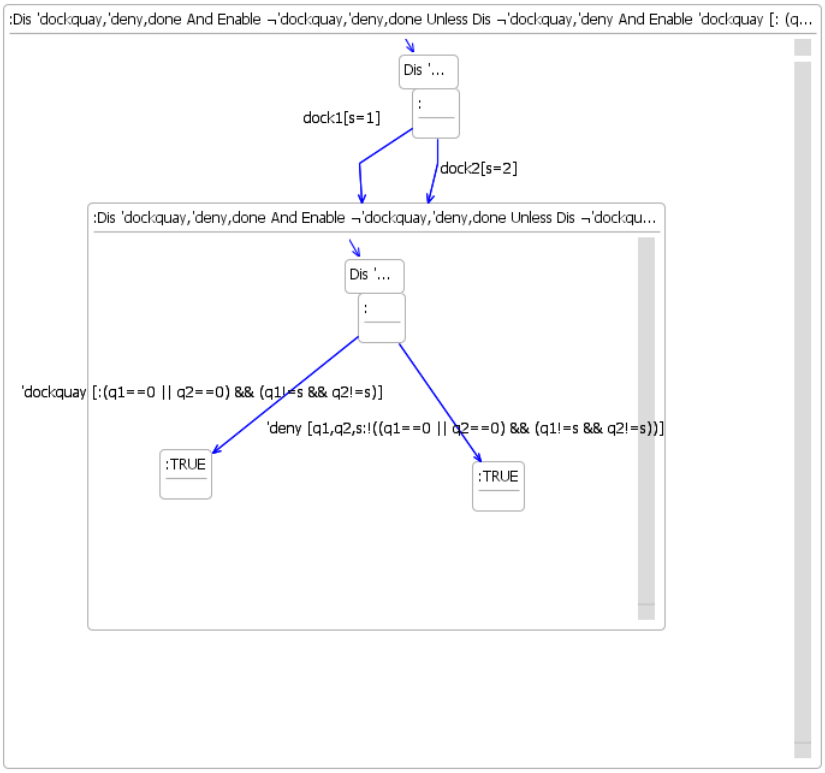

Figure 7.20: Redundant hierarchy pattern applied to state with contract A3

35. Apply the *Move target down* and the *Redundant hierarchy* patterns to the state with contract *A*. The *Move target down* pattern shifts the target state of the incoming transition to the composite state down from the composite state to the default/initial state of the composite. The *Redundant hierarchy* pattern is applicable if the selected component is a state. The pattern checks that the state has substates and no incoming/outgoing transitions. The pattern flattens the level of hierarchy by removing the selected component but retaining its substates (see Figure 7.21).

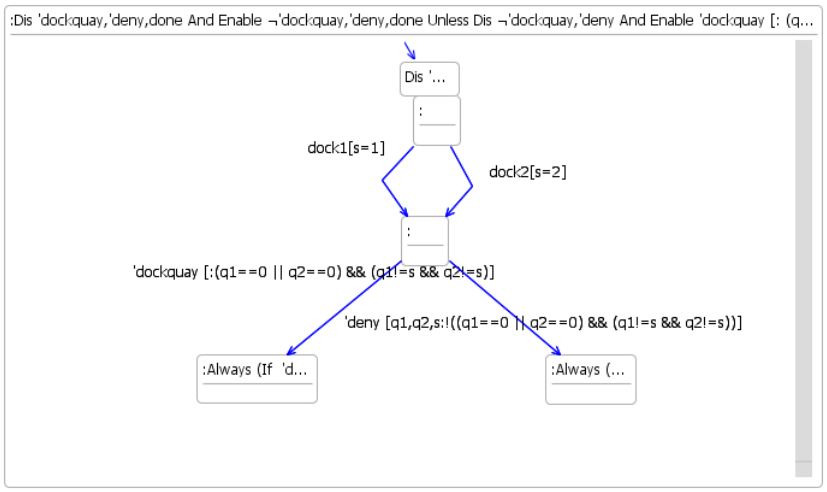

Figure 7.21: Move target down and Redundant hierarchy applied to contract A

The overall design for the *ShipReq* component is shown in Figure 7.22.

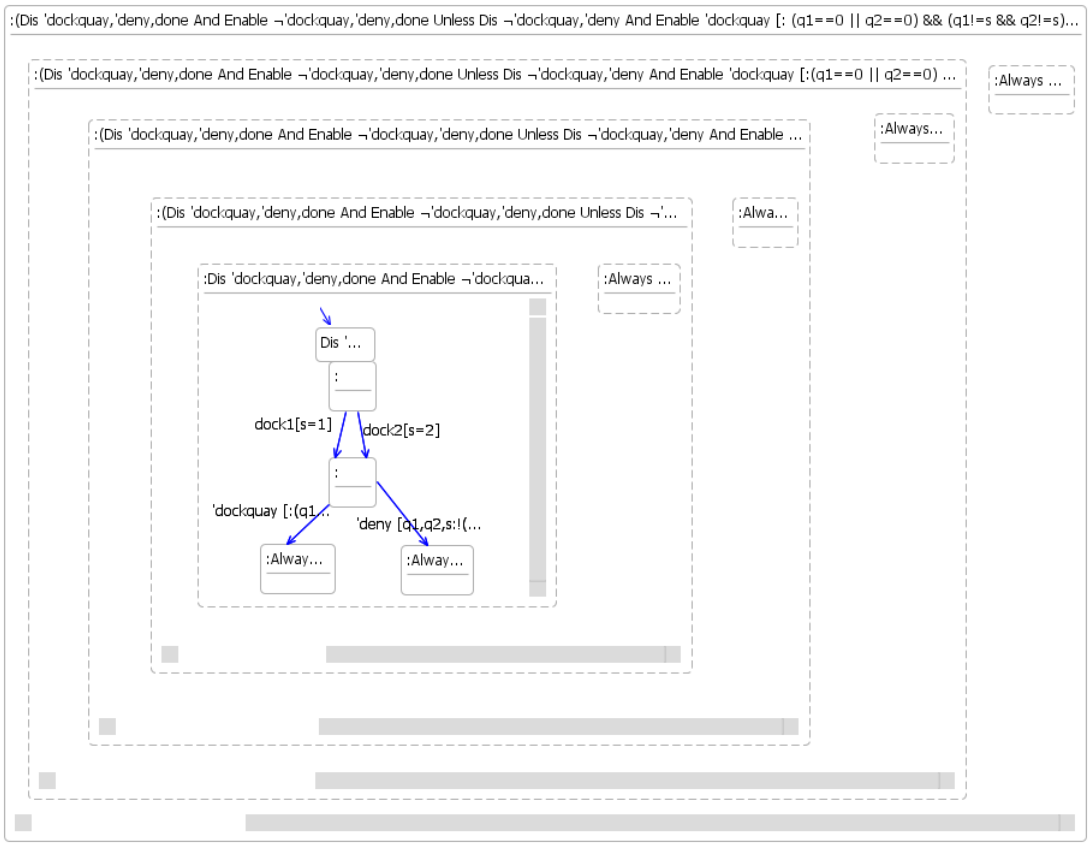

Figure 7.22: The ShipReq component

7.5.3.3 Refinement stage 3

The overall high-level process of stage 3 refines the conjunct with contract B to introduce the *dock1* and *dock2* transitions.

36. Refine the conjunction state with contract B. Apply the *Unfold Always* pattern. The *Unfold Always* pattern is applicable if the selected component is a state whose inner contract is an *Always* operator expression. The pattern checks that the contract for the selected state has the form "*Always* ϕ" and the pattern unfolds the contract to "ϕ *And (If* _ *(* *Always* ϕ*))*".

$$B = \Box[done]((\overline{\langle{}'dockquay,'\, deny, done\rangle} \wedge \langle[-'dockquay,'\, deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,'\, deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge \overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle'deny[:R]\rangle}))$$

B becomes $B1 \wedge B2$

$$B1 = [done]((\overline{\langle{}'dockquay,'\, deny, done\rangle} \wedge \langle[-'dockquay,'\, deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,'\, deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge \overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle'deny[:R]\rangle}))$$

$$B2 = [\_]\Box[done]((\overline{\langle{}'dockquay,'\, deny, done\rangle} \wedge \langle[-'dockquay,'\, deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,'\, deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge \overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle'deny[:R]\rangle}))$$

37. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

38. Apply the *If* pattern to the state with contract $B1$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ". *EventFormula* can be a set of events, a negated set of events or the underscore character.

For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist* the side-condition verifies that the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1).

$$B1 = [done]((\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle ['dockquay[:R]]\rangle \wedge$$
$$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle ['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle 'deny[:R]\rangle}))$$

The two new transitions required are: $dock1[s = 1]$ and $dock2[s = 2]$. Their subsequent behaviour will be *True*. The pattern checks and confirms that the event for each transition is not in the contract *eventlist*. The contract is replaced by a state machine with the two new transitions $dock1[s = 1]$ and $dock2[s = 2]$ (see Figure 7.23).

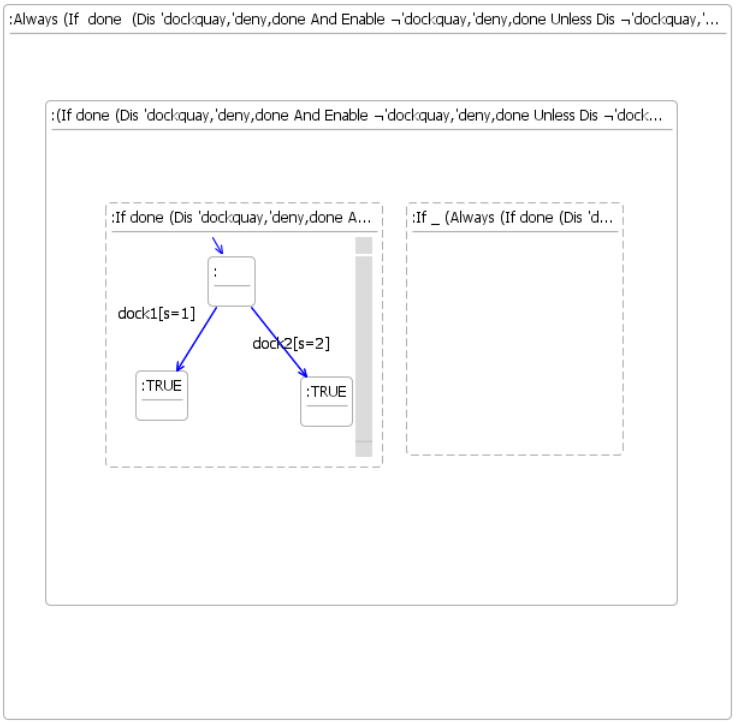

Figure 7.23: If pattern applied to introduce *dock1* and *dock2*

39. Apply the *If* pattern to the state with contract $B2$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ". *EventFormula* can be a set of events, a negated set of events or the underscore character. The contract is:

$$B2 = [\_]\Box[done]((\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle [-'dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle ['dockquay[:R]]\rangle \wedge$$

$$\overline{\langle{}'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle{}'deny[: R]\rangle}))$$

The two new transitions required are, $dock1[s = 1]$ and $dock2[s = 2]$. The contract $B2$ has the form $[\_]\phi$, *all* enabled transitions subsequently behave like ϕ.

$$\phi = \Box[done]((\overline{\langle{}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$$
$$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$$
$$\overline{\langle{}'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle{}'deny[: R]\rangle}))$$

The contract is replaced by a state machine with the two new transitions $dock1[s = 1]$ and $dock2[s = 2]$ that subsequently behave like ϕ (see Figure 7.24).

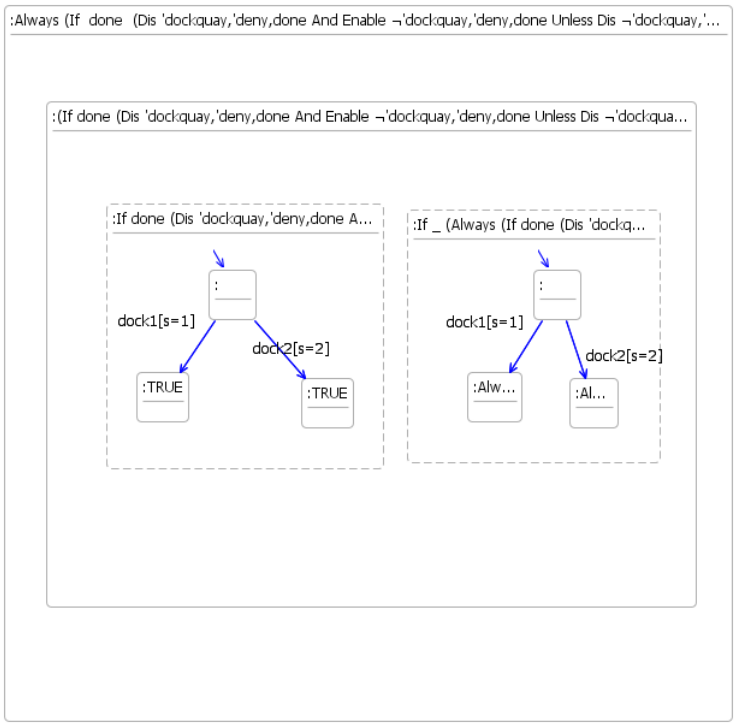

Figure 7.24: If pattern applied to contract B2

40. Apply the *Combine states* pattern to the new substates of $B1$ with the same contracts.

41. Apply the *Combine states* pattern to the new substates of $B2$ with the same contracts.

42. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

43. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract B of the parent state to the outer contract of the start state.

7.5.3.4 Refinement stage 4

The overall high-level process of stage 4 further refines the conjunct with contract B, in particular the target state of the *dock1* and *dock2* transitions. The refinement process introduces the *'dockquay* and *'deny* transitions. The conjunction is eliminated for the state with contract $A \wedge B$ as the A and B conjuncts are equivalent.

44. Further refine the substate with contract B. Apply the *Unfold Always* pattern, contract B becomes $B1 \wedge B2$.

45. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

46. Apply the *If* pattern to the state with contract $B1$ to introduce two new transitions *dockquay* and *deny*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist*.

$$
\begin{aligned}
B1 = [done]((&\overline{\langle {}'dockquay,{}' deny, done\rangle} \wedge \langle[-{}'dockquay,{}' deny, done]\rangle) \\
&\curvearrowright (\overline{\langle -{}'dockquay,{}' deny\rangle} \wedge \langle[{}'dockquay[:R]]\rangle \wedge \\
&\overline{\langle {}'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[{}'deny[q_1, q_2, s : \neg R]]\rangle \wedge \\
&\overline{\langle {}'deny[:R]\rangle}))
\end{aligned}
$$

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

Their subsequent behaviour will be *True*. The pattern checks and confirms that the event for each transition is not in the contract *eventlist* (see Figure 7.25).

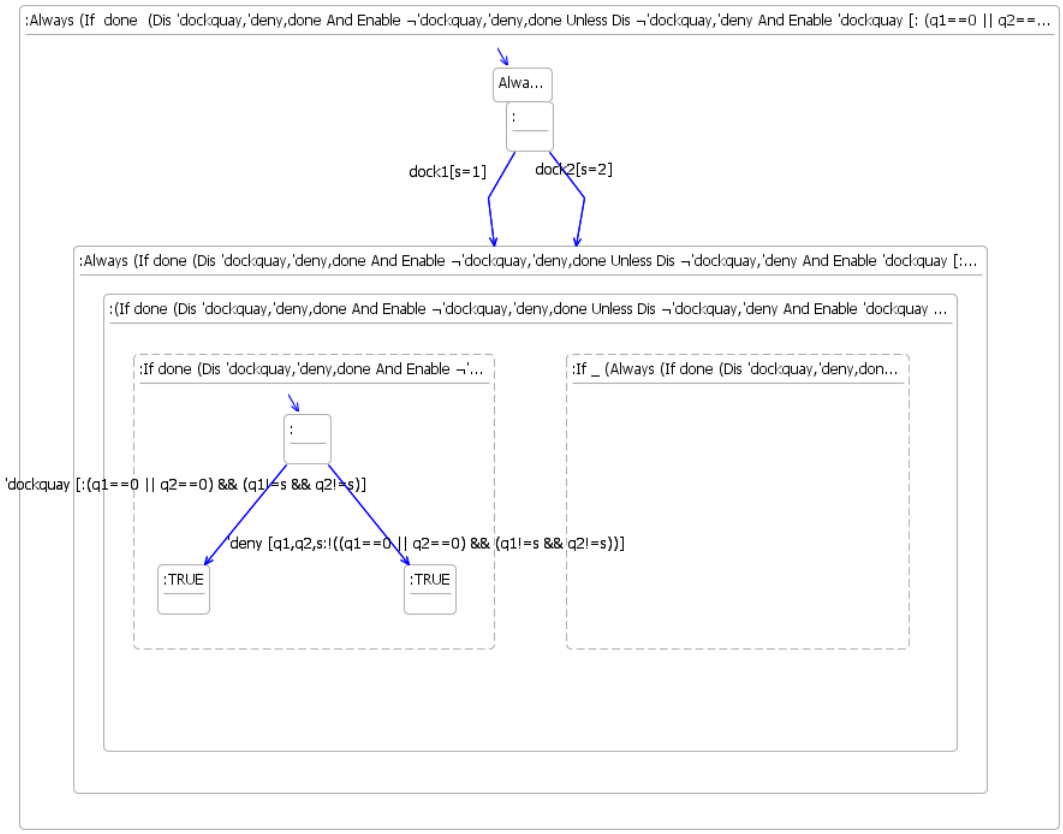

Figure 7.25: If pattern applied to contract B1

47. Apply the *If* pattern to the state with contract $B2$ to introduce two new transitions *dockquay* and *deny*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression.

The contract is:

$$B2 = [\_]\Box[done]((\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge \overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle'deny[:R]\rangle}))$$

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

The contract $B2$ has the form $[\_]\phi$, *all* enabled transitions subsequently behave like ϕ.

$$\phi = \Box[done]((\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle) \curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge \overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle'deny[:R]\rangle}))$$

The contract is replaced by a state machine with the transitions $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$ that subsequently behave like ϕ (see Figure 7.26).

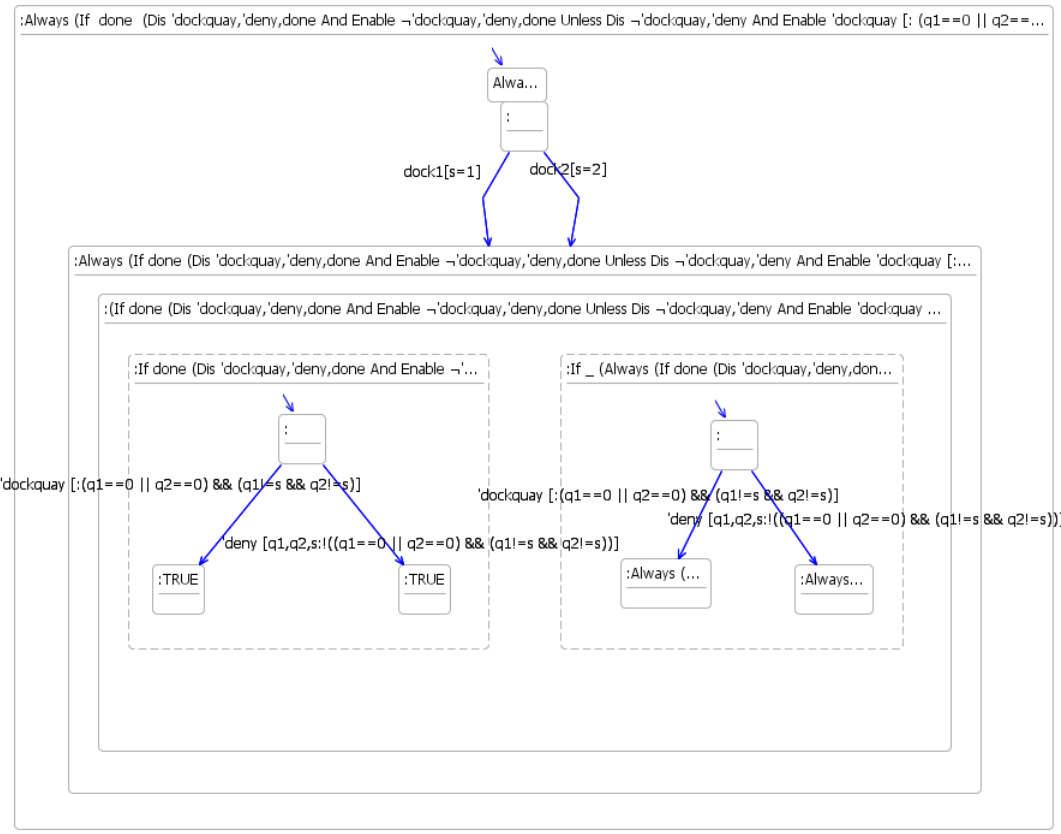

Figure 7.26: If pattern applied to introduce *dockquay* and *deny*

48. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

49. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

50. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent state to the outer contract of the start state (see Figure 7.27).

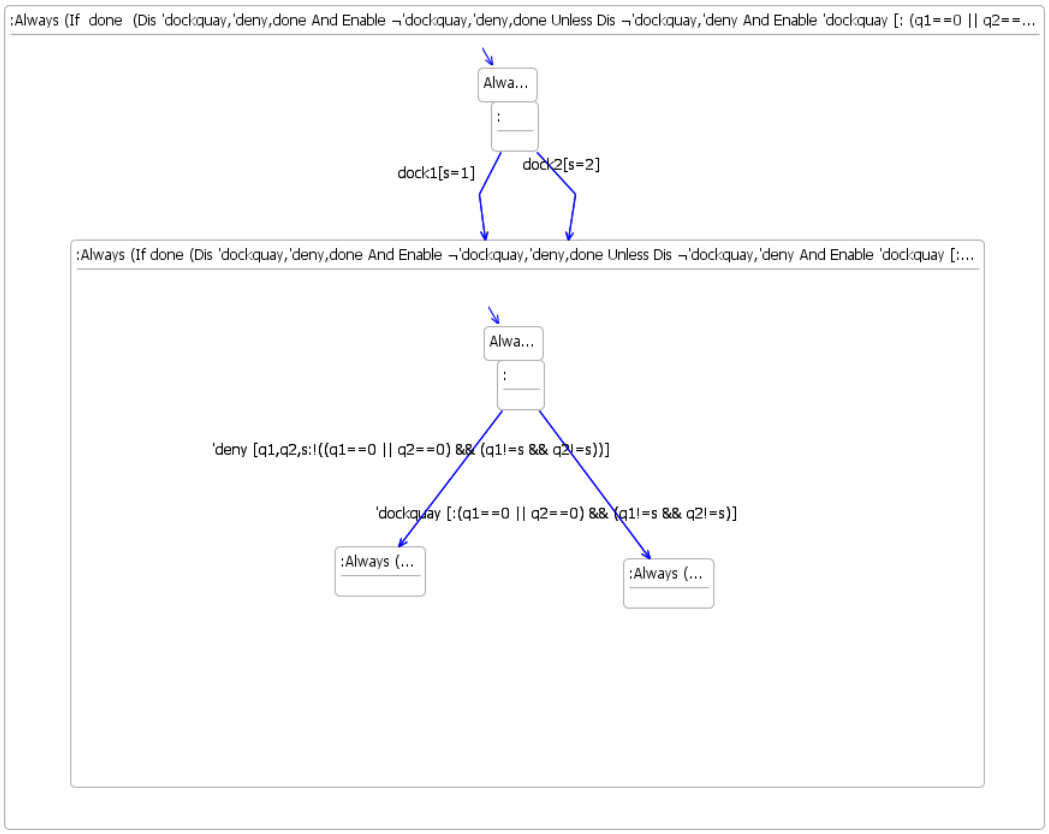

Figure 7.27: Redundant hierarchy pattern applied to state with contract B1

51. Apply the *Move target down* and the *Redundant hierarchy* patterns to the state with contract B.

The overall design for the *ShipReq* component at this stage is (see Figure 7.28):

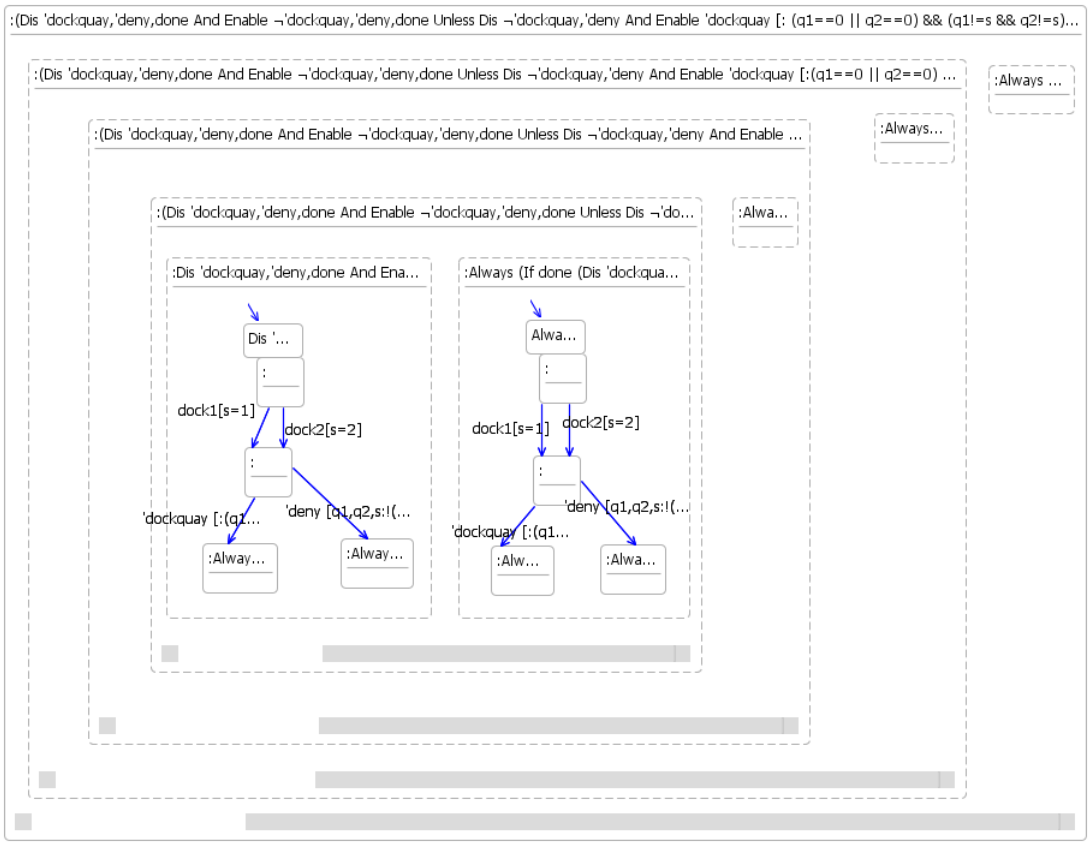

Figure 7.28: The ShipReq component

52. Apply the *Conjunction elimination* pattern to the state with contract $A \wedge B$.

53. Apply the *Redundant hierarchy* pattern. Assigns the inner contract $A \wedge B$ of the parent state to the outer contract of the start state (see Figure 7.29).

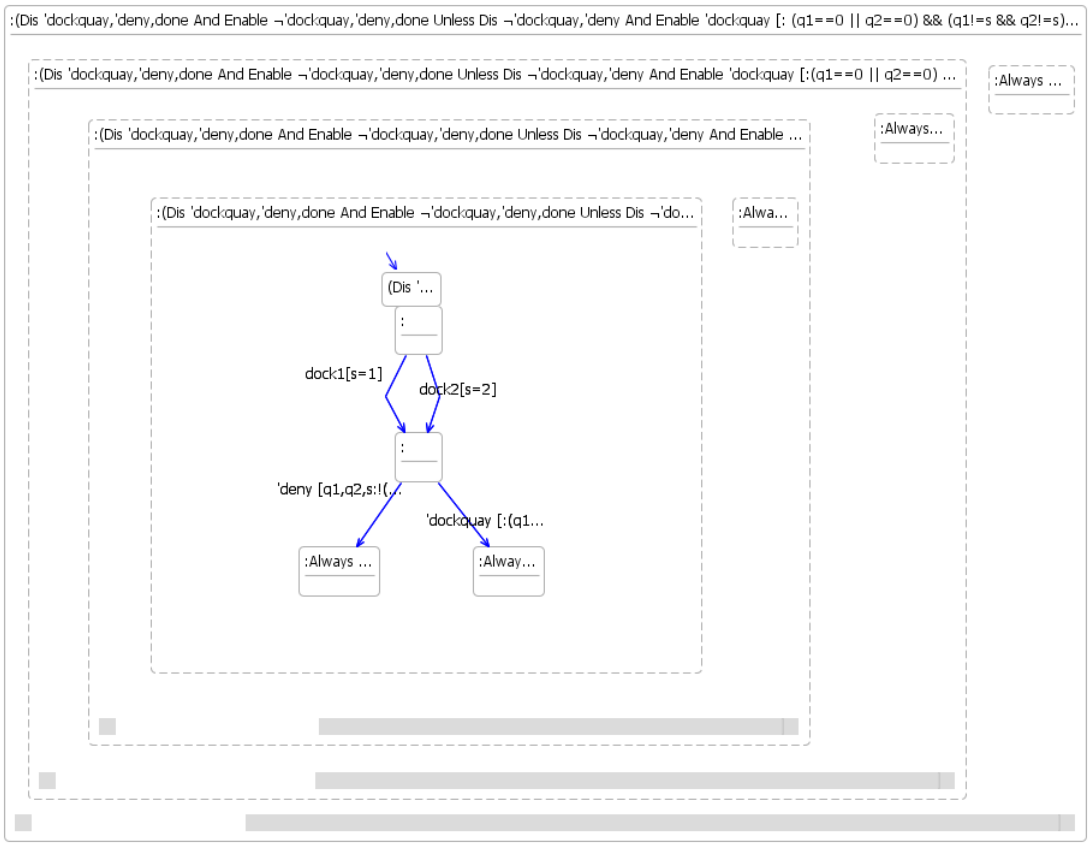

Figure 7.29: Conjunction elimination followed by Redundant hierarchy

7.5.3.5 Refinement stage 5

The overall high-level process of stage 5 further refines the state with contract $A \wedge B$, in particular the target state of the *'dockquay* transition. The refinement process introduces the *done* and *'docked* transitions. The *'docked* transition is rerouted to loop back up to the start state.

54. Substate with contract B and incoming transition *'dockquay* is further refined. Apply the *Unfold Always* pattern, B becomes $B1 \wedge B2$

55. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

56. Apply the *If* pattern to the state with contract $B1$ to introduce a new transition, *done*.

57. Apply the *If* pattern to the state with contract $B2$ to introduce a new transition *done*.

58. Further refine the substate of $B1$ with contract A. Apply the *Unfold Unless* pattern. Contract A becomes $A1 \wedge A2$.

59. Apply the *Conjunction introduction* pattern to the state with contract $A1 \wedge A2$.

60. Apply the *Conjunction introduction* pattern to the state with contract $A1$. Introduces two new states with contracts $A9$ and $A10$.

61. Apply the *Disable* pattern to the state with contract $A9$ to introduce the *docked* transition.

62. Apply the *TotEnable* pattern to the state with contract $A10$ to introduce the *docked* transition.

63. Apply the *Conjunction elimination* pattern to the state with contract $A1$.

64. Apply pattern *Redundant hierarchy*, inner contract $A1$ is assigned to the outer contract of the start state.

65. Apply the *If* pattern to state with contract $A2$ to introduce the *docked* transition.

66. Apply the *Conjunction elimination* pattern to the state with contract $A1 \wedge A2$.

67. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

68. Apply the *Redundant hierarchy* pattern. Assigns the inner contract A of the parent state to the outer contract of the start state.

69. Further refine the substate of $B2$ that is the target of the *done* transition with contract B. Apply the *Unfold Always* pattern, contract B becomes $B1 \wedge B2$.

70. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

71. Apply the *If* pattern to the state with contract $B1$ to introduce the *docked* transition.

72. Apply the *If* pattern to the state with contract $B2$ to introduce the *docked* transition.

73. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$ retaining substate with contract $B2$.

74. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

75. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent state to the outer contract of the start state.

76. Apply the *Move target down* and the *Redundant hierarchy* patterns to both the target states of the *done* transitions.

77. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

78. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

79. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent

state to the outer contract of the start state.

80. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target state of the *dockquay* transition (see Figure 7.30).

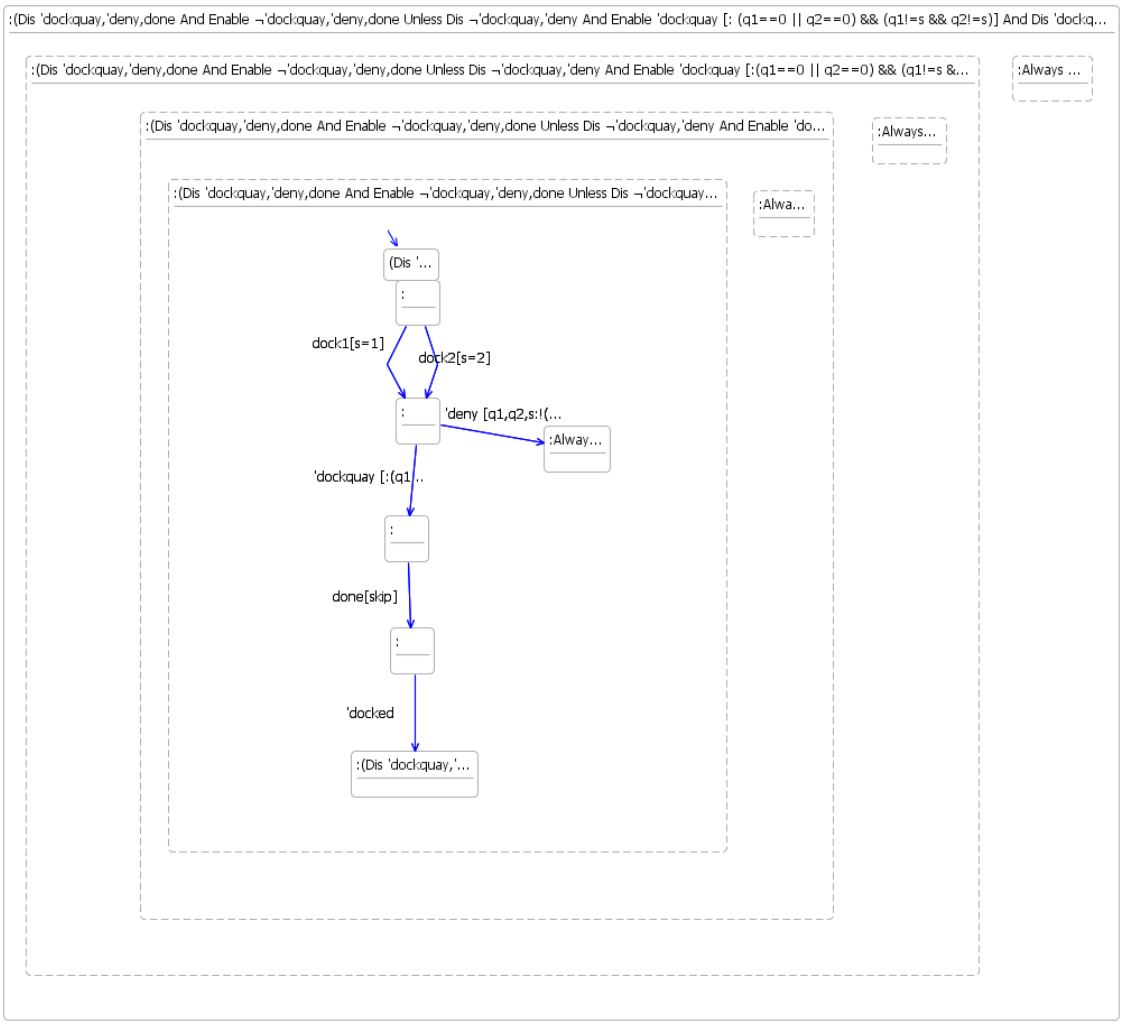

Figure 7.30: Move target down and the Redundant hierarchy patterns applied

81. Apply the *Reroute* pattern to the target of the *docked* transition. This reroutes the *docked* transition to the start state which has the same contract.

The *Reroute* pattern is applicable if the selected component is a state. The pattern checks that the state has incoming transitions but no outgoing transitions or substates. The pattern removes the selected state and reroutes its incoming transitions to another state, chosen by the user, whose outer contract is equivalent to the inner contract of the selected state. The pattern checks the contracts are syntactically equivalent (see Figure 7.31).

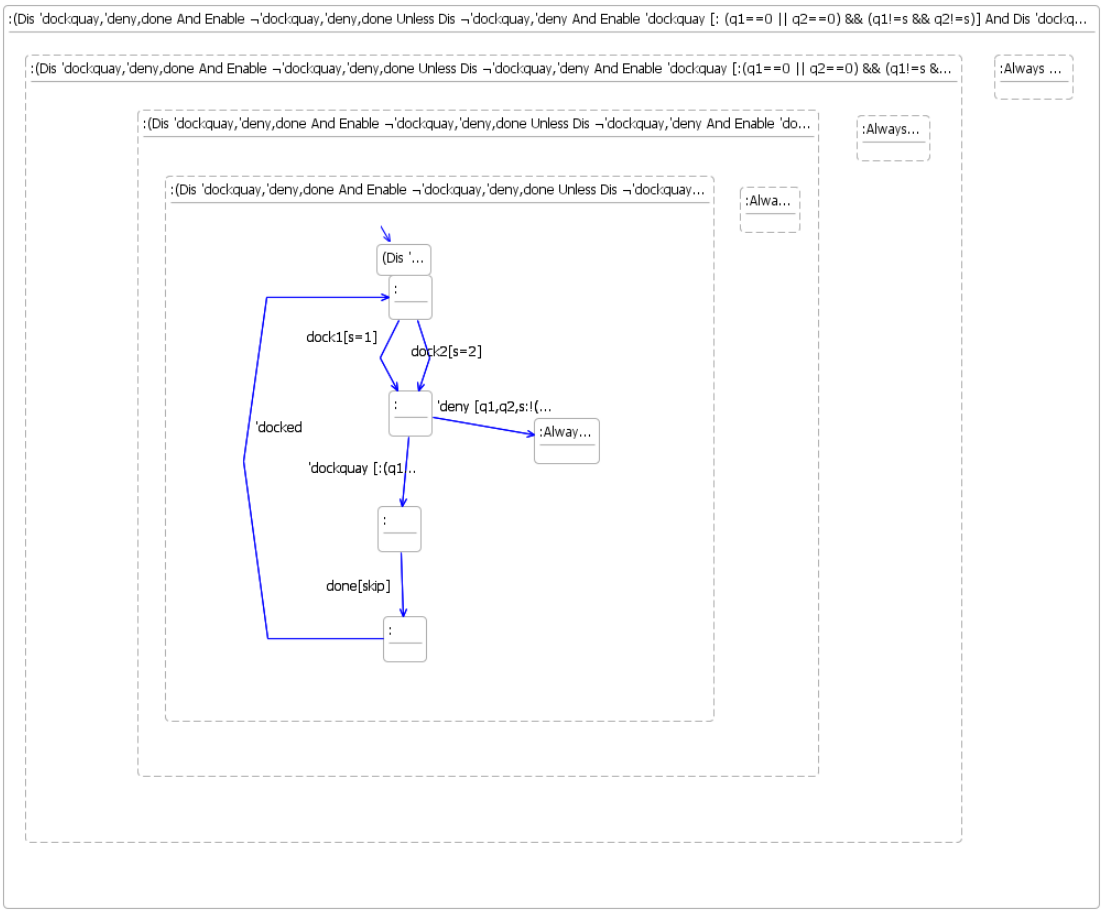

Figure 7.31: Reroute pattern applied to target of *docked* transition

Figure 7.32 shows the required final design for the parallel components *ShipReq* and *Quay*.

$$R = (q_1 == 0 \vee q_2 == 0) \wedge (q_1 != s \wedge q_2 != s)$$

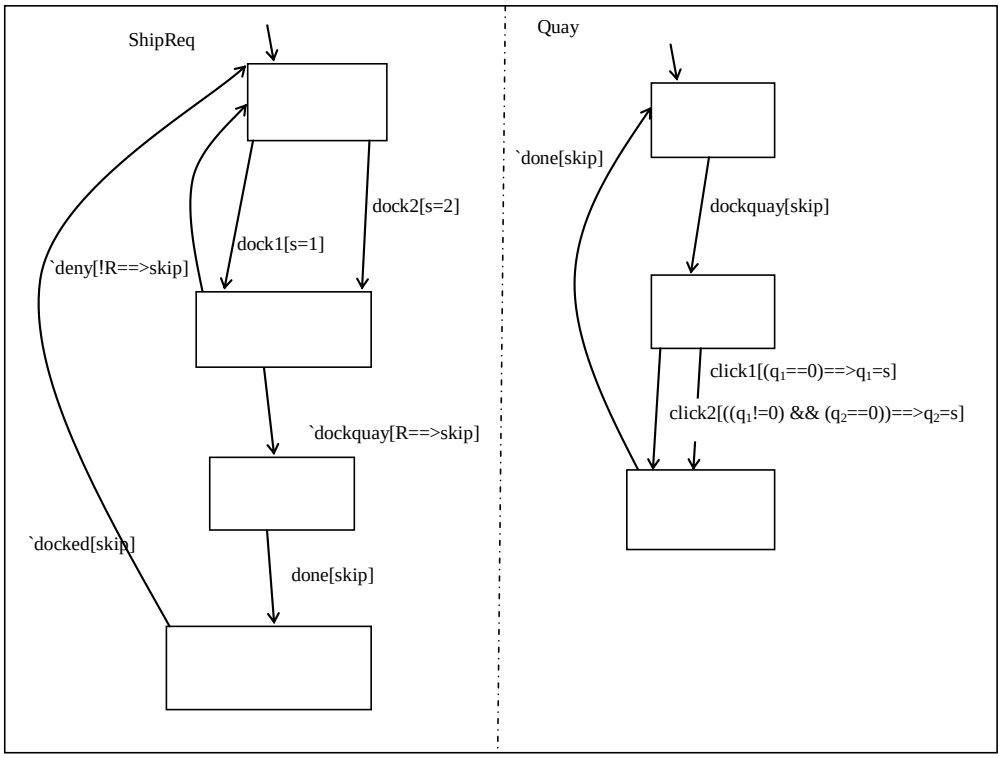

Figure 7.32: Required design of the parallel components

7.5.4 Summary of patterns applied to refine the ShipReq component

This section presents a summary of the patterns applied to refine the open contract for *ShipReq* to a CoSta state machine design. The first part of the design process for the *ShipReq* component has been described above. The refinement of the *ShipReq* component took more than two hundred steps in total. Appendix C gives details of the full refinement.

```
1  A AND B AND C AND D AND E
2  A AND B AND C AND D AND E Conjunction Introduction
3   A AND B AND C AND D Conjunction Introduction
4    A AND B AND C Conjunction Introduction
5     A AND B Conjunction Introduction
6       A Unfold Unless
7       A Conjunction Introduction
8         A1 Conjunction Introduction
```

```
9           A9 Disable //Introduce dock1 & dock2
10          A9 Combine States
11          A10 TotEnable //Introduce dock1 & dock2
12          A10 Combine States
13        A1 Conjunction Elimination
14        A1 Redundant Hierarchy
15        A2 If //Introduce dock1 & dock2
16        A2 Combine States
17      A Conjunction Elimination
18      A Redundant Hierarchy
19      A Redundant Hierarchy
20      A-leaf
21      A Unfold Unless
22       A3
23        A4 AND A5 AND A6 AND A7 AND A8 Conjunction Introduction
24        A4 AND A5 AND A6 AND A7 Conjunction Introduction
25          A4 AND A5 AND A6 Conjunction Introduction
26            A4 AND A5 Conjunction Introduction
27              A4 Disable //Introduce `dockquay & `deny
28              A5 TotEnable //Introduce `dockquay & `deny
29            A4 AND A5 Conjunction Elimination
30            A4 AND A5 Redundant Hierarchy
31            A6 Disable //Introduce `dockquay & `deny
32          A4 AND A5 AND A6 Conjunction Elimination
33          A4 AND A5 AND A6 Redundant Hierarchy
34          A7 TotEnable //Introduce `dockquay & `deny
35        A4 AND A5 AND A6 AND A7 Conjunction Elimination
36        A4 AND A5 AND A6 AND A7 Redundant Hierarchy
37        A8 Disable //Introduce `dockquay & `deny
38      A Conjunction Elimination
39      A Redundant Hierarchy
40      A Redundant Hierarchy
41      A-leaf Move Target Down & Redundant Hierarchy
42      B Always Unfold
43      B Conjunction Introduction
44        B1 If //Introduce dock1 & dock2
45        B1 Combine States
46        B2 If //Introduce dock1 & dock2
47        B2 Combine States
48      B Conjunction Elimination
49      B Redundant Hierarchy
50      B Redundant Hierarchy
51      B-leaf
52        B Always Unfold
53        B Conjunction Introduction
54          B1 If //Introduce `dockquay & `deny
55          B2 If //Introduce `dockquay & `deny
56        B Conjunction Elimination
57        B Redundant Hierarchy
58        B Redundant Hierarchy
59      B-leaf Move Target Down & Redundant Hierarchy
60    A AND B Conjunction Elimination
61    A AND B Redundant Hierarchy
62    A AND B-leaf //`dockquay
63      B Always Unfold
64      B Conjunction Introduction
65        B1 If //Introduce done transition
66        B1-leaf
67          A Unfold Unless
```

```
68          A Conjunction Introduction
69            A1 Conjunction Introduction
70              A9 Disable //Introduce 'docked
71              A10 TotEnable //Introduce 'docked
72            A1 Conjunction Elimination
73            A1 Redundant Hierarchy
74            A2 If //Introduce 'docked
75          A Conjunction Elimination
76          A Redundant Hierarchy
77          A Redundant Hierarchy
78        B1-leaf Move Target Down & Redundant Hierarchy
79        B2 If //Introduce done transition
80        B2-leaf
81          B Always Unfold
82          B Conjunction Introduction
83             B1 If //Introduce 'docked
84             B2 If //Introduce 'docked
85          B Conjunction Elimination
86          B Redundant Hierarchy
87          B Redundant Hierarchy
88        B2-leaf Move Target Down & Redundant Hierarchy
89      B Conjunction Elimination
90      B Redundant Hierarchy
91      B Redundant Hierarchy
92    A AND B-leaf Move Target Down & Redundant Hierarchy
93    A AND B Reroute
94    C Always Unfold
95    C Conjunction Introduction
96      C1 If //Introduce dock1 & dock2
97      C1 Combine States
98      C2 If //Introduce dock1 & dock2
99      C2 Combine States
100   C Conjunction Elimination
101   C Redundant Hierarchy
102   C Redundant Hierarchy
103   C-leaf
104     C Always Unfold
105     C Conjunction Introduction
106       C1 If //Introduce 'dockquay & 'deny
107       C2 If //Introduce 'dockquay & 'deny
108     C Conjunction Elimination
109     C Redundant Hierarchy
110     C Redundant Hierarchy
111   C-leaf Move Target Down & Redundant Hierarchy
112   C-leaf //'dockquay
113     C Always Unfold
114     C Conjunction Introduction
115       C1 If //Introduce done
116       C2 If //Introduce done
117     C Conjunction Elimination
118     C Redundant Hierarchy
119     C Redundant Hierarchy
120   C-leaf Move Target Down & Redundant Hierarchy
121   C-leaf //done
122     C Always Unfold
123     C Conjunction Introduction
124       C1 If //Introduce 'docked
125       C2 If //Introduce 'docked
126     C Conjunction Elimination
```

```
127       C Redundant Hierarchy
128       C Redundant Hierarchy
129      C-leaf Move Target Down & Redundant Hierarchy
130      C Reroute
131     A AND B AND C Conjunction Elimination
132     A AND B AND C Redundant Hierarchy
133    D Always Unfold
134    D Conjunction Introduction
135      D1 If //Introduce dock1 & dock2
136      D1 Combine States
137      D2 If //Introduce dock1 & dock2
138      D2 Combine States
139    D Conjunction Elimination
140    D Redundant Hierarchy
141    D Redundant Hierarchy
142    D-leaf
143      D Always Unfold
144      D Conjunction Introduction
145        D1 If //Introduce 'dockquay 'deny
146        D1-leaf ('dockquay)
147          D3 Unfold Unless
148            D4
149             D5 AND D6 AND D7 Conjunction Introduction
150                D5 AND D6 Conjunction Introduction
151                  D5 Disable //Introduce done
152                  D6 TotEnable //Introduce done
153                D5 AND D6 Conjunction Elimination
154                D5 AND D6 Redundant Hierarchy
155                D7 Disable //Introduce done
156          D3 Conjunction Elimination
157          D3 Redundant Hierarchy
158          D3 Redundant Hierarchy
159        D1-leaf Move Target Down & Redundant Hierarchy
160        D2 If //Introduce 'dockquay & 'deny
161        D2-leaf
162         D Always Unfold
163         D Conjunction Introduction
164           D1 If //Introduce done
165           D2 If //Introduce done
166          D Conjunction Elimination
167          D Redundant Hierarchy
168          D Redundant Hierarchy
169        D2-leaf Move Target Down & Redundant Hierarhy
170      D Conjunction Elimination
171      D Redundant Hierarchy
172      D Redundant Hierarchy
173     D-leaf Move Target Down & Redundant Hierarchy
174     D-leaf (done)
175      D Always Unfold
176      D Conjunction Introduction
177        D1 If //Introduce 'docked
178        D2 If //Introduce 'docked
179      D Conjunction Elimination
180      D Redundant Hierarchy
181      D Redundant Hierarchy
182     D-leaf Move Target Down & Redundant Hierarchy
183     D Reroute
184   A AND B AND C AND D Conjunction Elimination
185   A AND B AND C AND D Redundant Hierarchy
```

```
186  E Always Unfold
187  E Conjunction Introduction
188    E1 Disable //Introduce dock1 & dock2
189    E1 Combine States
190    E2 If //Introduce dock1 & dock2
191    E2 Combine States
192  E Conjunction Elimination
193  E Redundant Hierarchy
194  E Redundant Hierarchy
195  E-leaf
196    E Always Unfold
197    E Conjunction Introduction
198      E1 Disable //Introduce `dockquay & `deny
199      E2 If //Introduce `dockquay & `deny
200    E Conjunction Elimination
201    E Redundant Hierarchy
202    E Redundant Hierarchy
203  E-leaf Move Target Down & Redundant Hierarchy
204  E-leaf (`dockquay)
205    E Always Unfold
206    E Conjunction Introduction
207      E1 Disable //Introduce done
208      E2 If //Introduce done
209    E Conjunction Elimination
210    E Redundant Hierarchy
211    E Redundant Hierarchy
212  E-leaf Move Target Down & Redundant Hierarchy
213  E-leaf (done)
214    E Always Unfold
215    E Conjunction Introduction
216      E1 Disable //Introduce `docked
217      E2 If //Introduce `docked
218    E Conjunction Elimination
219    E Redundant Hierarchy
220    E Redundant Hierarchy
221  E-leaf Move Target Down & Redundant Hierarchy
222  E Reroute
223 A AND B AND C AND D AND E Conjunction Elimination
224 A AND B AND C AND D AND E Redundant Hierarchy
225 A AND B AND C AND D AND E (deny) Reroute
```

Listing 7.1: Patterns applied to refine the ShipReq contract to a design

7.6 Evaluation of results

This section evaluates the results of the case study to indicate the strengths and weaknesses of the patterns and the design process. The purpose of the case study is to gauge the viability and methodological usefulness of the refinement and refactoring patterns and design method for Contractual State Machines. Overall aims of the research are for a rigorous design process for statecharts to enable construction of models which support abstraction and the systematic and stepwise introduction of detail.

Each refinement or refactoring step has been expressed as a pattern in terms of model transformations. Software tools have been implemented to automate the design process with an emphasis on usability. The case study illustrates an application of the refinement and refactoring patterns to give us empirical confidence that the patterns are correct and applicable to the design of a system. The case study determines the practicality of the approach and suitability of the patterns. The criteria for evaluating the patterns and design strategy include utility, coverage, completeness, correctness and consistency.

7.6.1 Utility

The utility criteria concerns the utility of the patterns themselves, utility of the pattern catalogue as a whole, and practicality and effectiveness of the tool support and design strategy. Patterns were defined in Chapters 4 and 5 where arguments were presented to show why each pattern is valuable in terms of proof reduction and/or methodological utility. The case study demonstrates that patterns and the design strategy support the top-down, stepwise, component-based design of a program from a contract to a detailed operational design that is guaranteed to preserve the properties of the original specification. The case study develops the design through pattern application where each step permits the gradual introduction of more specific details about the behaviour of the system, by introducing design constructs and reducing nondeterminism.

Patterns reduce the overall proof/automated checking process into a sequence of steps requiring considerably less proof effort to discharge them. Patterns deal with specific applications of a refinement theory, and thus reduce the demonstration of correctness to showing that a specific side-condition is valid rather than performing a full refinement check. Simplifying proof burden reduces the required levels of resources (time to compute, memory requirements). In the case of a refinement check the algorithm may not succeed (as it requires too much memory or time to compute). More practically, when diagram components are analysed to enable and disable the applicable patterns, time constraints apply and it may not be practical to decide deeper semantic constraints.

The pattern catalogue is compact and succinct. The catalogue has a core set of *specific* patterns. A *specific* pattern, with respect to a more *general* pattern, is a pattern constrained for use in less situations, usually with a simpler side-condition which reduces the proof burden. These patterns perform simple refinement and refactoring steps and are thus easy to understand and apply. The specific patterns are augmented with some

general rules that are there in a supporting rather than an essential role should the specific patterns not cover what is required. As the size of the pattern catalogue increases it will become more important to explore ways in which it could be structured to assist the engineer in finding patterns to use.

The case study achieved the final design through application of the core patterns only, (the most complex side-conditions involved verifying conjectures supplied as predicates on the data variables to be proven using HST), the general rules (that perform full refinement checks) were not required. Applying fine-grained patterns however in full detail could be expensive. Compound patterns may be effective in reducing the number of design steps required. For example a compound pattern could be created to remove a redundant transition as an alternative transition exists and thus simplify a design. The compound pattern could consist of the sequential composition of a pattern to *Strengthen a transition guard to false* followed by a pattern to *Remove a transition with a false guard*. Compound patterns are discussed further in Chapter 8, section 8.3.2.

The catalogue is succinct and avoids an excessive choice of patterns that can be applied to substructures of a model to achieve a particular result. At each step in the refinement process it was obvious which pattern to apply to achieve the desired model, as typically only one pattern was applicable to the part of the system selected for the type of transformation required. After the engineer has chosen part of the design to elaborate further, if more than one pattern is applicable the engineer selects the pattern to perform the required model transformation (such as introducing transitions or flattening hierarchy).

Software tools have been developed that include a graphical editor for Contractual State Machines and support for the repository of refinement and refactoring patterns and their application. The software tools further reduce the development and proof effort during the refinement of a design through automating the model transformations and the discharge of side-conditions and integration of the software with model checking technology. Automation and patterns greatly reduced development and proof effort over comparable approaches such as formal proof and thus improved usability. The case study demonstrated that the tools are practical and they have the advantage of speed and accuracy when compared to proof by hand. The time and effort required to work through the pilot versions of the case study by hand, (while the tool was still being developed) was approximately reduced by half, once the software was available. Although familiarity with the languages and refinement relations had an impact, a speed up of 50% using the

tools is significant. Development by hand was slower in some places (due to it being conducted manually) but maybe faster in others as it was possible to miss out some steps by informally grouping them into compound patterns intuitively.

7.6.2 Completeness

The research presented in this thesis is concerned with the practicability of refinement rather than its completeness. The thesis does not seek a completeness result and it has been acknowledged that the refinement and refactoring patterns are not complete. The formal underpinning of the language is not yet mature enough to achieve a formal proof of completeness. The completeness criteria discussed here concerns coverage of transformations and ensures that the catalogue consists of a comprehensive set of patterns for the common types of refinement or refactoring that are frequently required during the stepwise design of a system. General patterns are also included in the catalogue should the specific patterns not cover what is required.

The case study demonstrated that a contract could express high-level properties of a design (e.g. safety properties) and abstract functional behaviour, and patterns could support a top-down design process from a contract to a detailed operational design. The case study showed that the final design could be achieved through application of the core patterns only. The general patterns, that conduct a full refinement check, were not required. Patterns cover all operators of the language and there were no instances of *missing* patterns during the case study for transformations that were not covered by the existing core patterns within the catalogue. No additional patterns needed to be added to the catalogue. As yet no patterns have been removed from the catalogue but some patterns were not required in the case study, (such as *Remove transition with a false guard*, and *Move source up*). The case study also demonstrated applicability of the patterns to the design problem. The coverage of required transformations by the core patterns went some way towards showing that the patterns are a complete and minimal set. Complete, as all required refinements and refactorings to achieve the required design were covered by the specific patterns; and minimal in the sense that there was no ambiguity as to which pattern to apply to the design at a particular step to obtain the desired model.

7.6.3 Soundness

An essential check is to ensure that the refinement method is sound, i.e. that it doesn't result in programs that are not compliant with their specifications. Refinement steps must be correct and result only in consistent designs that are valid with respect to the semantics. The soundness check establishes the feasibility of applying the proposed refinement and refactoring patterns.

Ideally a fully formal argument of correctness for the patterns would be provided. However, priorities (the patterns themselves) and the practical difficulties of concurrent research meant this was not possible. An alternative approach would have been to provide an empirical argument, e.g. use the model checker (HST) to show each *instance* of applying a pattern was correct. Unfortunately this was not possible either as implementation of the model checking algorithms was not complete.

A strong argument that the patterns are correct is that they have been inspected by peers, including supervisors, those with expertise internal to the department, collaborators and peer reviewers of the publication. They have been used experimentally in the case study, which gives us empirical confidence that they are correct. Use of the patterns on the case study did not produce false positives (a proven but bad design, by misapplication of the rules) or false negatives (correct designs that are not provable). There is further work required to formally prove the soundness and completeness of the refinement and refactoring steps.

7.6.4 Weaknesses

The case study demonstrated the patterns and method were viable for achieving a successful result, (the patterns were complete enough to get from a specification to a design) and that the tools worked well. Refinement is a relatively complex process, the excessive number of models involved during the design of a system may be considered a weakness. It can be seen from the case study that applying patterns to transform contracts/designs at the primitive level i.e. using only the simplest (axiomatic) patterns may result in many design steps.

It may be argued that patterns are not very practical for designing systems as they are at a relatively low level and perform only simple refinement and refactoring steps. To reduce the number of design steps required and improve practicality of the approach a possible solution is to define patterns for larger design steps, e.g compound patterns which

would have more engineering value. This provides an important agenda for extending the work. It can be observed from the case study that the application of patterns was also to some extent repetitive. Commonly occurring pattern structures were evident during the case study. It would save effort to find a way of automating them, this is discussed in Chapter 8. Further case studies of larger scope will be required to test the scalability of the approach as the language and pattern catalogue develops.

Open contracts provide the engineer with the ability to express high-level properties of a design (e.g. safety properties) or to abstract functional behaviour which must be preserved by the development. Patterns support a top-down design process from an open contract to a detailed operational design that is guaranteed to preserve the properties of the original specification. It may be argued however that even for a fairly simple system it is difficult to express high-level properties and abstract functional behaviour with the contract language in a clear and uncomplicated manner where it is easy to see that the contracts are obviously correct.

7.7 Conclusions

The case study demonstrated that a contract could express high-level properties of a design (e.g. safety properties) and abstract functional behaviour. Patterns supported a practical top-down design process from a contract to a detailed operational design that maintained consistency between designs. The case study illustrated that the effective tool support/method can be the basis of an automated refinement process providing rigour but not potentially at the cost of practicality.

In this chapter the refinement and refactoring patterns are successfully applied in practice to a design example. The case study provided some evidence that the patterns were applicable to the problem and complete enough to refine an abstract specification of a system into a concrete design. The tools were effective and practical and reduced design time when compared to proof by hand. The case study was also used to analyse the expressiveness of the language and thus supported the work of the group as a whole.

Drawbacks were the verbosity of the languages, and the fastidiousness of having to apply many steps. Compound patterns may be a possible solution to reduce the number of design steps. The case study has shown proof of concept of the use of patterns/MDE in the formal development of visual design languages. The case study has identified the need for further work, such as compound patterns and some further research for the languages

and their formal underpinning and the tool support.

Chapter 8

Conclusions and further work

8.1 Introduction

This chapter will summarise the conclusions of the research and make proposals for further work. The extent to which the research objectives (presented in chapter 3) have been met and the thesis hypothesis addressed is discussed. Finally the chapter concludes with a discussion of possible future work to improve or complement this research.

The thesis has presented a rigorous, model-based and tool supported approach for building reactive systems. The approach is based on an extension to state machines, allowing mixed declarative and operational specifications. The engineering process involves application of refinement and refactoring patterns, a catalogue of which has been developed, and implemented using update-in-place transformation technology. The process and use of the extended language was thoroughly demonstrated in the case study.

The rest of this chapter is organised as follows: Section 8.2 assesses how well the thesis objectives and hypothesis have been addressed. Section 8.3 explores new research challenges which lead on from the work presented in this thesis. Section 8.4 summarises conclusions and the contribution of the thesis.

8.2 Review findings

This section summarises the ways in which the research for this thesis has contributed to the research field and the extent to which the hypothesis/aims of the thesis have been fulfilled. The motivation and rationale behind the thesis proposition were discussed in Chapters 1 and 2. Chapter 2 gave an overview of the research area, discussing the current

state of practice and an analysis of closely related research for this thesis leading to an identification of the remaining open problems for the research area.

Recall from chapters 1 and 2 that a wider acceptance of formal methods within industry has been hindered for many reasons; these include usability and scalability issues [3, 4, 35, 46, 95]. Semi-formal languages are considered to be intuitive but generally lack support for a systematic refinement process. Statecharts, for example, have no rigorous process for constructing models which support abstraction and the systematic and stepwise introduction of detail.

Related research so far has considered the stepwise refinement of concurrent systems [51] and refinement calculi for formal languages for concurrent systems [217]. Research by [94] [242] [222] [206] and [253] has concerned refactorings for state machines. Refinement transformations for variants of state machines have also been proposed [148] [166] [221] [216]. Previous research has considered pattern-based approaches to state machine design [133] [222] [121]. Research has been conducted into a refinement calculus for state machines [87–89] and refinement patterns that allow the reduction of nondeterminism and support verifiable top-down development of state diagrams [218].

Notably, in previous research, the languages of use have typically not been heterogeneous, and did not consider data or shared variables. Specifically they did not integrate contracts with state machines, and also did not provide refinement steps to refine a high-level abstract contract to an operational state machine design. Although some previous research has proposed a formal refinement calculus [221], refinement/refactoring patterns in terms of model transformations have not been considered. Additionally there is an absence of tool support for pattern application and verification of refinement/refactoring steps.

Chapter 2 identified a number of open problems. They included the importance of a graphical specification/design language for safety-critical, concurrent, reactive systems with an underlying formal theory that supports refinement with techniques for abstraction and the ability to express both functional behaviour and high-level properties. In addition, supported by a set of refinement and refactoring patterns that enable designs to be correctly constructed in a stepwise fashion with software tools to automate the design process. The research hypothesis proposed in Chapter 3 (and restated below) incrementally improves on the state-of-the-art by addressing the above open research problems.

Research hypothesis

We can identify for Contractual State Machines, a comprehensive set of refinement and refactoring patterns that ensure consistency between designs and enable the refinement of an abstract contract (that specifies high-level properties) to a fully specified concrete design (that preserves the high-level properties). The aim is to ensure the completeness, correctness, utility and consistency of the patterns. Furthermore we can express each refinement or refactoring step as a pattern, in terms of model transformations. We can automate the systematic engineering process for Contractual State Machines by implementing software with an emphasis on usability.

Current reactive systems engineering practice provides limited *systematic* support for an approach like that proposed in the thesis hypothesis and objectives, to refine an abstract specification to a concrete state machine model. With reference to the thesis proposition and aims, the ways in which this research meets the objectives will be discussed next. The objectives of the thesis are summarised in the table below.

Number	Objective
01	Identify a comprehensive set of refinement and refactoring patterns for CoSta to support the top-down, stepwise design of a system and preserve the functionality of the original specification.
02	Specify the refinement and refactoring steps as update-in-place model transformations (patterns).
03	Implement software to support a repository of refinement and refactoring patterns and their application (to validate side-conditions and perform model transformations) during the refinement of a design.
04	Further validate the hypothesis by conducting a case study.

Table 8.1: Thesis objectives

Chapters 4 and 5 address objectives *01* and *02*. They present a set of refinement and refactoring patterns expressed as model transformations for Contractual State Machines that resolve design choices or reduce nondeterminism to support a top-down design process from a contract to a CoSta state machine design. Chapter 6 addresses objectives *02* and *03*. Software has been implemented to support a repository of patterns (expressed as update-in-place model transformations) and their application to validate side-conditions and perform model transformations. The case study in Chapter 7 addresses objective *04*

to further validate the hypothesis.

An aim of objective *01* is to ensure the completeness of the patterns which concerns coverage of transformations for the common types of refinement or refactoring that are frequently required during the stepwise design of a system. The thesis addresses this as follows.

The case study in Chapter 7 demonstrated applicability of the patterns to the design problem. An abstract contract was refined to a fully elaborated state machine model using only the refinement and refactoring patterns in the catalogue (via the tool support). The coverage of required transformations by the core patterns went some way towards showing that the patterns are a *complete* and *minimal* set. They are complete in the sense that all necessary refinements/refactorings to achieve the required design were covered by the patterns; and minimal in the sense that there was no ambiguity as to which pattern to apply to the design at a particular step to obtain the desired model.

An aim of objectives *01, 02* and *03* was for the refinement and refactoring patterns to ensure consistency between designs and preserve the functionality of the original specification. Contracts express formal properties that describe the behaviour of the system under development and refinement and refactoring patterns guarantee that the properties are satisfied as the system evolves. This is achieved by adding rigour to the process of introducing design detail, through the use of model transformations to implement patterns. The patterns presented in Chapters 4 and 5 are implemented as update-in-place model transformations (in Chapter 6) that preserve correctness of the models that are produced throughout the development process to prevent the inadvertent introduction of mistakes and omissions. Additionally the patterns maintain desired properties between designs by validating constraints. These range from simpler syntax-based checks to more complex side-conditions (e.g relating to conditions/actions on data) that require the model checker (HST) to discharge them.

A strong argument that the patterns are correct is that they have been inspected by peers and used experimentally on a case study, which gives us empirical confidence that they are correct. Correctness is also indicated by familiarity/similarity with patterns from related research. Informally it has been checked for each pattern that the refinement or refactoring step preserves meaning by comparing the semantic interpretation of the target model with that of the source model and ensuring that the ready sets do not change and new non-determinism is not introduced. Of course formal proofs would be helpful; this is

future work.

The research for this thesis has successfully addressed and demonstrated the viability of the proposals in the thesis hypothesis. An important outcome from the case study was the identification of existing limitations and how these might be addressed. Although the case study showed that the concept worked (and the hypothesis is viable), it highlighted the fact that a design process based only on the application of basic patterns is not very tractable for the engineer as it requires too many design steps to be manageable.

The work presented in this thesis has gone a considerable way towards demonstrating proof of concept for the proposed method. However, further development will be necessary before the approach can be applied to a full scale safety-critical software project. Refinement and refactoring patterns successfully embody and capture the strengths of a rigorous model-driven approach for the design of concurrent, reactive systems; but some improvement is needed in order to scale the method to large and complex software systems. The next section gives suggestions for further work that is now appropriate.

8.3 Further work

This section discusses future work to improve or complement the research in this thesis. Although this thesis has gone a considerable way to providing proof of concept, much remains to be done and an interesting agenda of research can be envisaged. There are several potential courses for furthering the research presented in this thesis, some of which are discussed in this section. Making progress in the areas outlined below will produce a combined approach which is more efficient and complete.

8.3.1 Extend the pattern catalogue

Further research could be conducted to extend the pattern catalogue. This could be achieved in several ways, for instance, by identifying more patterns for parallel states and hierarchy. There are three main approaches that could be adopted for patterns for hierarchy. A new hierarchy operator could be introduced into the contract language. With the current contract notation it is not possible to design an outer level of hierarchy with transitions and then design inner substates. To support this a richer contract notation could be specified to describe behaviour within nested levels of hierarchy.

Another approach could be to define state machine to state machine patterns for

hierarchy, e.g. *Introduce hierarchy* and *Flatten hierarchy* patterns. With this approach hierarchy is only permitted in designs at the state machine level when all contracts have been fully refined. This however will lead to large state spaces, rendering designs difficult to understand and use during the part of the design process that does not permit hierarchical abstraction. An alternative approach is to modify the existing patterns for hierarchy. The continuation behaviour of a composite state would be specified by a contract that is the conjunction of all the contracts from the substates within the composite whose behaviour has not yet been specified.

Patterns for parallel composition could also be defined, for example, to introduce (or remove) orthogonality by replacing a sequential (or orthogonal) design with an orthogonal (or non-orthogonal) equivalent. The pattern catalogue could be extended in other ways for example by systematically investigating different refinement choices for the existing patterns and deriving new patterns based on these. Additionally, certain types of patterns could be researched, such as general patterns that integrate behavioural model checking further into the pattern technology by having very general side-conditions, for example, based on a refinement check. These patterns might be invoked to eliminate certain forms of redundancy.

There may also be potential in exploiting the duality of contracts, by interpreting contracts as their characteristic processes (rather than the temporal specification they characterise). It would then be possible to offer multi-target transformations, where the user chooses one of several valid refinements satisfying a contract, i.e. Contractual State Machines that are a mixture of diagramming elements and further contracts. The user would supply the number of transitions into the future they wish to go and the tool would offer them a selection of viable targets.

There is a trade-off to negotiate here. An increase in the number of transitions taken into account when proposing potential targets will make more progress with the design in a single step. This may help to reduce the overall number of design steps required and thus streamline the design process. On the other hand it could lead to greater complexity for the design step, with more potential targets being possible, that the user will have to navigate and select from, in order to proceed. Further work is required to investigate practical solutions to manage this trade-off.

8.3.2 Compound patterns

The case study demonstrated that the concept of refinement and refactoring patterns to embody and capture the strengths of a rigorous model-driven approach for the design of concurrent, reactive systems was successful. A conclusion of the case study however was that a refinement process that is based on the application of only primitive, core patterns can become prohibitive for the engineer as it may require too many steps. There are several complementary directions that could be developed to improve practicability such as compound patterns and these are discussed next.

The purpose of compound patterns is to make the design process more structured and reduce the need for an excessive number of fine-grained refinement/refactoring steps. Compound patterns permit more control over the size of the refinement/refactoring steps and can be used to simplify the development. Compound patterns could be constructed by sequencing primitive patterns. There are instances where the same sequences of patterns are repeatedly required during the design process. These are candidates for compound patterns which capture frequently used large steps in terms of trusted atomic steps, leading to fewer design steps and enabling their reuse to make the design process more tractable for the engineer.

Examples of potential compound patterns for Contractual State Machines.

Name: *Flatten hierarchy*

Type: Refactoring

Rationale: The purpose of this pattern is to remove a level of hierarchy.

Constraints: The pattern is applicable to a composite state with substates.

Parameters: A composite state.

Transformation: This is a compound pattern consisting of the sequential composition of the following patterns, *Move source down*, *Move target down*, *Remove a composite superstate*. The composite parent state is removed to flatten a level of hierarchy leaving the set of substates that were grouped within it.

Name: *Introduce hierarchy*

Type: Refactoring

Rationale: The purpose of this pattern is to cluster states and aggregate state transitions having the same labels and target states.

Constraints: The applicability constraint restricts application to a group of states having the same parent state and an identical set of outgoing transitions, (the transitions have the same labels and the same target state).

Parameters: A set of states.

Transformation: This is a compound pattern consisting of the sequential composition of the following patterns, *Create composite superstate*, *Move source up*, *Move target up*. The set of states are grouped together into a superstate.

Diagram:

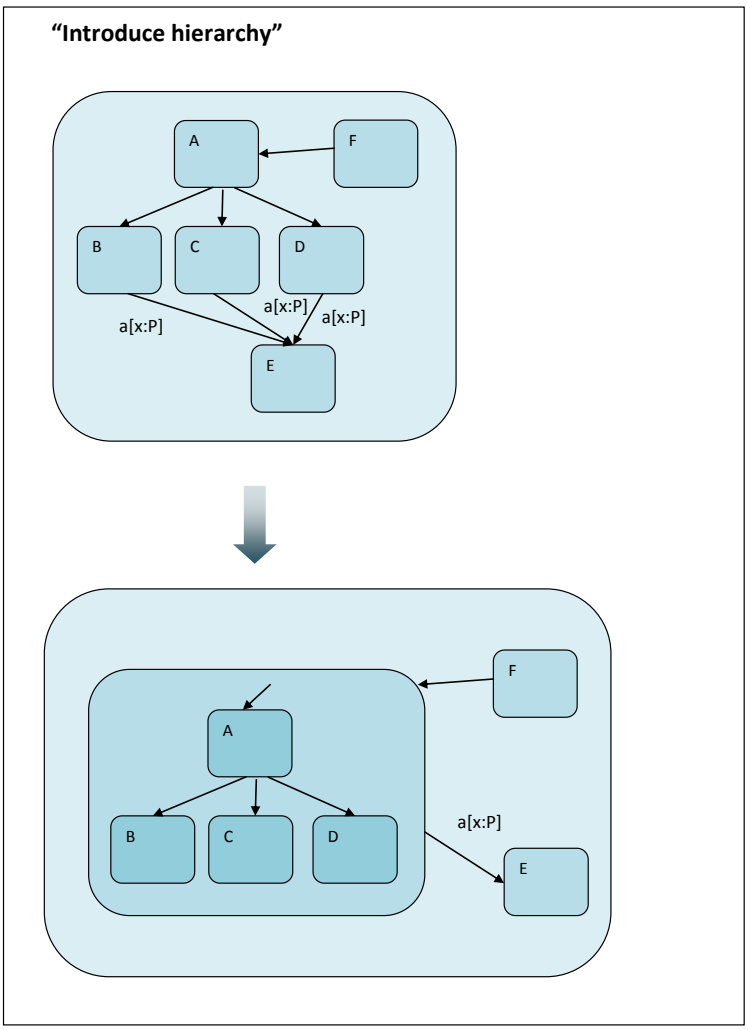

Figure 8.1: Introduce hierarchy

Name: *Remove redundant transition*, as an alternative transition exists.

Type: Refinement.

Rationale: The purpose of the pattern is to simplify a design by removing transitions that are redundant.

Constraints: It must be possible to strengthen the guard of the transition to *False*.

Parameters: A transition.

Transformation: This is a compound pattern consisting of the sequential composition of the following patterns, *Strengthen guard* to *False* followed by *Remove transition with a False guard*. The transition is removed from the model.

Introduce hierarchy and *Remove hierarchy* are compound refactoring steps for state machines found in the literature. A common sequence of patterns to be applied during the case study is the application of an *Unfold* pattern that creates a contract with conjuncts, followed by the *Conjunction introduction* pattern, then the *If* or *TotEnable* patterns are applied to the conjuncts. This is then followed by the *Conjunction elimination* and *Redundant hierarchy* patterns. A more generic sequence of patterns evident in the case study consists of applying repetitions of *Unfold Always* terminating with the application of the *Reroute* pattern.

Further research could be conducted to identify candidates for new compound patterns. The aim also would be for the user to specify their own compound patterns. The correctness of a compound pattern is guaranteed subject to the correctness of its component patterns. However, the number of sequences of patterns increases exponentially with the length of the sequence. This may lead to an unmanageably large number of compound patterns. Therefore further research is required to explore different ways of defining and invoking compound patterns and establishing more appropriate ways of using the compound pattern concept to reduce the number of compound patterns that need to be defined.

One promising approach, which reflects a common usage scenario as evidenced in the case study, is to partition compound patterns into a preprocessing step, a post-processing step and an intermediate step, which is where another sequence of patterns or compound pattern is applied. This structure supports a nesting of patterns where the outer nestings (preprocessing and post-processing steps) are predefined sequences of patterns. The intermediate step allows the user to select the next sequence of patterns.

A common scenario in the case study was to introduce a conjunction of states (pre-processing step), refine each into designs (intermediate step), then at the end the process is wrapped up by recombining the conjunction states (post-processing step).

The post-processing step may consist of a sequence of patterns that finalise the initial sequence of compounds in reverse order one by one until the transformation is complete. The user could guide the finalisation steps by confirming each one has completed successfully. User interaction could be incorporated into compound patterns in a number of different ways. For example, compounds could have an optional post-process which the user needs to confirm or a set of alternative processes from which the user selects.

8.3.3 Prove correctness of patterns

There is further work required to prove the correctness of patterns and validate them against their formal semantics. Proving the correctness of a pattern is not the only way of ensuring it is safe to use however. For example, the model checker could verify the patterns on each application in lieu of a single, upfront, correctness proof.

8.3.4 Structuring the pattern space

An interesting related area of research concerns novel approaches to structuring the pattern space in a more intuitive and navigable way, to aid engineers when querying the pattern catalogue during the design process. For example to support pattern matching on the patterns in the catalogue, to answer questions such as:

1. Which patterns transform a model like $model1$?
2. Which patterns transform any model into a model like $model1$?
3. Which patterns transform a model like $model1$ into a model like $model2$?

8.3.5 Tactics

Further research could be conducted to investigate automating pattern application in a way akin to tactics in theorem provers. Tactics are strategies used in proof, they decide the order to apply inference rules to discharge a proof obligation based on its structure [186]. In a similar vein a *tactic* for designing Contractual State Machines could suggest which patterns would be most appropriate to apply given the form of the contract and form of the required Contractual State Machine design. The intention is that by identifying

these strategies, documenting them as tactics and using them as single transformation rules would reduce the time and effort required during the design process. Tactics could prioritise the order in which patterns are considered applicable, building a goal-oriented layer on top of the existing pattern technology. An automated design engine which applies patterns repeatedly to a contract in order to produce a design based on up front information provided by the user would greatly reduce effort.

8.3.6 Extend software tools

Another possible direction for further work would be to improve scalability of the approach. One of the scalability issues is to allow the kinds of composite patterns discussed in the previous section to be implemented. The ability to combine existing transformations into new composite ones would be essential to support real developments. Future work would therefore include providing the infrastructure to enable the user to compose arbitrary patterns in this way and also devising an initial catalogue of composite patterns based on experience with the case study.

The software developed for this research could be extended in a number of other ways with additional functionality to implement new tools, most importantly for simulation. New tools could also be added to offer more control over the engineering process and support features such as versioning and differencing of models. More elaborate support for *undo* and *redo* transformations as part of the pattern application tool would permit different development paths to be explored. Other features that could also be included in the future are support for traceability and change propagation, to maintain an explicit link between source and target models, and a history of transformations. One benefit of being disciplined in design is that the traceability relationships from design to requirement should be easier to identify and use.

8.4 Summary

The thesis has demonstrated that a model-based approach to refinement of designs for reactive systems is feasible in practice. Closely related research to-date has achieved refinement calculi for restricted forms of state machines with an underlying formal theory [166, 218, 221]. The research in this thesis is an incremental improvement on what has been achieved so far as it considers refinement for a heterogeneous, data-rich language

that combines contracts with state machines. Our approach provides the engineer with the ability, (through contracts), to express high-level properties of a design (e.g. safety properties) and abstract functional behaviour. Patterns support a top-down design process from a contract to a detailed operational design that is guaranteed to preserve the properties of the original specification. This research provides refinement and refactoring patterns from contracts to contracts and contracts to mixed designs as well as between state machine designs for the data-rich, shared variable language.

The research in this thesis is an evolution on research to-date by expressing refinement/refactoring steps as patterns in terms of model transformations. Patterns lead to the possibility of further reducing the development and proof effort through automating the model transformations and the discharge of side-conditions and integration of the software with model checking technology.

Appendix A

Implementation

This appendix contains examples of the code for the software developed for this thesis and described in chapter 6 to implement the tool support for patterns and the design of Contractual State Machines. The full set of code for the design tool and wizards can be found at `https://svn.cs.york.ac.uk/extsvn/sosym/costa-wizards`.

A.1 Contractual State Machine metamodel

The Emfatic metamodel description for the contractual state machine language extended with Eugenia annotations.

```
1  @namespace(uri="sm1", prefix="sm1")
2  @gmf(foo="bar")
3  package sm1;
4  import "platform:/resource/sm1/model/tp.ecore";
5  import "platform:/resource/sm1/model/cp.ecore";
6  enum StateType {
7   sequentialState;
8   parallelState;
9   conjState;
10 }
11 @gmf.diagram(foo="bar")
12 class Statemachine {
13  !ordered val State[*] states;
14  !ordered val Transition[*] transitions;
15 }
16 @gmf.node(border.style = "dash")
17 class conjState extends State {
18 }
19 @gmf.node(label = "name, innercontract", label.pattern = "{0}:{1}",
20 label.icon = "false")
21 class State {
22  attr String name;
23  attr StateType type;
24  attr String innercontract;
25  derived volatile transient ref cp.Formula parsedinnercontract;
```

```
26   @gmf.affixed(foo="bar")
27   !ordered val Outercontract outercontract;
28   @gmf.affixed(foo="bar")
29   !ordered val DefaultTransition[*] defaulttransitions;
30   !ordered ref Transition[*]#source outgoing;
31   !ordered ref Transition[*]#target incoming;
32   @gmf.compartment(foo="bar")
33   !ordered val State[*]#parent substates;
34   !ordered ref State#substates parent;
35  }
36  @gmf.node(size="50,29", label="contract", label.icon="false")
37  class Outercontract {
38   attr String contract;
39   derived volatile transient ref cp.Formula parsedoutercontract;
40   @gmf.affixed(foo="bar")
41   !ordered val DefaultTransition[*] defaulttransitions;
42  }
43  @gmf.node(figure="sm1.figures.ArrowFigure", size="20,20", label.icon="false",
44  label="label", label.placement="external")
45  class DefaultTransition {
46   attr String label;
47  }
48  @gmf.link(label="label", source="source", target="target",
49  target.decoration="arrow", width = "1", color = "0,0,255")
50  class Transition {
51    attr String label;
52    derived volatile transient ref tp.TransitionExpression parsedtransition;
53    !ordered ref State#outgoing source;
54    !ordered ref State#incoming target;
55  }
```

Listing A.1: Metamodel specification in Emfatic with Eugenia annotations - file sm1.emf

A.2 EMFtext parser for the contract language

The Emfatic description required by EMFText of the Ecore metamodel for the abstract syntax of the CoSta contract language.

```
1  @namespace(uri="cp", prefix="cp")
2  package cp;
3  abstract class Formula {
4  }
5  class Constant extends Predicate {
6   attr Constants value;
7  }
8  class ConConstant extends Formula {
9   attr Constants value;
10 }
11 class ConAnd extends Formula {
12  val Formula[+] children;
13 }
14 class ConOr extends Formula {
15  val Formula[+] children;
16 }
17 class Unless extends Formula {
18  val Formula[+] children;
```

```
19 }
20 class Always extends Formula {
21  val Formula[1] body;
22 }
23 class If extends Formula {
24  val EventsAndData[1] eventsanddata;
25  val Formula[1] body;
26 }
27 class TotEnable extends Formula {
28  val EventsAndData[1] eventsanddata;
29 }
30 class ExEnable extends Formula {
31  val EventsAndData[1] eventsanddata;
32 }
33 class Disable extends Formula {
34  val EventsAndData[1] eventsanddata;
35 }
36 class ConNested extends Formula {
37  val Formula[1] body;
38 }
39 class Diamond extends Formula {
40  attr Integer k;
41  val Formula[1] body;
42 }
43 abstract class EventsAndData {
44 }
45 class EventFormula {
46 }
47 class NegatedEventList extends EventFormula {
48   val Event[*] events;
49 }
50 class Underscore extends EventFormula {
51 }
52 class EventList extends EventFormula {
53  val Event[*] events;
54 }
55 class Event {
56 }
57 class UnPrimedEvent extends Event {
58  attr String name;
59 }
60 class PrimedEvent extends Event {
61  attr String name;
62 }
63 class EventsAndDataAssignment extends EventsAndData {
64  val EventFormula[1] eventformula;
65  val AssignmentList[1] assignmentlist;
66 }
67 class EventsAndDataEvents extends EventsAndData {
68  val EventFormula[1] eventformula;
69 }
70 class EventsAndDataPredicateAssignment extends EventsAndData {
71  val EventFormula[1] eventformula;
72  val Predicate[1] predicate;
73  val AssignmentList[1] assignmentlist;
74 }
75 class EventsAndDataPredicate extends EventsAndData {
76  val EventFormula[1] eventformula;
77  val Predicate[1] predicate;
```

```
78  }
79  class EventsAndDataPredicateVars extends EventsAndData {
80    val EventFormula[1] eventformula;
81    val VariableList[1] variablelist;
82    val Predicate[1] predicate;
83  }
84  class VariableList {
85    val Variable[*] variables;
86  }
87  abstract class Expression {
88  }
89  class Additive extends Expression {
90    val Expression[1] left;
91    attr String operator;
92    val Expression[1] right;
93  }
94  class Multiplicative extends Expression {
95    val Expression[1] left;
96    attr String operator;
97    val Expression[1] right;
98  }
99  class Negation extends Expression {
100   attr String[1] operator;
101   val Expression[1] body;
102 }
103 class IntegerLiteralExp extends Expression {
104   attr int intValue;
105 }
106 class RealLiteralExp extends Expression {
107   attr float floatValue;
108 }
109 class BracketExp extends Expression {
110   val Expression body;
111 }
112 class ExpVariable extends Expression {
113 }
114 class Variable {
115 }
116 class PrimedVariable extends Variable {
117   attr String name;
118 }
119 class UnPrimedVariable extends Variable {
120   attr String name;
121 }
122 class PrimedExpVariable extends ExpVariable {
123   attr String name;
124 }
125 class UnPrimedExpVariable extends ExpVariable {
126   attr String name;
127 }
128 class Assignment {
129   val Variable[1] variable;
130   val Expression[1] expression;
131 }
132 class AssignmentList {
133 }
134 class Assignments extends AssignmentList {
135   val Assignment[*] assignments;
136 }
```

```
137 class Skip extends AssignmentList {
138 }
139 enum Constants {
140   TRUE;
141   FALSE;
142 }
143 abstract class Predicate {
144 }
145 class And extends Predicate {
146   val Predicate[+] children;
147 }
148 class Or extends Predicate {
149   val Predicate[+] children;
150 }
151 class Nested extends Predicate {
152   val Predicate[1] body;
153 }
154 class PredNegation extends Predicate {
155   val Predicate[1] body;
156 }
157 class Relation extends Predicate {
158   val Expression[1] left;
159   val Expression[1] right;
160   attr String operator;
161 }
```

Listing A.2: EMFtext Emfatic specification for abstract syntax of CoSta contract language - file cp.emf

The concrete syntax specification required by EMFtext for the CoSta contract language.

```
1  SYNTAXDEF cp
2  FOR     <cp>
3  START   Formula
4
5  OPTIONS {
6    forceEOF = "true";
7    reloadGeneratorModel = "true";
8    generateCodeFromGeneratorModel = "true";
9    tokenspace = "1";
10   usePredefinedTokens = "false";
11 }
12
13 TOKENS {
14   DEFINE CHAR $('a'..'z')('a'..'z'|'0'..'9')*$;
15   DEFINE CAP $('A'..'Z')('A'..'Z')*$;
16   DEFINE ADDITIVE_OPERATOR $ '+' | '-'$;
17   DEFINE MULTIPLICATIVE_OPERATOR $ '*' | '/' $;
18   DEFINE REL_OP $('<'|'<='|'=='|'!='|'>'|'>=')$;
19   DEFINE INTEGER_LITERAL $('1'..'9') ('0'..'9')* | '0'$;
20   DEFINE REAL_LITERAL $ (('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ (('e'|'E') (
         +'|'-')? ('0'..'9')*)?$;
21   DEFINE WHITESPACE $(' '|'\t'|'\f')$;
22   DEFINE LINEBREAKS $('\r\n'|'\r'|'\n')$;
23 }
24
25 RULES {
```

```
26      @Operator(type="binary_left_associative",weight="2", superclass="Formula")
27      ConAnd ::= children "And" children;
28
29      @Operator(type="binary_left_associative",weight="1", superclass="Formula")
30      ConOr ::= children "Or" children;
31
32      @Operator(type="binary_left_associative",weight="1", superclass="Formula")
33      Unless ::= children "Unless" children;
34
35      @Operator(type="unary_prefix", weight="3", superclass="Formula")
36      If ::= "If" eventsanddata body;
37
38      @Operator(type="primitive", weight="5", superclass="Formula")
39      ConNested ::= "(" body ")";
40
41      @Operator(type="unary_prefix", weight="4", superclass="Formula")
42      Always ::= "Always" body;
43
44      @Operator(type="unary_prefix", weight="3", superclass="Formula")
45      Diamond ::= "Diamond" k[INTEGER_LITERAL] body;
46
47      @Operator(type="primitive", weight="5", superclass="Formula")
48      TotEnable ::= "TotEnable" eventsanddata;
49
50      @Operator(type="primitive", weight="5", superclass="Formula")
51      ExEnable ::= "Ex" eventsanddata;
52
53      @Operator(type="primitive", weight="5", superclass="Formula")
54      Disable ::= "Dis" eventsanddata;
55
56      @Operator(type="primitive", weight="5", superclass="Formula")
57      ConConstant ::= value[CAP];
58
59      EventsAndDataPredicate ::= eventformula "[" predicate "]";
60      EventsAndDataPredicateVars ::= eventformula "[" variablelist ":" predicate "]";
61      EventsAndDataPredicateAssignment ::= eventformula "[" predicate "==>"
            assignmentlist "]";
62      EventsAndDataAssignment ::= eventformula "[" assignmentlist "]";
63      EventsAndDataEvents ::= eventformula;
64      NegatedEventList ::= ""events ("," events)*;
65      Underscore ::= "_";
66      EventList ::= events ("," events)*;
67      UnPrimedEvent ::= name[CHAR];
68      PrimedEvent ::= "'"name[CHAR];
69
70      @Operator(type="binary_left_associative",weight="2", superclass="Predicate")
71      And ::= children "&&" children;
72
73      @Operator(type="binary_left_associative",weight="1", superclass="Predicate")
74      Or ::= children "||" children;
75
76      @Operator(type="unary_prefix", weight="3", superclass="Predicate")
77      PredNegation ::= "!" body;
78
79      @Operator(type="primitive", weight="4", superclass="Predicate")
80      Nested ::= "(" body ")";
81
82      @Operator(type="primitive", weight="4", superclass="Predicate")
83      Constant ::= value[CAP];
```

```
84
85      @Operator(type="primitive", weight="4", superclass="Predicate")
86      Relation ::= left operator[REL_OP] right;
87
88      VariableList ::= (variables)? ("," variables)*;
89      Skip ::= "skip";
90      Assignments ::= assignments ("," assignments)*;
91      Assignment ::= variable "=" expression;
92
93      @Operator(type="binary_left_associative", weight="1", superclass="Expression")
94      Additive ::= left (operator[ADDITIVE_OPERATOR]) right;
95
96      @Operator(type="binary_left_associative", weight="2", superclass="Expression")
97      Multiplicative ::= left operator[MULTIPLICATIVE_OPERATOR] right;
98
99      @Operator(type="unary_prefix", weight="3", superclass="Expression")
100     Negation ::= operator[ADDITIVE_OPERATOR] body;
101
102     @Operator(type="primitive", weight="4", superclass="Expression")
103     BracketExp ::= "(" body ")";
104
105     @Operator(type="primitive", weight="4", superclass="Expression")
106     IntegerLiteralExp ::= intValue[INTEGER_LITERAL];
107
108     @Operator(type="primitive", weight="4", superclass="Expression")
109     RealLiteralExp ::= floatValue[REAL_LITERAL];
110
111     @Operator(type="primitive", weight="4", superclass="Expression")
112     UnPrimedExpVariable ::= name[CHAR];
113
114     @Operator(type="primitive", weight="4", superclass="Expression")
115     PrimedExpVariable ::= name[CHAR]"'";
116
117     PrimedVariable ::= name[CHAR]"'";
118     UnPrimedVariable ::= name[CHAR];
119 }
```

Listing A.3: EMFtext CS specification for the concrete syntax description of the CoSta contract language - file cp.cs

A.3 EMFtext parser for the transition label language

The concrete syntax definition in EMFtext for the CoSta transition label language.

```
1  SYNTAXDEF tp
2  FOR     <tp>
3  START   TransitionExpression
4
5  OPTIONS {
6    forceEOF = "true";
7    reloadGeneratorModel = "true";
8    generateCodeFromGeneratorModel = "true";
9    tokenspace = "1";
10   usePredefinedTokens = "false";
11 }
12
```

```
13 TOKENS {
14   DEFINE CHAR $('a'..'z')('a'..'z'|'0'..'9')*$;
15   DEFINE CAP $('A'..'Z')('A'..'Z')*$;
16   DEFINE ADDITIVE_OPERATOR $ '+' | '-'$;
17   DEFINE MULTIPLICATIVE_OPERATOR $ '*' | '/' $;
18   DEFINE REL_OP $('<'|'<='|'=='|'!='|'>'|'>=')$;
19   DEFINE INTEGER_LITERAL $('1'..'9') ('0'..'9')* | '0'$;
20   DEFINE REAL_LITERAL $ (('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ (('e'|'E') ('
         +'|'-')? ('0'..'9')*)?$;
21   DEFINE WHITESPACE $(' '|'\t'|'\f')$;
22   DEFINE LINEBREAKS $('\r\n'|'\r'|'\n')$;
23 }
24
25 RULES {
26    TransitionExprPredicate ::= action "[" predicate "]";
27    TransitionExprPredicateVars ::= action "[" variablelist ":" predicate "]";
28    TransitionExprPredicateAssignment ::= action "[" predicate "==>" assignmentlist
          "]";
29    TransitionExprAssignment ::= action "[" assignmentlist "]";
30    TransitionExpression ::= action;
31    UnPrimedAction ::= name[CHAR];
32    PrimedAction ::= "'"name[CHAR];
33    TauAction ::= "tau";
34
35    @Operator(type="binary_left_associative",weight="2", superclass="TPredicate")t
36    TAnd ::= children "&&" children;
37
38    @Operator(type="binary_left_associative",weight="1", superclass="TPredicate")
39    TOr ::= children "||" children;
40
41    @Operator(type="unary_prefix", weight="3", superclass="TPredicate")
42    TPredNegation ::= "!" body;
43
44    @Operator(type="primitive", weight="4", superclass="TPredicate")
45    TNested ::= "(" body ")";
46
47    @Operator(type="primitive", weight="4", superclass="TPredicate")
48    TConstant ::= value[CAP];
49
50    @Operator(type="primitive", weight="4", superclass="TPredicate")
51    TRelation ::= left operator[REL_OP] right;
52
53    TVariableList ::= (variables)? ("," variables)*;
54    TSkip ::= "skip";
55    TAssignments ::= assignments ("," assignments)*;
56    TAssignment ::= variable "=" expression;
57
58    @Operator(type="binary_left_associative", weight="1", superclass="TExpression")
59    TAdditive ::= left (operator[ADDITIVE_OPERATOR]) right;
60
61    @Operator(type="binary_left_associative", weight="2", superclass="TExpression")
62    TMultiplicative ::= left operator[MULTIPLICATIVE_OPERATOR] right;
63
64    @Operator(type="unary_prefix", weight="3", superclass="TExpression")
65    TNegation ::= operator[ADDITIVE_OPERATOR] body;
66
67    @Operator(type="primitive", weight="4", superclass="TExpression")
68    TBracketExp ::= "(" body ")";
69
```

```
70      @Operator(type="primitive", weight="4", superclass="TExpression")
71      TIntegerLiteralExp ::= intValue[INTEGER_LITERAL];
72
73      @Operator(type="primitive", weight="4", superclass="TExpression")
74      TRealLiteralExp ::= floatValue[REAL_LITERAL];
75
76      @Operator(type="primitive", weight="4", superclass="TExpression")
77      TUnPrimedExpVariable ::= name[CHAR];
78
79      @Operator(type="primitive", weight="4", superclass="TExpression")
80      TPrimedExpVariable ::= name[CHAR]"'";
81
82      TPrimedVariable ::= name[CHAR]"'";
83      TUnPrimedVariable ::= name[CHAR];
84
85  }
```

Listing A.4: EMFtext CS specification for the transition label language - file tp.cs

The Emfatic abstract syntax description required by EMFtext for the CoSta transition label language.

```
1  @namespace(uri="tp", prefix="tp")
2  package tp;
3  class TransitionExpression {
4    val Action[1] action;
5  }
6  class Action {
7  }
8  class UnPrimedAction extends Action {
9    attr String name;
10 }
11 class PrimedAction extends Action {
12   attr String name;
13 }
14 class TauAction extends Action {
15 }
16 class TransitionExprAssignment extends TransitionExpression {
17   val TAssignmentList[1] assignmentlist;
18 }
19 class TransitionExprPredicateAssignment extends TransitionExpression {
20   val TAssignmentList[1] assignmentlist;
21   val TPredicate[1] predicate;
22 }
23 class TransitionExprPredicate extends TransitionExpression {
24   val TPredicate[1] predicate;
25 }
26 class TransitionExprPredicateVars extends TransitionExpression {
27   val TPredicate[1] predicate;
28   val TVariableList[1] variablelist;
29 }
30 class TVariableList {
31   val TVariable[*] variables;
32 }
33 abstract class TExpression {
34 }
35 class TAdditive extends TExpression {
36  val TExpression[1] left;
```

```
37    attr String operator;
38    val TExpression[1] right;
39  }
40  class TMultiplicative extends TExpression {
41    val TExpression[1] left;
42    attr String operator;
43    val TExpression[1] right;
44  }
45  class TNegation extends TExpression {
46    attr String[1] operator;
47    val TExpression[1] body;
48  }
49  class TIntegerLiteralExp extends TExpression {
50    attr int intValue;
51  }
52  class TRealLiteralExp extends TExpression {
53    attr float floatValue;
54  }
55  class TBracketExp extends TExpression {
56    val TExpression body;
57  }
58  class TExpVariable extends TExpression {
59  }
60  class TVariable {
61  }
62  class TPrimedVariable extends TVariable {
63     attr String name;
64  }
65  class TUnPrimedVariable extends TVariable {
66     attr String name;
67  }
68  class TPrimedExpVariable extends TExpVariable {
69     attr String name;
70  }
71  class TUnPrimedExpVariable extends TExpVariable {
72     attr String name;
73  }
74  class TAssignment {
75     val TVariable[1] variable;
76     val TExpression[1] expression;
77  }
78  class TAssignmentList {
79  }
80  class TAssignments extends TAssignmentList {
81     val TAssignment[*] assignments;
82  }
83  class TSkip extends TAssignmentList {
84  }
85  enum TConstants {
86    TRUE;
87      FALSE;
88  }
89  class TConstant extends TPredicate {
90     attr TConstants value;
91  }
92  abstract class TPredicate {
93  }
94  class TAnd extends TPredicate {
95    val TPredicate[+] children;
```

```
96  }
97  class TOr extends TPredicate {
98    val TPredicate[+] children;
99  }
100 class TNested extends TPredicate {
101   val TPredicate[1] body;
102 }
103 class TPredNegation extends TPredicate {
104   val TPredicate[1] body;
105 }
106 class TRelation extends TPredicate {
107   val TExpression[1] left;
108   val TExpression[1] right;
109   attr String operator;
110 }
```

Listing A.5: EMFtext Emfatic specification for the CoSta transition label language - file tp.emf

A.4 EWL wizards for refinement patterns

This appendix contains some examples of the EWL wizards that implement the refinement and refactoring patterns presented in Chapters 4 and 5. The model checker supports evaluation of quantifier-free conjectures on data. Patterns with more complex side-conditions (e.g the TotEnable pattern may require a conjecture with quantifiers to be verified) may rely on inspection for verification as the functionality required to implement these patterns fully was not available via the model-checker.

A.4.1 UnfoldAlways wizard

The UnfoldAlways wizard is applicable if the selected component is a state whose inner contract is an *Always* operator expression. The pattern checks that the contract for the selected state has the form "*Always* ϕ" and the wizard unfolds the contract to "ϕ *And (If* _ *(Always* ϕ*))*". The EMFtext generated parser is invoked and parses the inner contract to an EMF model. Relevant parts of the EMF model are converted to a textual representation using the EMFtext generated printer for the contract language and used to construct a new contract of the required form. A new substate is created and the new contract is assigned to its inner contract.

```
1  wizard wizUnfoldAlways {
2  guard : self.isKindOf(State)
3  and (self.parsedinnercontract.isTypeOf(Always)
4  or (self.parsedinnercontract.isTypeOf(ConNested) and self.parsedinnercontract.body.
       isTypeOf(Always)))
5  title : "Unfold always"
6  do {
7  --declare a reference to the Epsilon tool, printlanguageTool, then can call any of
       its routines.
8    var printlanguageTool : new Native("printlanguage.printlanguageTool");
9    var newContract:String;
10   var pc1:Formula;
11   var pc2:Formula;
12   var substate:State;
13   if (UserInput.confirm('Unfold always' + ' ?', true))
14   {
15        pc1 := self.parsedinnercontract;
16       while (pc1.isTypeOf(ConNested))
17       pc1 := pc1.body;
18         pc2 := pc1.body;
19       while (pc2.isTypeOf(ConNested))
20       pc2 := pc2.body;
21   --calls the routine from the Epsilon tool to generate a textual representation of
       the
22   --contract language from its EMF model
23       newContract:="(".concat(printlanguageTool.printcp(pc2));
24       newContract:=
```

```
25      newContract.concat(") And (If _ (").concat(printlanguageTool.printcp(pc1).
            concat("))"));
26      substate := State.createInstance();
27      substate.innercontract := newContract;
28      self.substates.add(substate);
29      forceRefresh();
30    }
31    }
32  }
33  operation forceRefresh() {
34        var s : new State;
35        Statemachine.all.first.states.add(s);
36        delete s;
37  }
```

Listing A.6: Wizard to unfold the Always operator

A.4.2 TotEnable wizard

The TotEnable wizard refines a *TotEnable* contract to a state machine model. The TotEnable wizard is applicable if the selected component is a state. The pattern checks that the state's inner contract has the form "*TotEnable eventFormula*" or "*TotEnable eventFormula [updateExpression]*". *EventFormula* can be a set of events, a negated set of events or the underscore character.

The wizard introduces a set of substates for the selected component consisting of a start state with outgoing transitions created by the user that each target a new state with the inner contract *True*. The wizard prompts the user for the new transitions. The pattern checks that the set of events provided as parameters and the set of enabled events in the contract are not disjoint.

The state's inner contract is parsed to an EMF model. Similarly the transition labels that are input by the user are parsed to EMF models by the EMFtext generated parser for the transition language. The parse trees permit easier manipulation and comparison of the contract and transition labels.

Permitted refinements to the update expression are to weaken the guard or strengthen the update within the guard. If no update expression is specified in the contract, new transitions with some update from some source data state are possible. If the update is defined as *skip* for the contract, transitions must ensure values remain the same and no update is possible. If an update expression $[x : P]$ is specified in the contract, transitions with an update $[x : P']$ consistent with and from a source state satisfying $[x : P]$ are possible. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract.

Checks based on syntax are performed initially. To determine if the transition satisfies the contract the event must be an event included in the contract's *eventlist*. If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the transition's update expression is syntactically equivalent to the contract's update expression then the transition satisfies the contract. If the contract is of the form $\langle[eventlist]\rangle$ it is satisfied by transitions of the form *event*, $event[assignmentlist]$, $event[skip]$. If the contract has the form $\langle[eventlist[skip]]\rangle$ or $\langle[eventlist[G \Longrightarrow skip]]\rangle$ it is satisfied by transitions of the form *event*, $event[skip]$. Otherwise a more detailed check is required to determine if the transition satisfies the contract. To ensure that the update expressions are consistent the following check is required.

The side-condition is $\vdash ((\exists x' : P) \Rightarrow (\exists x' : P')) \wedge ((\exists x' : P) \wedge P' \Rightarrow P)$

Currently this check is verified by inspection as it requires the evaluation of a conjecture with quantifiers on data and this functionality is not yet supported by HST.

```
1  wizard wizEnable {
2  guard : self.isKindOf(State)
3  and (self.parsedinnercontract.isTypeOf(TotEnable)
4  or (self.parsedinnercontract.isTypeOf(ConNested) and self.parsedinnercontract.body.
       isTypeOf(TotEnable)))
5  title : "Refine contract TotEnable to state machine"
6  do {
7    var s1:String;
8    var parameters:Bag;
9    var events:Bag;
10   var pc:Formula;
11   var te:TransitionExpression;
12   var len1:Integer;
13   var i:Integer;
14   var j:Integer;
15   var found:Boolean;
16   var inupdate:Boolean;
17   var event:String;
18   var intersect:Boolean;
19   var substate1 : new State;
20   var substate2 : State;
21   var defaulttransition : new DefaultTransition;
22   var transition : Transition;
23   var sm : Statemachine;
24   var updateExpr1 : String;
25   var updateExpr2 : String;
26   var allvars:Bag;
27   if (UserInput.confirm('Refine contract Enable to stga' + ' ?', true))
28   {
29     pc := self.parsedinnercontract;
30     while (pc.isTypeOf(ConNested))
```

```
31      pc := pc.body;
32     if (pc.eventsanddata.eventformula.isTypeOf(EventList) or
33                 pc.eventsanddata.eventformula.isTypeOf(NegatedEventList))
34     {
35       for (e1 in pc.eventsanddata.eventformula.events)
36       {
37        if (e1.isTypeOf(PrimedEvent)) events.add('\''+e1.name);
38        else events.add(e1.name);
39        }
40     }
41      s1 := UserInput.prompt('Enter required transitions');
42     --Create bag of parameters
43     --length gives the number of characters
44     len1 := s1.length();
45     i := 0;
46     j:= 0;
47     --strings are indexed from 0 to length-1
48     while (i < len1)
49     {
50       found:=false;
51       inupdate:=false;
52       while ((j < len1) and not found)
53       {
54        if ((s1.charAt(j).toString == ",") and (not inupdate)) found := true;
55        if (s1.charAt(j).toString == "[") inupdate := true;
56        if (s1.charAt(j).toString == "]") inupdate := false;
57        if (not found) j := j + 1;
58       }
59       --substring(i,j) will give characters from and including i to j-1
60       if (j==len1) event := s1.substring(i).remBlanks();
61       else event := s1.substring(i,j).remBlanks();
62       if (found) j := j+1;
63       i := j;
64       parameters.add(event);
65      }
66      if (parameters.size() > 0)
67      {
68       updateExpr1 := pc.eventsanddata.data2String().strip();
69       updateExpr1 := updateExpr1.contract2CommonFormat(allvars, pc);
70       updateExpr1 := "(".concat(updateExpr1.substring(1,updateExpr1.length()-1)).
             concat(")");
71       --Check the intersection of events and pararameters is not the empty list
72       intersect := false;
73       for (e1 in parameters)
74       {
75        updateExpr2 := e1.updexp().strip();
76        te := e1.parseTransitionExpression();
77        updateExpr2 := updateExpr2.parameter2CommonFormat(allvars, te);
78        updateExpr2 := "(".concat(updateExpr2.substring(1,updateExpr2.length()-1)).
              concat(")");
79        if (pc.eventsanddata.eventformula.isTypeOf(Underscore))
80        {
81        if ((updateExpr2 == updateExpr1) or
82           (pc.eventsanddata.isTypeOf(EventsAndDataEvents) and
83             (te.isTypeOf(TransitionExpression) or
84              (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                   TAssignments)) or
85              (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                   TSkip)))) or
```

```
86            ((pc.eventsanddata.isTypeOf(EventsAndDataAssignment) and pc.eventsanddata
                  .assignmentlist.isTypeOf(Skip)) and
87             (te.isTypeOf(TransitionExpression) or
88            (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                  TSkip)))) or
89            ((pc.eventsanddata.isTypeOf(EventsAndDataPredicateAssignment) and pc.
                  eventsanddata.assignmentlist.isTypeOf(Skip)) and
90             (te.isTypeOf(TransitionExpression) or
91            (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                  TSkip)))) or
92              acceptProofObligation(updateExpr1.implicitGuard(allvars),updateExpr2.
                    implicitGuard(allvars),updateExpr1,updateExpr2))
93          {
94             intersect := true;
95          }
96        }
97         for (e2 in events)
98         {
99         if ((pc.eventsanddata.eventformula.isTypeOf(EventList) and (e1.strip().ev()
               == e2)) or
100           (pc.eventsanddata.eventformula.isTypeOf(NegatedEventList) and (e1.strip()
                .ev() <> e2)))
101        {
102         if ((updateExpr2 == updateExpr1) or
103           (pc.eventsanddata.isTypeOf(EventsAndDataEvents) and
104            (te.isTypeOf(TransitionExpression) or
105             (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                    TAssignments)) or
106             (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                    TSkip)))) or
107            ((pc.eventsanddata.isTypeOf(EventsAndDataAssignment) and pc.eventsanddata
                  .assignmentlist.isTypeOf(Skip)) and
108             (te.isTypeOf(TransitionExpression) or
109            (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                  TSkip)))) or
110            ((pc.eventsanddata.isTypeOf(EventsAndDataPredicateAssignment) and pc.
                  eventsanddata.assignmentlist.isTypeOf(Skip)) and
111             (te.isTypeOf(TransitionExpression) or
112            (te.isTypeOf(TransitionExprAssignment) and te.assignmentlist.isTypeOf(
                  TSkip)))) or
113             acceptProofObligation(updateExpr1.implicitGuard(allvars),updateExpr2.
                    implicitGuard(allvars),updateExpr1,updateExpr2))
114          {
115             intersect := true;
116             break;
117          }
118         }
119        }
120         if (intersect) break;
121      }
122       if (intersect)
123      {
124      sm := Statemachine.allInstances().first;
125      self.substates.add(substate1);
126      defaulttransition.label := " ";
127      substate1.defaulttransitions.add(defaulttransition);
128      for (e3 in parameters)
129      {
130       substate2 := State.createInstance();
```

```
131         substate2.innercontract := "TRUE";
132         self.substates.add(substate2);
133         transition := Transition.createInstance();
134         transition.source := substate1;
135         transition.target := substate2;
136         transition.label := e3.toString();
137         sm.transitions.add(transition);
138       }
139       forceRefresh();
140       }
141       else UserInput.inform("Parameters must contain at least one event from the
              Enable contract");
142     }
143   }
144   }
145 }
146 operation TExpression texpression2String():String
147 {
148   if (self.isTypeOf(TIntegerLiteralExp)) return(self.intValue.toString());
149   else if (self.isTypeOf(TRealLiteralExp)) return(self.floatValue.toString());
150   else if (self.isKindOf(TExpVariable)) return(self.texpvariable2String());
151   else if (self.isTypeOf(TAdditive) or self.isTypeOf(TMultiplicative))
152     return(self.left.texpression2String().concat(self.operator).concat(self.right.
          texpression2String()));
153   else if (self.isTypeOf(TNegation)) return(self.operator.concat(self.body.
         texpression2String()));
154   else if (self.isTypeOf(TBracketExp)) return('('.concat(self.body.
         texpression2String().concat(')')));
155   else return("");
156 }
157 operation TVariable tvariable2String():String
158 {
159   if (self.isTypeOf(TPrimedVariable))
160     return(self.name.concat('\''));
161   else if (self.isTypeOf(TUnPrimedVariable)) return(self.name);
162       else return("");
163 }
164 operation String updexp():String
165 {
166   var s1:String;
167   var i:Integer;
168   var found:Boolean;
169   i:=0;
170   found:=false;
171   while ((i < self.length()) and (not found))
172   {
173     if (self.charAt(i).toString == "[") found:=true;
174     if (not found) i:=i+1;
175   }
176   if (found) s1 := self.substring(i);
177   else s1 := "";
178   return s1;
179 }
180 operation String implicitGuard(allvars):String
181 {
182   var len1:Integer;
183   var i:Integer;
184   var j:Integer;
185   var variable:String;
```

```
186   var vars:Bag;
187   var found:Boolean;
188   var end:Boolean;
189   var s1:String;
190   var s2:String;
191   var first:Boolean;
192       --Create bag of variables
193     --length gives the number of characters
194     len1 := self.length();
195     i := 1;
196     j:= 1;
197     end:=false;
198     --strings are indexed from 0 to length-1
199     while ((i < len1) and (not end))
200     {
201      found:=false;
202      while ((j < len1) and not found and not end)
203      {
204        if (self.charAt(j).toString == ":") end := true;
205        else if (self.charAt(j).toString == ",") found := true;
206        if (not found and not end) j := j + 1;
207      }
208      --substring(i,j) will give characters from and including i to j-1
209      if (j==len1) variable := self.substring(i).remBlanks();
210      else variable := self.substring(i,j).remBlanks();
211      if (found) j := j+1;
212      i := j;
213      vars.add(variable);
214     }
215     s1 := self.substring(0,j).remBlanks();
216     s2 := self.substring(j,len1-1).remBlanks();
217     first:=true;
218     for (v in allvars)
219     {
220       if (vars.excludes(v))
221       {
222         if (first)
223         {
224           if (s1 == "(")
225           {
226             s1:=s1.concat(v);
227           }
228           else s1:=s1.concat(",").concat(v);
229           first:=false;
230         }
231         else s1:=s1.concat(",").concat(v);
232         s2:=s2.concat("&&").concat(v).concat("'==").concat(v);
233       }
234     }
235    return("(".concat("2203".toUnicode()).concat(s1.substring(1).concat(s2).concat
           (")")));
236 }
237 operation String toUnicode() {
238 var unicode = Native("java.lang.Integer").parseInt(self, 16);
239 var chars = Native("java.lang.Character").toChars(unicode);
240 return Native("java.util.Arrays").toString(chars).substring(1,2);
241 }
242 operation TAssignmentList tassignmentListConvert2Predicate(allvars:Bag, p:String):
       String
```

```
243 {
244   var s1:String;
245   var s2:String;
246   var first:Boolean;
247     s2:="";
248     s1:="[";
249     first:=true;
250     if (self.isTypeOf(TAssignments))
251     {
252       for (a in self.assignments)
253       {
254         if (not first)
255         {
256           s1:=s1.concat(",");
257           s2:=s2.concat("&&");
258         }
259         s1:=s1.concat(a.variable.name);
260         s2:=s2.concat(a.variable.name);
261         s2:=s2.concat("'==");
262         s2:=s2.concat(a.expression.texpression2String());
263         if (first) first:=false;
264       }
265     }
266     else if (self.isTypeOf(TSkip))
267       {
268         first:=true;
269         for (v in allvars)
270         {
271           if (first) first:=false;
272           else
273           {
274             s1 := s1.concat(",");
275             s2 := s2.concat("&&");
276           }
277           s1 := s1.concat(v);
278           s2 := s2.concat(v);
279           s2:=s2.concat("'==");
280           s2 := s2.concat(v);
281         }
282       }
283     if (p <> "") s1:=s1.concat(":").concat(p).concat("&&").concat(s2).concat("]");
284     else s1:=s1.concat(":").concat(s2).concat("]");
285   return(s1);
286 }
287 operation String parameter2CommonFormat(allvars:Bag, tp:TransitionExpression):
        String
288 {
289   var tu:String;
290   var first:Boolean;
291   tu:="";
292   if (self == "")
293   {
294     tu:="[";
295     first:=true;
296     for (v in allvars)
297     {
298       if (first) first:=false;
299       else tu := tu.concat(",");
300       tu := tu.concat(v);
```

```
301     }
302     tu:=tu.concat(":");
303     first:=true;
304     for (v in allvars)
305     {
306       if (first) first:=false;
307       else tu := tu.concat("&&");
308       tu := tu.concat(v);
309       tu:=tu.concat("'==");
310       tu := tu.concat(v);
311     }
312     tu:=tu.concat("]");
313   }
314   else if (tp.isTypeOf(TransitionExprPredicate))
315   {
316     tu:="[";
317     first:=true;
318     for (v in allvars)
319     {
320       if (first) first:=false;
321       else tu := tu.concat(",");
322       tu := tu.concat(v);
323     }
324     tu:=tu.concat(":");
325     tu:=tu.concat(self.substring(1,self.length()-1));
326     tu:=tu.concat("&&");
327     first:=true;
328     for (v in allvars)
329     {
330       if (first) first:=false;
331       else tu := tu.concat("&&");
332       tu := tu.concat(v);
333       tu:=tu.concat("'==");
334       tu := tu.concat(v);
335     }
336     tu:=tu.concat("]");
337   }
338   else if (tp.isTypeOf(TransitionExprAssignment))
339   {
340     tu:=tp.assignmentlist.tassignmentListConvert2Predicate(allvars,"");
341   }
342   else if (tp.isTypeOf(TransitionExprPredicateAssignment))
343   {
344     tu:=tp.assignmentlist.tassignmentListConvert2Predicate(allvars,tp.predicate.
          tpredicate2String());
345   }
346   else tu:=self;
347   return(tu);
348 }
349 operation TExpVariable texpvariable2String():String
350 {
351   if (self.isTypeOf(TPrimedExpVariable))
352     return(self.name.concat('\''));
353   else if (self.isTypeOf(TUnPrimedExpVariable)) return(self.name);
354       else return("");
355 }
356 operation TPredicate tpredicate2String():String
357 {
358   if (self.isTypeOf(TConstant)) return(self.value.toString());
```

```
359   else if (self.isTypeOf(TNested)) return('('.concat(self.body.tpredicate2String().
          concat(')')));
360   else if (self.isTypeOf(TAnd))
361     return(self.children.first().tpredicate2String().concat(' && ').concat(self.
            children.last().tpredicate2String()));
362   else if (self.isTypeOf(TOr))
363     return(self.children.first().tpredicate2String().concat(' || ').concat(self.
            children.last().tpredicate2String()));
364   else if (self.isTypeOf(TRelation))
365     return(self.left.texpression2String().concat(self.operator).concat(self.right.
            texpression2String()));
366   else if (self.isTypeOf(TPredNegation)) return('!'.concat(self.body.
          tpredicate2String()));
367   else return "";
368 }
369 operation String contract2CommonFormat(allvars:Bag, pc:Formula):String
370 {
371   var cu:String;
372   var first:Boolean;
373   cu:="";
374   if (self == "")
375   {
376     cu:="[";
377     first:=true;
378     for (v in allvars)
379     {
380       if (first) first:=false;
381       else cu := cu.concat(",");
382       cu := cu.concat(v);
383     }
384     cu:=cu.concat(":TRUE]");
385   }
386   else if (pc.eventsanddata.isTypeOf(EventsAndDataPredicate))
387   {
388     cu:="[";
389     first:=true;
390     for (v in allvars)
391     {
392       if (first) first:=false;
393       else cu := cu.concat(",");
394       cu := cu.concat(v);
395     }
396     cu:=cu.concat(":");
397     cu:=cu.concat(self.substring(1,self.length()));
398   }
399   else if (pc.eventsanddata.isTypeOf(EventsAndDataAssignment))
400   {
401     cu:=pc.eventsanddata.assignmentlist.assignmentListConvert2Predicate(allvars,"");
402   }
403   else if (pc.eventsanddata.isTypeOf(EventsAndDataPredicateAssignment))
404   {
405     cu:=pc.eventsanddata.assignmentlist.assignmentListConvert2Predicate(allvars,pc.
            eventsanddata.predicate.predicate2String());
406   }
407   else cu:=self;
408   return cu;
409 }
410 operation AssignmentList assignmentListConvert2Predicate(allvars:Bag, p:String):
        String
```

```
411 {
412   var s1:String;
413   var s2:String;
414   var first:Boolean;
415     s2:="";
416     s1:="[";
417     first:=true;
418     if (self.isTypeOf(Assignments))
419     {
420       for (a in self.assignments)
421       {
422         if (not first)
423         {
424           s1:=s1.concat(",");
425           s2:=s2.concat("&&");
426         }
427         s1:=s1.concat(a.variable.name);
428         s2:=s2.concat(a.variable.name);
429         s2:=s2.concat("'==");
430         s2:=s2.concat(a.expression.expression2String());
431         if (first) first:=false;
432       }
433     }
434     else if (self.isTypeOf(Skip))
435       {
436         first:=true;
437         for (v in allvars)
438         {
439           if (first) first:=false;
440           else
441           {
442             s1 := s1.concat(",");
443             s2 := s2.concat("&&");
444           }
445           s1 := s1.concat(v);
446           s2 := s2.concat(v);
447           s2:=s2.concat("'==");
448           s2 := s2.concat(v);
449         }
450       }
451     if (p <> "") s1:=s1.concat(":").concat(p).concat("&&").concat(s2).concat("]");
452     else s1:=s1.concat(":").concat(s2).concat("]");
453   return(s1);
454 }
455 operation Predicate predicate2String():String
456 {
457   if (self.isTypeOf(Constant)) return(self.value.toString());
458   else if (self.isTypeOf(Nested)) return('('.concat(self.body.predicate2String().
        concat(')')));
459   else if (self.isTypeOf(And))
460     return(self.children.first().predicate2String().concat(' && ').concat(self.
          children.last().predicate2String()));
461   else if (self.isTypeOf(Or))
462     return(self.children.first().predicate2String().concat(' || ').concat(self.
          children.last().predicate2String()));
463   else if (self.isTypeOf(Relation))
464     return(self.left.expression2String().concat(self.operator).concat(self.right.
          expression2String()));
465   else if (self.isTypeOf(PredNegation)) return('!'.concat(self.body.predicate2String
```

```
        ()));
466   else return "";
467  }
468  operation VariableList variablelist2String():String
469  {
470   var s:String;
471   s:="";
472   for (v in self.variables)
473   {
474    if (s == "") s := v.variable2String();
475    else s := s.concat(',').concat(v.variable2String());
476   }
477   return(s);
478  }
479  operation Variable variable2String():String
480  {
481   if (self.isTypeOf(PrimedVariable))
482     return(self.name.concat('\''));
483   else if (self.isTypeOf(UnPrimedVariable)) return(self.name);
484       else return("");
485  }
486  operation ExpVariable expvariable2String():String
487  {
488   if (self.isTypeOf(PrimedExpVariable))
489     return(self.name.concat('\''));
490   else if (self.isTypeOf(UnPrimedExpVariable)) return(self.name);
491       else return("");
492  }
493  operation Expression expression2String():String
494  {
495   if (self.isTypeOf(IntegerLiteralExp)) return(self.intValue.toString());
496   else if (self.isTypeOf(RealLiteralExp)) return(self.floatValue.toString());
497   else if (self.isKindOf(ExpVariable)) return(self.expvariable2String());
498   else if (self.isTypeOf(Additive) or self.isTypeOf(Multiplicative))
499    return(self.left.expression2String().concat(self.operator).concat(self.right.
         expression2String()));
500   else if (self.isTypeOf(Negation)) return(self.operator.concat(self.body.
       expression2String()));
501   else if (self.isTypeOf(BracketExp)) return('('.concat(self.body.expression2String
       ().concat(')')));
502   else return("");
503  }
504  operation EventsAndData data2String():String
505  {
506   if (self.isTypeOf(EventsAndDataAssignment))
507     return("[".concat(self.assignmentlist.assignmentList2String()).concat("]"));
508   else if (self.isTypeOf(EventsAndDataPredicateAssignment))
509     return("[".concat(self.predicate.predicate2String()).concat("==>").concat(self.
         assignmentlist.assignmentList2String()).concat("]"));
510   else if (self.isTypeOf(EventsAndDataPredicate))
511     return("[".concat(self.predicate.predicate2String()).concat("]"));
512   else if (self.isTypeOf(EventsAndDataPredicateVars))
513   {
514    if (self.variablelist == null)
515       return("[".concat(":").concat(self.predicate.predicate2String()).concat("]"))
           ;
516    else
517       return("[".concat(self.variablelist.variablelist2String()).concat(":").concat
           (self.predicate.predicate2String()).concat("]"));
```

```
518   }
519   else if (self.isTypeOf(EventsAndDataEvents)) return("");
520  }
521  operation AssignmentList assignmentList2String():String
522  {
523   if (self.isTypeOf(Skip)) return("skip");
524   else return(self.assignments2String());
525  }
526  operation Assignments assignments2String():String
527  {
528   var s:String;
529   s:="";
530   for (a in self.assignments)
531   {
532    if (s == "") s := a.assignment2String();
533    else s := s.concat(',').concat(a.assignment2String());
534   }
535   return(s);
536  }
537  operation Assignment assignment2String():String
538  {
539   return(self.variable.variable2String().concat("=").concat(self.expression.
         expression2String()));
540  }
541  operation acceptProofObligation(oldGuard, newGuard, oldPost, newPost):Boolean
542  {
543   var accept:Boolean;
544   accept := UserInput.confirm("Do you agree the proof obligation holds : " +
         oldGuard + " => " + newGuard + " && (" + oldGuard + " && " + newPost + " => "
         + oldPost + ")", true);
545   return(accept);
546  }
547  operation String parseTransitionExpression():TransitionExpression
548  {
549   var transition:Transition;
550   var tp:TransitionExpression;
551   transition := Transition.createInstance();
552   transition.label := self;
553   tp := transition.parsedtransition;
554   delete transition;
555   return tp;
556  }
557  operation String remBlanks():String
558  {
559   var s1:String;
560   var i:Integer;
561   var pos:Integer;
562   var found:Boolean;
563   s1:=self;
564   found := false;
565   i:=0;
566   while ((i < s1.length()) and not found)
567   {
568    if (s1.charAt(i).toString <> " ")
569    {
570       pos:=i;
571       found:=true;
572    }
573    i:=i+1;
```

```
574   }
575   if (not found) return "";
576   else return s1.substring(pos);
577  }
578  operation String strip():String
579  {
580   var s1:String;
581   var i:Integer;
582   var j:Integer;
583   i:=0;
584   s1:="";
585   while (i < self.length())
586   {
587    if (self.charAt(i).toString <> " ")
588    {
589     s1:=s1.concat(self.charAt(i).toString);
590    }
591    i:=i+1;
592   }
593   return s1;
594  }
595  operation String ev():String
596  {
597   var s1:String;
598   var i:Integer;
599   var found:Boolean;
600   i:=0;
601   found:=false;
602   while ((i < self.length()) and (not found))
603   {
604    if (self.charAt(i).toString == "[") found:=true;
605    if (not found) i:=i+1;
606   }
607   if (found) s1 := self.substring(0,i);
608   else s1 := self;
609   return s1;
610  }
611  operation forceRefresh() {
612        var s : new State;
613        Statemachine.all.first.states.add(s);
614        delete s;
615  }
```

Listing A.7: Wizard to refine the TotEnable operator to a state machine

Appendix B

CoSta contract language

This appendix presents a summary of the contract operators for Contractual State Machines.

B.1 Summary of contract operators

The following section presents summary tables of the satisfaction relation (ground semantics) for the core (and derived) operators of the contract language. For the derived operators there is additionally a definition in terms of the core operators. The purpose of the derived operators is to:

- Provide specific short forms for the more general operators (and logical constructions thereof)
- Provide direct simplified definitions for certain forms (rather than computing conjunctions over more complex definitions)

Note that the theory provided only pertains to the *shared variable* (rather than standard encapsulated) setting. That is, it is based on a stronger satisfaction relation. Accordingly, the notion of refinement is shared variable ready simulation.

The operators described form the mathematical core of the language. In order to express useful contracts succinctly a further set of derived operators are required. This set is dependent on the embedding of statecharts into STGA/LTS and will entail further developments of the theory as will also be required for expressing notions such as data constraints across multiple transitions.

Satisfaction relation (ground semantics)

The following tables provide the conditions necessary for a process to satisfy a given operator. That is, they provide definitions for:

$$E \models \phi$$

to hold, for each process E and each formula ϕ.

Each process is characterised by an LLTS $\langle V, Val, S, Act, Proc, T, F, E_0 \rangle$, where V is a (finite) set of variables, Val is a (finite) set of values, $S = V \to Val$ is the set of valuation functions (data states), Act is the set of observable actions (where $\tau \notin Act$) which we extend to $Act_\tau = Act \cup \{\tau\}$, $Proc$ is the set of process expressions (behaviour states), $T \subseteq ((S \times Proc) \times (Act_\tau) \times (S \times Proc))$ is the transition relation, $F \subseteq (S \times Proc)$ is the set of data state/process expression pairs deemed inconsistent (the $False$ states), E_0 is the initial process expression (initial behavioural state). We write $(\sigma, E) \xrightarrow{\alpha} (\sigma', E')$ for $((\sigma, E), \alpha, (\sigma', E')) \in T$.

We insist (for this version of the theory) that:

$$(\sigma, E) \xrightarrow{\tau} (\sigma', E') \Rightarrow \sigma = \sigma'$$

The transition relation $\xrightarrow{\alpha}$ is closed to form $\stackrel{\epsilon}{\Longrightarrow}$ and $\stackrel{a}{\Longrightarrow}$ $(a \in Act)$ in the usual way:

$$(\sigma, E) \stackrel{\epsilon}{\Longrightarrow} (\sigma', E') \Leftrightarrow (\sigma, E) \xrightarrow{\tau}{}^{*} (\sigma', E')$$

$$(\sigma, E) \stackrel{a}{\Longrightarrow} (\sigma''', E''') \Leftrightarrow$$
$$\exists E', E'', \sigma', \sigma'' \bullet (\sigma, E) \stackrel{\epsilon}{\Longrightarrow} (\sigma', E') \wedge (\sigma', E') \xrightarrow{a} (\sigma'', E'') \wedge$$
$$(\sigma'', E'') \stackrel{\epsilon}{\Longrightarrow} (\sigma''', E''')$$

i.e. $(\sigma, E) \stackrel{\epsilon}{\Longrightarrow} (\sigma', E')$ iff $E = E' \wedge \sigma = \sigma'$ or some sequence of τ transitions take you from (σ, E) to (σ', E'); $(\sigma, E) \stackrel{a}{\Longrightarrow} (\sigma', E')$, for a non-tau action a iff a (possibly empty) sequence of τ actions, followed by an a action, followed by a (possibly empty) sequence of τ actions, takes you from (σ, E) to (σ', E').

We also restrict $\xrightarrow{a}$, $\stackrel{\epsilon}{\Longrightarrow}$ and $\stackrel{a}{\Longrightarrow}$ to $\xrightarrow{a}_F$, $\stackrel{\epsilon}{\Longrightarrow}_F$ and $\stackrel{a}{\Longrightarrow}_F$ respectively as follows:

$$(\sigma, E) \xrightarrow{\alpha}_F (\sigma', E') \Leftrightarrow (\sigma, E) \xrightarrow{\alpha} (\sigma', E') \wedge (\sigma, E) \notin F \wedge (\sigma', E') \notin F$$
$$(\sigma, E) \overset{\epsilon}{\Longrightarrow}_F (\sigma', E') \Leftrightarrow (\sigma, E) \xrightarrow{\tau}{}^*_F (\sigma', E')$$

$$(\sigma, E) \overset{a}{\Longrightarrow}_F (\sigma''', E''') \Leftrightarrow$$
$$\exists E', E'', \sigma', \sigma'' \bullet \ (\sigma, E) \overset{\epsilon}{\Longrightarrow}_F (\sigma', E') \wedge (\sigma', E') \xrightarrow{a}_F (\sigma'', E'') \wedge$$
$$(\sigma'', E'') \overset{\epsilon}{\Longrightarrow}_F (\sigma''', E''')$$

We define the *Ready Set* of a process E and state σ, $\mathcal{I}(E, \sigma)$, as the set of actions it may immediately perform from that state:

$$\mathcal{I}(\sigma, E) = \{\alpha \in Act_\tau | (\sigma, E) \xrightarrow{\alpha}\}$$

A process E is *stable* from state σ iff $\tau \notin \mathcal{I}(\sigma, E)$.

We further restrict $\overset{\epsilon}{\Longrightarrow}_F$ and $\overset{a}{\Longrightarrow}_F$ to $\overset{\epsilon}{\Longrightarrow\!|}$ and $\overset{a}{\Longrightarrow\!|}$ respectively as follows:

$(\sigma, E) \overset{\epsilon}{\Longrightarrow\!|} (\sigma', E')$ iff $(\sigma, E) \overset{\epsilon}{\Longrightarrow}_F (\sigma', E')$ and E' is stable from state σ'

$(\sigma, E) \overset{a}{\Longrightarrow\!|} (\sigma', E')$ iff $(\sigma, E) \overset{a}{\Longrightarrow}_F (\sigma', E')$ and E' is stable from state σ'

A process E *can stabilise*[1]. from state σ iff $\exists E', \sigma' \bullet \ (\sigma, E) \overset{\epsilon}{\Longrightarrow\!|} (\sigma', E')$.

The various transition definitions are lifted to *sets* of actions in the usual way. E.g.:

$$(\sigma, E) \overset{A}{\Longrightarrow\!|} (\sigma', E') \Leftrightarrow (\sigma, E) \overset{a}{\Longrightarrow\!|} (\sigma', E') \wedge a \in A$$

[1] Conversely, process E *cannot stabilise* from state σ iff $\neg\exists E', \sigma' \bullet \ (\sigma, E) \overset{\epsilon}{\Longrightarrow\!|} (\sigma', E')$

We now formalise the constraints for LLTS:

- The LLTS must be τ-*pure*

  $\forall E, \sigma \bullet \mathcal{I}(\sigma, E) = \{\tau\} \vee \tau \notin \mathcal{I}(\sigma, E)$

- Process/state pairs which cannot stabilise must be in F

  E *cannot stabilise* from state $\sigma \Rightarrow \ (\sigma, E) \in F$

- False processes *backtrack*

  $(\exists \alpha \bullet \alpha \in \mathcal{I}(\sigma, E) \wedge$

  $\forall E', \sigma' \bullet (\sigma, E) \xrightarrow{\alpha} (\sigma', E') \Rightarrow (\sigma', E') \in F))$

  $\Rightarrow (\sigma, E) \in F$

We assume that the SOS rules (and LLTS operators) preserve the LLTS constraints. These are omitted.

The satisfaction relation for the core operators is defined by the following table:

$E \models \phi$	Condition	Notes
$E \models True$	$True$	Every process satisfies $True$
$E \models False$	$\forall \sigma \bullet (\sigma, E) \in F$	Only processes inconsistent from all states satisfy $False$
$E \models [a[x : P_{x'}]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge(\sigma, \sigma') \models P_{x'})$ $\Rightarrow E' \models \phi$	E satisfies $[a[x : P_{x'}]\phi$ iff all processes reachable by a from E (from any state) with an update consistent with $P_{x'}$ (from $\exists x' \bullet P_{x'}$) satisfy ϕ. We refer to this as the *if* operator.
$E \models \langle a[x : P_{x'}]\rangle$	$\exists E', \sigma, \sigma' \bullet (\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $(\sigma, \sigma') \models P_{x'}$	E satisfies $\langle a[x : P_{x'}]\rangle$ iff it is possible to take an a transition which is consistent with $P_{x'}$ (from $\exists x' \bullet P_{x'}$). We refer to this as the *enable* operator.
$E \models \langle\!\langle a[x : P_{x'}]\rangle\!\rangle$	$\forall \sigma \bullet \sigma \models (\exists x' \bullet P_{x'}) \Rightarrow$ $\exists E', \sigma' \bullet (\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $(\sigma, \sigma') \models P_{x'}$	E satisfies $\langle\!\langle a[x : P_{x'}]\rangle\!\rangle$ iff it is possible to take an a transition which is consistent with $P_{x'}$ from every state in $\exists x' \bullet P_{x'}$. We refer to this as the *totalised enable* operator.
$E_0 \models \phi \curvearrowright \psi$	$\big[\forall \sigma_1, \sigma_1', \ldots, \sigma_n, \sigma_n',$ $E_1, \ldots, E_n \bullet$ $(\sigma_1, E_0) \stackrel{Act}{\Longrightarrow\!\!|} (\sigma_1', E_1) \wedge \ldots \wedge$ $(\sigma_n, E_{n-1}) \stackrel{Act}{\Longrightarrow\!\!|} (\sigma_n', E_n) \Rightarrow\big]$ $E_n \models \phi$ or $\exists i | i \leq n \bullet E_i \models \psi$ Must hold for all $n \geq 0$	E_0 satisfies $\phi \curvearrowright \psi$ if every process reachable from E_0 satisfies ϕ, **unless** the process satisfies ψ, at which point, and subsequently, it is no longer required to satisfy ϕ. A process E_n is *reachable* from E_0 if it is E_0 itself, or if some sequence of source data states exist enabling transitions from E_0 through all E_i to E_n. NB. The source data state of each transition need not be the same as the target of the previous transition. We refer to this as the *unless* operator.
$E \models \phi \wedge \psi$	$E \models \phi$ and $E \models \psi$	E satisfies $\phi \wedge \psi$ if it satisfies both ϕ and ψ
$E \models \phi \vee \psi$	$E \models \phi$ or $E \models \psi$	E satisfies $\phi \wedge \psi$ if either it satisfies ϕ or it satisfies ψ

Where x, x' generalise to vectors of variables, and e to a vector of expressions.

Given the core operators, we now extend the language by providing more convenient forms for common constructs. First, we define the iterator $\mu_k X$:

Formula	Equivalent	Notes
$\mu_0 X.\phi$	$False$	The 0^{th} iterator is defined False
$\mu_k X.\phi$ (where $k > 0$)	$\phi[X/(\mu_{k-1} X.\phi)]$	The k^{th} iterator holds if ϕ with all occurrences of X replaced with the $(k-1)^{th}$ iterator holds.

The next set of derived operators provide specific forms of the $[\ldots]$, $\langle\ldots\rangle$ and $\langle\!\![\ldots]\!\!\rangle$ operators. They are defined by interpretation in the other (more general) operators, plus we give a direct characterisation for completeness. Their definitions are:

Formula	Equivalent	Notes
$[a]\phi$	$[a[x : \text{true}]]\phi$	All processes reachable by a, from any source data state, with any update, must behave according to ϕ
$[a[x = e]]\phi$	$[a[x : x' = e]]\phi$	All processes reachable by a, from any source data state, with an update setting x to e, must behave according to ϕ
$[a[\text{skip}]]\phi$	$[a[x : x' = x]]\phi$	All processes reachable by a, from any source data state, with no update (values remain unchanged), must behave according to ϕ
$[a[P \Longrightarrow x = e]]\phi$	$[a[x : P \wedge x' = e]]\phi$	All processes reachable by a, from a source data state satisfying P, with an update setting x to e, must behave according to ϕ
$[a[P \Longrightarrow \text{skip}]]\phi$	$[a[x : P \wedge x' = x]]\phi$	All processes reachable by a, from a source data state satisfying P, with no update (values remain the same), must behave according to ϕ
$[a[P]]\phi$	$[a[x : P]]\phi$	All processes reachable by a, from a source data state satisfying P, with any update, must behave according to ϕ

Formula	Equivalent	Notes
$\langle a \rangle$	$\langle a[x : \text{true}] \rangle$	An a transition with some update is possible from some source data state.
$\langle a[x = e] \rangle$	$\langle a[x : x' = e] \rangle$	An a transition with the update $x = e$ is possible from some source data state.
$\langle a[\text{skip}] \rangle$	$\langle a[x : x' = x] \rangle$	An a transition with no update (values remain the same) is possible from some source data state.
$\langle a[P \Longrightarrow x = e] \rangle$	$\langle a[x : P \wedge x' = e] \rangle$	An a transition with update $x = e$ is possible from a source data state satisfying P.
$\langle a[P \Longrightarrow \text{skip}] \rangle$	$\langle a[x : P \wedge x' = x] \rangle$	An a transition with no update (values remain the same) is possible from a source data state satisfying P.
$\langle a[P] \rangle$	$\langle a[x : P] \rangle$	An a transition with some update is possible from a source data state satisfying P.

Formula	Equivalent	Notes
$[\![a]\!]$	$[\![a[x : \text{true}]]\!]$	An a transition with some update is possible from all source data states.
$[\![a[x = e]]\!]$	$[\![a[x : x' = e]]\!]$	An a transition with the update $x = e$ is possible from all source data states.
$[\![a[\text{skip}]]\!]$	$[\![a[x : x' = x]]\!]$	An a transition with no update (values remain the same) is possible from all source data states.
$[\![a[P \Longrightarrow x = e]]\!]$	$[\![a[x : P \wedge x' = e]]\!]$	An a transition with the update $x = e$ is possible from all source data states satisfying P.
$[\![a[P \Longrightarrow \text{skip}]]\!]$	$[\![a[x : P \wedge x' = x]]\!]$	An a transition with no update (values remain the same) is possible from all source data states satisfying P.
$[\![a[P]]\!]$	$[\![a[x : P]]\!]$	An a transition with some update is possible from all source data states satisfying P.

Here is the direct characterisation of the above operators.

$E \models \phi$	Condition	Notes
$E \models [a]\phi$	$\forall E', \sigma, \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\Rightarrow E' \models \phi$	All processes reachable by a, from any source data state, with any update, must behave according to ϕ
$E \models [a[x = e]]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge \sigma' = \sigma\{e/x\})$ $\Rightarrow E' \models \phi$	All processes reachable by a, from any source data state, with an update setting x to e, must behave according to ϕ
$E \models [a[\text{skip}]]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge \sigma' = \sigma)$ $\Rightarrow E' \models \phi$	All processes reachable by a, from any source data state, with no update (values remain unchanged), must behave according to ϕ
$E \models [a[P \Longrightarrow x = e]]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge \sigma' = \sigma\{e/x\}$ $\wedge \sigma \models P)$ $\Rightarrow E' \models \phi$	All processes reachable by a, from a source data state satisfying P, with an update setting x to e, must behave according to ϕ
$E \models [a[P \Longrightarrow \text{skip}]]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge \sigma' = \sigma$ $\wedge \sigma \models P)$ $\Rightarrow E' \models \phi$	All processes reachable by a, from a source data state satisfying P, with no update (values remain the same), must behave according to ϕ
$E \models [a[P]]\phi$	$\forall E', \sigma, \sigma' \bullet$ $((\sigma, E) \stackrel{a}{\Longrightarrow\!\!|} (\sigma', E')$ $\wedge \sigma \models P)$ $\Rightarrow E' \models \phi$	All processes reachable by a, from a source data state satisfying P, with any update, must behave according to ϕ

In the above table, $\sigma\{e/x\}$ denotes the data valuation function σ with the value of x replaced with the value of e evaluated in σ:

$E \models \phi$	Condition	Notes
$E \models \langle a \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E')$	An a transition with some update is possible from some source data state.
$E \models \langle a[x = e] \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $\sigma' = \sigma\{e/x\}$	An a transition with the update $x = e$ is possible from some source data state.
$E \models \langle a[\text{skip}] \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $\sigma' = \sigma$	An a transition with no update (values remain the same) is possible from some source data state.
$E \models \langle a[P \Longrightarrow x = e] \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $\sigma' = \sigma\{e/x\} \wedge$ $\sigma \models P$	An a transition with update $x = e$ is possible from a source data state satisfying P.
$E \models \langle a[P \Longrightarrow \text{skip}] \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $\sigma' = \sigma \wedge$ $\sigma \models P$	An a transition with no update (values remain the same) is possible from a source data state satisfying P.
$E \models \langle a[P] \rangle$	$\exists E', \sigma, \sigma' \bullet$ $(\sigma, E) \overset{a}{\Longrightarrow\!\!|} (\sigma', E') \wedge$ $\sigma \models P$	An a transition with some update is possible from a source data state satisfying P.

$E \models \phi$	Condition	Notes
$E \models \langle\!\langle a \rangle\!\rangle$	$\forall \sigma \bullet$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E')$	An a transition with some update is possible from all source data states.
$E \models \langle\!\langle a[x = e] \rangle\!\rangle$	$\forall \sigma \bullet$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E') \wedge$ $\sigma' = \sigma\{e/x\}$	An a transition with the update $x = e$ is possible from all source data states.
$E \models \langle\!\langle a[\text{skip}] \rangle\!\rangle$	$\forall \sigma \bullet$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E') \wedge$ $\sigma' = \sigma$	An a transition with no update (values remain the same) is possible from all source data states.
$E \models \langle\!\langle a[P \Longrightarrow x = e] \rangle\!\rangle$	$\forall \sigma \bullet \sigma \models P \Rightarrow$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E') \wedge$ $\sigma' = \sigma\{e/x\}$	An a transition with the update $x = e$ is possible from all source data states satisfying P.
$E \models \langle\!\langle a[P \Longrightarrow \text{skip}] \rangle\!\rangle$	$\forall \sigma \bullet \sigma \models P \Rightarrow$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E') \wedge$ $\sigma' = \sigma$	An a transition with no update (values remain the same) is possible from all source data states satisfying P.
$E \models \langle\!\langle a[P] \rangle\!\rangle$	$\forall \sigma \bullet \sigma \models P \Rightarrow$ $\exists E', \sigma' \bullet$ $(\sigma, E) \stackrel{a}{\Longrightarrow\!\mid} (\sigma', E')$	An a transition with some update is possible from all source data states satisfying P.

The next set of derived operators lift the action operators to *sets* of actions.

Formula	Equivalent	Notes
$[a_1 \dots a_n]\phi$	$\bigwedge_{b \in \{a_1,\dots,a_n\}} [b]\phi$	All processes reachable by some a_k, from any source data state, with any update, must behave according to ϕ
$[a_1 \dots a_n[\gamma]]\phi$	$\bigwedge_{b \in \{a_1,\dots,a_n\}} [b[\gamma]]\phi$	All processes reachable by some a_k, with an update consistent with γ (from a source data state satisfying γ), must behave according to ϕ
$[-a_1 \dots a_n]\phi$	$\bigwedge_{b \in Act \setminus \{a_1,\dots,a_n\}} [b]\phi$	All processes reachable by some b not in $\{a_1, \dots a_n\}$, from any source data state, with any update, must behave according to ϕ
$[-a_1 \dots a_n[\gamma]]\phi$	$\bigwedge_{b \in Act \setminus \{a_1,\dots,a_n\}} [b[\gamma]]\phi$	All processes reachable by some b not in $\{a_1 \dots a_n\}$, with an update consistent with γ (from a source data state satisfying γ), must behave according to ϕ
$[\_]\phi$	$\bigwedge_{b \in Act} [b]\phi$	All processes reachable by some action, from any source data state, with any update, must behave according to ϕ
$[\_[\gamma]]\phi$	$\bigwedge_{b \in Act} [b[\gamma]]\phi$	All processes reachable by some action, with an update consistent with γ (from a source data state satisfying γ), must behave according to ϕ

We omit the direct characterisation of the operators.

Formula	Equivalent	Notes
$\langle a_1 \dots a_n \rangle$	$\bigvee_{b \in \{a_1,\dots,a_n\}} \langle b \rangle$	It is possible to take some transition a_k, from some source data state, with some update.
$\langle a_1 \dots a_n[\gamma] \rangle$	$\bigvee_{b \in \{a_1,\dots,a_n\}} \langle b[\gamma] \rangle$	It is possible to take some transition a_k, with an update consistent with γ (from a source state satisfying γ).
$\langle -a_1 \dots a_n \rangle$	$\bigvee_{b \in Act \setminus \{a_1,\dots,a_n\}} \langle b \rangle$	It is possible to take some transition b not in $\{a_1, \dots, a_n\}$, from some source data state, with some update.
$\langle -a_1 \dots a_n[\gamma] \rangle$	$\bigvee_{b \in Act \setminus \{a_1,\dots,a_n\}} \langle b[\gamma] \rangle$	It is possible to take some transition b not in $\{a_1, \dots, a_n\}$, with an update consistent with γ (from a source state satisfying γ).
$\langle \_ \rangle$	$\bigvee_{b \in Act} \langle b \rangle$	It is possible to take some transition, from some source data state, with some update.
$\langle \_[\gamma] \rangle$	$\bigvee_{b \in Act} \langle b[\gamma] \rangle$	It is possible to take some transition, with an update consistent with γ (from a source state satisfying γ).

Formula	Equivalent	Notes
$\langle\!\langle a_1 \dots a_n \rangle\!\rangle$	$\bigvee_{b \in \{a_1,\dots,a_n\}} \langle\!\langle b \rangle\!\rangle$	One of the actions a_k is available as a transition from every source data state, with some update.
$\langle\!\langle a_1 \dots a_n[\gamma] \rangle\!\rangle$	$\bigvee_{b \in \{a_1,\dots,a_n\}} \langle\!\langle b[\gamma] \rangle\!\rangle$	One of the actions a_k is available as a transition from every source state satisfying γ, with an update consistent with γ.
$\langle\!\langle -a_1 \dots a_n \rangle\!\rangle$	$\bigvee_{b \in Act \setminus \{a_1,\dots,a_n\}} \langle\!\langle b \rangle\!\rangle$	One of the actions b not in $\{a_1, \dots, a_n\}$ is available as a transition from every source data state, with some update.
$\langle\!\langle -a_1 \dots a_n[\gamma] \rangle\!\rangle$	$\bigvee_{b \in Act \setminus \{a_1,\dots,a_n\}} \langle\!\langle b[\gamma] \rangle\!\rangle$	One of the actions b not in $\{a_1, \dots, a_n\}$ is available as a transition from every source state satisfying γ, with an update consistent with γ.
$\langle\!\langle \_ \rangle\!\rangle$	$\bigvee_{b \in Act} \langle\!\langle b \rangle\!\rangle$	There is some action available as a transition from every source data state, with some update.
$\langle\!\langle \_[\gamma] \rangle\!\rangle$	$\bigvee_{b \in Act} \langle\!\langle b[\gamma] \rangle\!\rangle$	There is some action available as a transition from every source state satisfying γ, with an update consistent with γ.

The next set of derived operators are for *disabled* and *exclusively enabled* actions.

Formula	Equivalent	Notes
$\overline{\langle a_1, \ldots, a_n \rangle}$	$\bigwedge_{b \in \{a_1,\ldots,a_n\}} [b]False$	$\{a_1, \ldots, a_n\}$ are disabled. I.e. No actions $\{a_1, \ldots, a_n\}$ are enabled.
$\overline{\langle a_1, \ldots, a_n[\gamma] \rangle}$	$\bigwedge_{b \in \{a_1,\ldots,a_n\}} [b[\gamma]]False$	$\{a_1, \ldots, a_n\}$ are disabled for updates consistent with γ. I.e. actions $\{a_1, \ldots, a_n\}$ are only enabled for updates inconsistent with γ (i.e. from source data states not satisfying γ, or from source states satisfying γ whose updates are not admitted by γ).
$\overline{\langle -a_1, \ldots, a_n \rangle}$	$\bigwedge_{b \in Act \setminus \{a_1,\ldots,a_n\}} [b]False$	All actions except $\{a_1, \ldots, a_n\}$ are disabled. I.e. No actions other than $\{a_1, \ldots, a_n\}$ are enabled.
$\overline{\langle -a_1, \ldots, a_n[\gamma] \rangle}$	$\bigwedge_{b \in Act \setminus \{a_1,\ldots,a_n\}} [b[\gamma]]False$	All actions other than $\{a_1, \ldots, a_n\}$ are disabled for updates consistent with γ. I.e. actions other than $\{a_1, \ldots, a_n\}$ are only enabled for updates inconsistent with γ (i.e. from source data states not satisfying γ, or from source states satisfying γ whose updates are not admitted by γ).
$\overline{\langle \_ \rangle}$	$\bigwedge_{b \in Act} [b]False$	All actions are disabled. I.e. No actions are enabled.
$\overline{\langle \_[\gamma] \rangle}$	$\bigwedge_{b \in Act} [b[\gamma]]False$	All actions are disabled for updates consistent with γ. I.e. actions are only enabled for updates inconsistent with γ (i.e. from source data states not satisfying γ, or from source states satisfying γ whose updates are not admitted by γ).

NB. $\overline{\langle -a_1, \ldots, a_n \rangle}$ is not the same as $\langle a_1, \ldots, a_n \rangle$. E.g. A deadlocked process satisfies $\overline{\langle -a \rangle}$ but does not satisfy $\langle a \rangle$.

Formula	Equivalent	Notes
$\underline{\langle a_1, \ldots, a_n \rangle}$	$\bigvee_{b \in \{a_1, \ldots, a_n\}} (\langle b \rangle \wedge \overline{\langle -b \rangle})$	One of the actions $\{a_1, \ldots, a_n\}$ is exclusively enabled. I.e. One of the actions a_k is enabled whilst all other actions (not a_k) are disabled.
$\underline{\langle a_1, \ldots, a_n[\gamma] \rangle}$	$\bigvee_{b \in \{a_1, \ldots, a_n\}} (\langle b[\gamma] \rangle \wedge \overline{\langle -b[\gamma] \rangle})$	One of the actions $\{a_1, \ldots, a_n\}$ is exclusively enabled for update γ. I.e. One of the actions a_k can perform an update consistent with γ whilst all other actions (not a_k) cannot.
$\underline{\langle -a_1, \ldots, a_n \rangle}$	$\bigvee_{b \in Act \setminus \{a_1, \ldots, a_n\}} (\langle b \rangle \wedge \overline{\langle -b \rangle})$	One of the actions not in $\{a_1, \ldots, a_n\}$ is exclusively enabled. I.e. An actions b not in $\{a_1, \ldots, a_n\}$ is enabled whilst all other actions (not b) are disabled.
$\underline{\langle -a_1, \ldots, a_n[\gamma] \rangle}$	$\bigvee_{b \in Act \setminus \{a_1, \ldots, a_n\}} (\langle b[\gamma] \rangle \wedge \overline{\langle -b[\gamma] \rangle})$	One of the actions not in $\{a_1, \ldots, a_n\}$ is exclusively enabled for update γ. I.e. One of the actions b not in $\{a_1, \ldots, a_n\}$ can perform an update consistent with γ whilst all other actions (not a_k) cannot.
$\underline{\langle \_ \rangle}$	$\bigvee_{b \in Act} (\langle b \rangle \wedge \overline{\langle -b \rangle})$	Some action is exclusively enabled. I.e. Some action b is enabled whilst all other actions (not b) are disabled.
$\underline{\langle \_[\gamma] \rangle}$	$\bigvee_{b \in Act} (\langle b[\gamma] \rangle \wedge \overline{\langle -b[\gamma] \rangle})$	Some action is exclusively enabled for update γ. I.e. Some actions b can perform an update consistent with γ whilst all other actions (not b) cannot.

An example of the use of $\underline{\langle a[\gamma] \rangle}$ is $\underline{\langle a[P] \rangle}$ meaning a is exclusively enabled from all source data states satisfying P.

The direct characterisation of the above operators is omitted.

Formula	Equivalent	Notes
$\Box\phi$	$\phi \curvearrowright False$	"Always ϕ" is defined as ϕ unless false. Satisfied by both the (immediately) $False$ process and processes which always satisfy ϕ.
$\Diamond_k\phi$	$\mu_k X.(\phi \vee (\langle\_\rangle \wedge [\_]X))$	"ϕ within k" is satisfied by processes that make ϕ true within k steps.

Direct characterisation of the above operators is omitted.

Appendix C

Case study - the docking system

This appendix describes in detail all of the steps performed to refine the ShipReq component in the case study.

C.1 Refinement of the ShipReq component

The contract for the ShipReq component is specified as: $(((A \wedge B) \wedge C) \wedge D) \wedge E$

$\mathbf{R} = (q_1 == 0 \vee q_2 == 0) \wedge (q_1! = s \wedge q_2! = s)$

$\mathbf{A} = ((\overline{\langle{}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$
$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$
$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle 'deny[: R]\rangle}))$

$\mathbf{B} = \Box[done]((\overline{\langle{}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$
$\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$
$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle 'deny[: R]\rangle}))$

$\mathbf{C} = \Box['deny]((\overline{\langle{}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$
noindent $\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$
$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle 'deny[: R]\rangle}))$

$\mathbf{D} = \Box['dockquay]((\overline{\langle{}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[s : s! = s']\rangle}$
$\curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$

$\mathbf{E} = \Box\overline{\langle \_[q_1, q_2 : (q_1'! = q_1) \vee (q_2'! = q_2)]\rangle}$

1. Apply the *Conjunction introduction* pattern four times. The *Conjunction introduction* pattern is applicable if the selected component is a state whose inner contract is an *And* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *And* ψ". The pattern introduces two new substates, one with inner contract "ϕ" and the other with inner contract "ψ" (see Figure C.1).

Figure C.1: Conjunction introduction pattern applied four times

2. Refine the state with contract A.

$\mathbf{R} = (q_1 == 0 \vee q_2 == 0) \wedge (q_1! = s \wedge q_2! = s)$

$\mathbf{A} = ((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$
$\curvearrowright (\overline{\langle-' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$
$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$
$\overline{\langle' deny[: R]\rangle}))$

Apply the *Unfold unless* pattern. The *Unfold unless* pattern is applicable if the selected component is a state whose inner contract is an *Unless* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *Unless* ψ" and the pattern

unfolds the contract to "ϕ *And (If* _ (ϕ *Unless* ψ))".

ϕ is $(\overline{\langle\!\langle \prime dockquay, \prime deny, done \rangle\!\rangle} \wedge \langle\!\langle [\!-\prime dockquay, \prime deny, done] \rangle\!\rangle)$

ψ is $(\overline{\langle\!\langle -\prime dockquay, \prime deny \rangle\!\rangle} \wedge \langle\!\langle [\prime dockquay[: R]] \rangle\!\rangle \wedge$
$\overline{\langle\!\langle \prime dockquay[q_1, q_2, s : \neg(R \wedge skip)] \rangle\!\rangle} \wedge \langle\!\langle [\prime deny[q_1, q_2, s : \neg R]] \rangle\!\rangle \wedge$
$\overline{\langle\!\langle \prime deny[: R] \rangle\!\rangle})$

$A1 = \phi$

$A2 =$ *(If* _ (ϕ *Unless* ψ)).

The pattern unfolds the Contract A to $A1 \wedge A2$

A1 $= \overline{\langle\!\langle \prime dockquay, \prime deny, done \rangle\!\rangle} \wedge \langle\!\langle [\!-\prime dockquay, \prime deny, done] \rangle\!\rangle$

A2 $= [\_]((\overline{\langle\!\langle \prime dockquay, \prime deny, done \rangle\!\rangle} \wedge \langle\!\langle [\!-\prime dockquay, \prime deny, done] \rangle\!\rangle)$
$\curvearrowright (\overline{\langle\!\langle -\prime dockquay, \prime deny \rangle\!\rangle} \wedge \langle\!\langle [\prime dockquay[: R]] \rangle\!\rangle \wedge$
$\overline{\langle\!\langle \prime dockquay[q_1, q_2, s : \neg(R \wedge skip)] \rangle\!\rangle} \wedge \langle\!\langle [\prime deny[q_1, q_2, s : \neg R]] \rangle\!\rangle \wedge$
$\overline{\langle\!\langle \prime deny[: R] \rangle\!\rangle}))$

3. Apply the *Conjunction introduction* pattern to the state with contract $A1 \wedge A2$. Two new states are introduced, one with contract $A1$ and the other with contract $A2$.

4. Apply the *Conjunction introduction* pattern to the state with contract $A1$. The contract is $A1 = A9 \wedge A10$. The pattern creates two new substates one with contract $A9$ and the other with contract $A10$ (see Figure C.2).

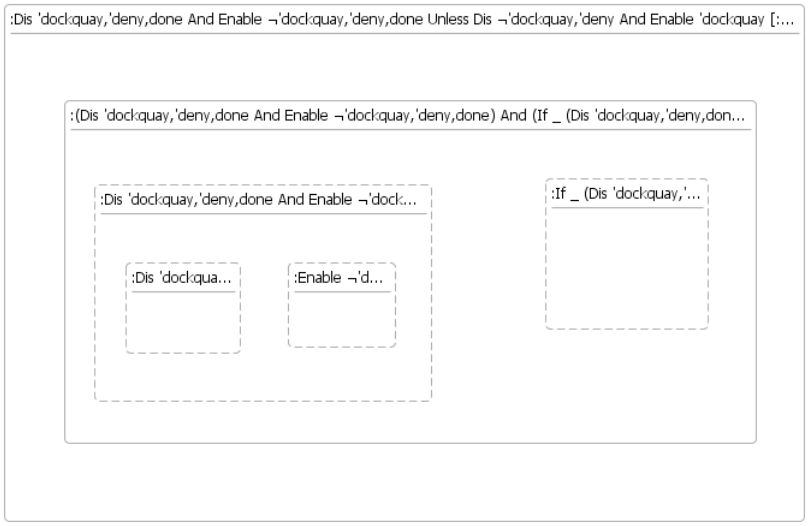

Figure C.2: Conjunction introduction pattern applied to contract A1

A1 $= \overline{\langle\!\langle \prime dockquay, \prime deny, done \rangle\!\rangle} \wedge \langle\!\langle [\!-\prime dockquay, \prime deny, done] \rangle\!\rangle$

A9 $= \overline{\langle\!\langle \prime dockquay, \prime deny, done \rangle\!\rangle}$

A10 $= \langle[-'dockquay,' deny, done]\rangle$

5. Apply the *Disable* pattern to the state with contract $A9$ to introduce two new transitions *dock1* and *dock2*. The *Disable* operator specifies that none of the events in a set are available. When accompanied by a variable update the action is disabled for all the updates consistent with the expression. It may still be enabled within its guard as long as its update is not consistent.

The *Disable* operator contract $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed as an equivalent *If* operator expression $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*. To introduce transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, HST is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological.
A9 $= \overline{\langle'dockquay,' deny, done\rangle}$ is rewritten as an *If* operator expression.
A9 $= [' dockquay,' deny, done]False$.
The two new transitions required are, $dock1[s = 1]$ and $dock2[s = 2]$. The pattern checks firstly whether the transition event is in the contract *eventlist* and if it is then a further check is conducted invoking HST to ensure that the transition update expression and the contract update expression are disjoint and there is no overlap. The first transition is $dock1[s = 1]$ whose event is not included in the contract *eventlist* so the transition is added with subsequent behaviour *True* and no further check is required. Similarly the second transition is $dock2[s = 2]$ whose event is not included in the contract *eventlist* so the transition is added with subsequent behaviour *True* and no further check is required (see Figure C.3).
6. Apply the *Combine states* pattern to combine the substates of A9 with contract *True*. The *Combine states* pattern is applicable to two states. The pattern checks that the states to be combined have the same parent state and no substates or outgoing transitions. It also checks that their inner contracts are syntactically equivalent. The states both have inner contracts of *True* (see Figure C.4).

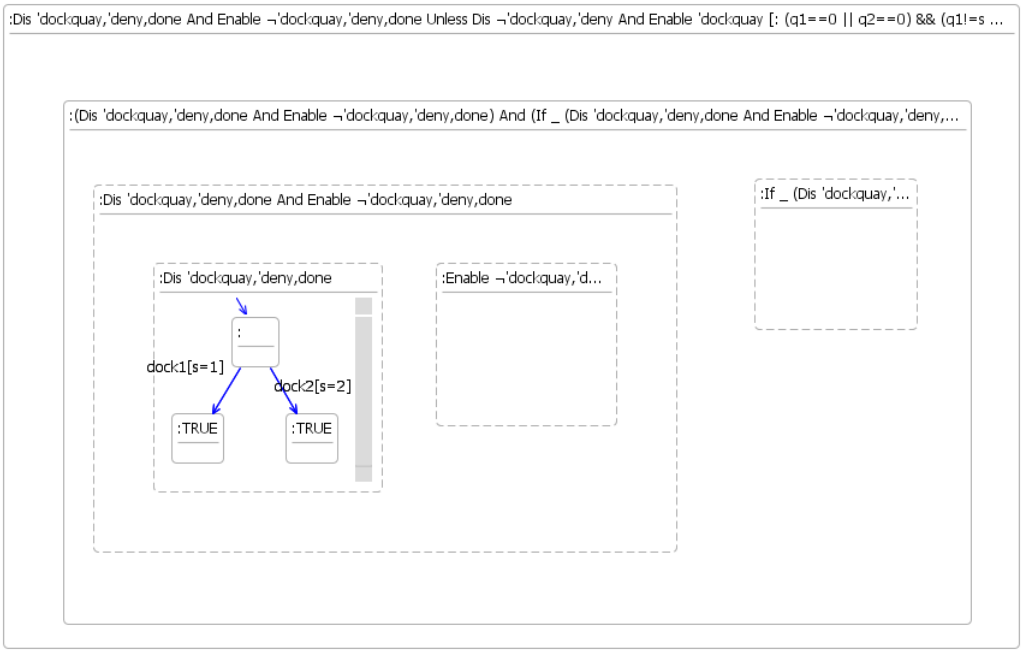

Figure C.3: Disable pattern applied to contract A9

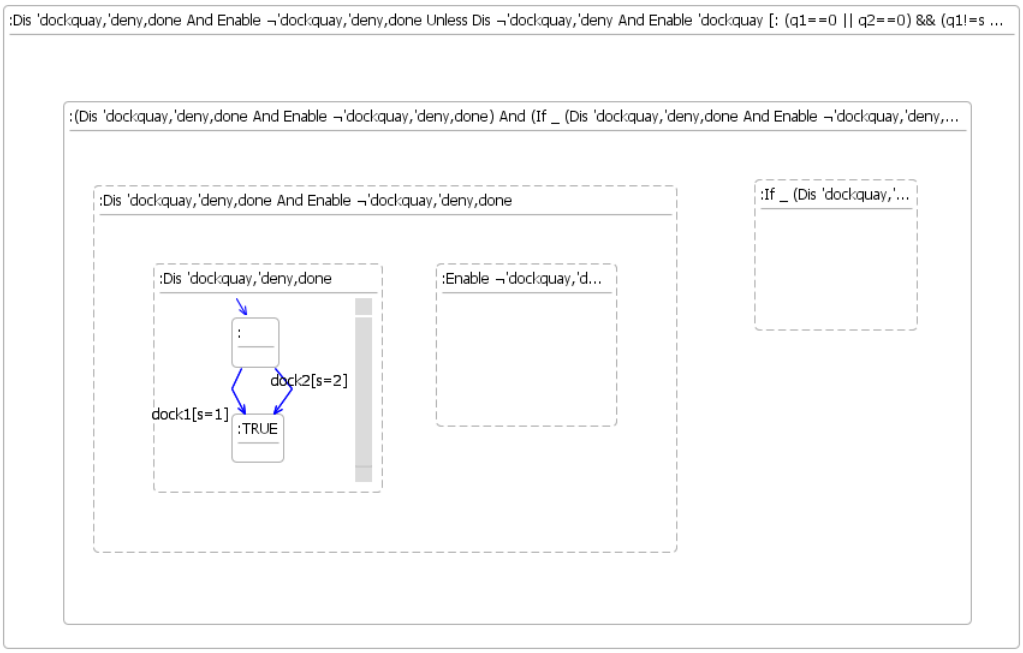

Figure C.4: Combine states pattern applied to state with contract A9

7. Apply the *TotEnable* pattern to the state with contract $A10$ to introduce two new transitions *dock1* and *dock2*. The *TotEnable* pattern refines a *TotEnable* contract to a state machine model. The *TotEnable* pattern is applicable if the selected component is a state. The pattern checks that the state's inner contract has the form "*TotEnable eventFormula*" or "*TotEnable eventFormula [updateExpression]*". *EventFormula* can be a set of events, a negated set of events or the underscore character. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract.

Checks based on syntax are performed initially. To determine if the transition satisfies the contract the transition event must be an event included in the contract's *eventlist*. If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the contract's update expression is syntactically equivalent to the transition's update expression then the transition satisfies the contract. If the contract is of the form $\langle[eventlist]\rangle$ it is satisfied by transitions of the form *event*, *event[assignmentlist]*, *event[skip]*. If the contract has the form $\langle[eventlist[skip]]\rangle$ or $\langle[eventlist[G \Longrightarrow skip]]\rangle$ it is satisfied by transitions of the form *event*, $event[skip]$. Otherwise a more detailed check is required to determine if the transition satisfies the contract. To ensure that the update expressions are consistent the following check is required. The side-condition is $\vdash ((\exists x' : P) \Rightarrow (\exists x' : P')) \wedge ((\exists x' : P) \wedge P' \Rightarrow P)$. Currently this check is verified by inspection as it requires the evaluation of a conjecture with quantifiers on data and this functionality is not yet supported by HST.

The contract is, $\mathbf{A10} = \langle[-'dockquay,' deny, done]\rangle$.

The two new transitions required are, $dock1[s = 1]$ and $dock2[s = 2]$.

The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract, and conducts syntactic checks initially to establish this. If the transition event is not in the contract's negated list of events and if the contract is of the form $\langle[eventlist]\rangle$ it is satisfied by a transition of the form *event[assignmentlist]*.

For the first transition $dock1[s = 1]$, *dock1* is not in the negated list of events in the contract $\langle[-'dockquay,' deny, done]\rangle$ also the contract is of the form $\langle[eventlist]\rangle$ and the transition is of the form, *event[assignmentlist]* so the pattern concludes that the transition satisfies the contract and the contract is replaced by a state machine with both new

transitions (see Figure C.5).

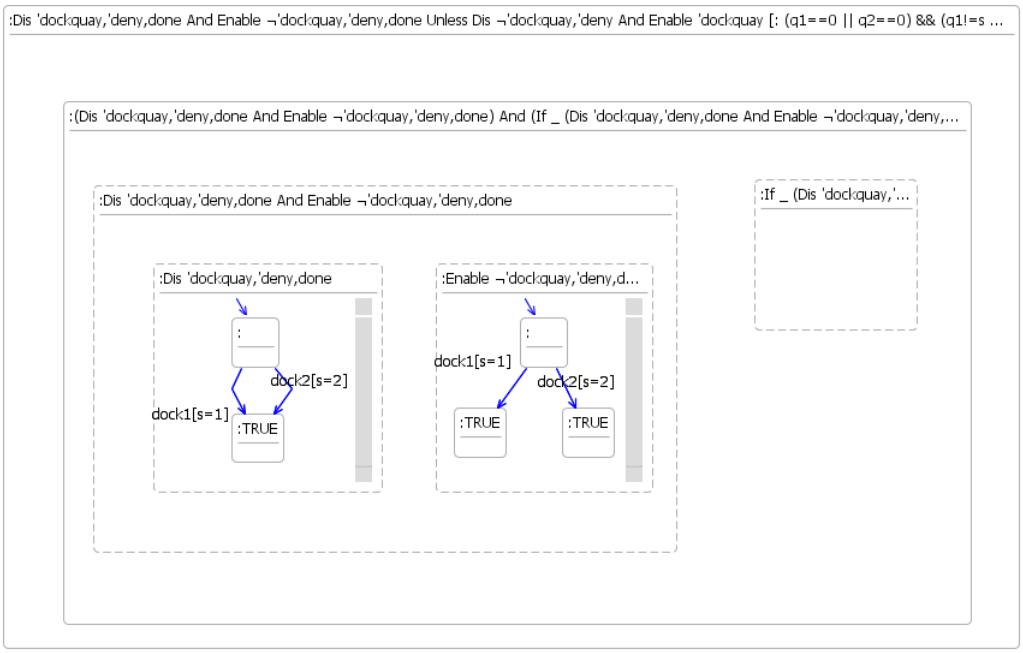

Figure C.5: TotEnable pattern applied to contract A10

8. Apply the *Combine states* pattern to combine the substates of $A10$ with contract *True*.

9. Apply the *Conjunction elimination* pattern to the state with contract $A1$. The *Conjunction elimination* pattern is applicable if the selected component is a state of type *conjunction*. It eliminates a conjunction of states. This pattern can be applied to the conjunction of like diagrams and throws away the copies. The pattern relies on inspection to determine equivalence (see Figure C.6).

10. Apply pattern *Redundant hierarchy*, inner contract $A1$ is assigned to the outer contract of the start state. The *Redundant hierarchy* pattern is applicable if the selected component is a state. The pattern checks that the state has substates and no incoming/outgoing transitions. The pattern flattens the level of hierarchy by removing the selected component but retaining its substates. If the selected component is a substate and a start state, the pattern assigns the inner contract of its parent state to the outer contract of its start state. The pattern verifies that the state has substates and no outgoing or incoming transitions, it is itself a substate so the pattern removes the selected component, (the redundant level of hierarchy), retains its substates and assigns the inner contract of its parent state, $A1$ to the outer contract of its start state (see Figure C.7).

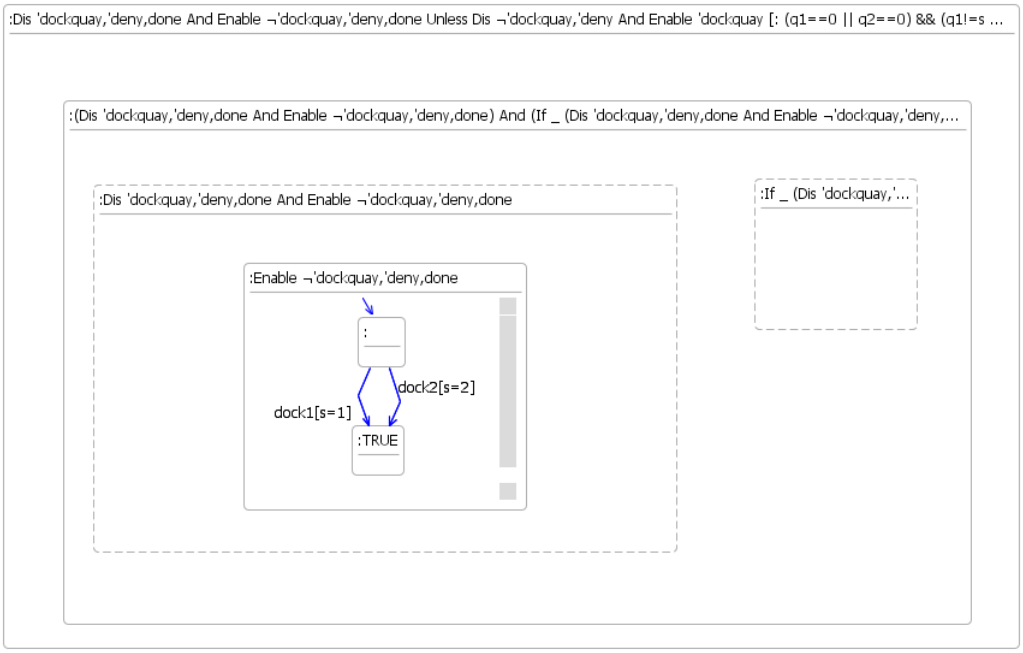

Figure C.6: Conjunction elimination pattern applied to state with contract A1

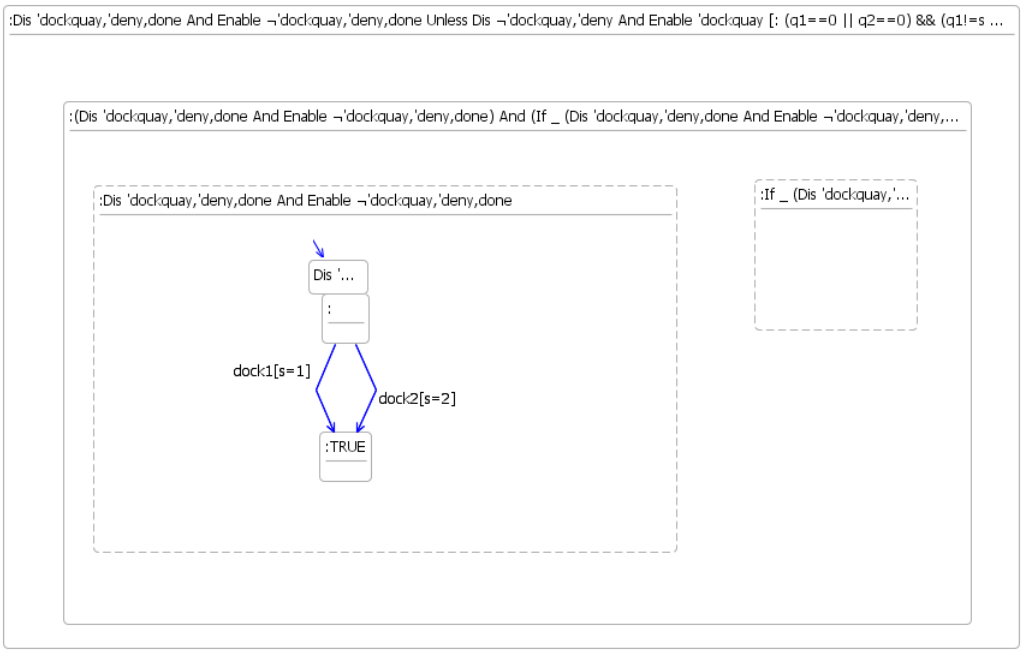

Figure C.7: Redundant hierarchy pattern applied to state with contract A10

11. Apply the *If* pattern to state with contract $A2$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ". *EventFormula* can be a set of events, a negated set of events or the underscore character.

For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist* the side-condition verifies that the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological.

It is not necessary to perform any checks for transitions with subsequent behaviour "ϕ" as if the conditions for the subsequent behaviour "ϕ" do not hold, then the subsequent behaviour is *True* and this contract can then be refined to "ϕ". The If pattern does not permit the introduction of transitions with subsequent behaviour *False*. A permitted refinement is to weaken the update expression. If the update expression for the contract is [x:P] and the update expression for the transition is [x:P'], the proof obligation $\vdash P \Rightarrow P'$ must hold for the transition to subsequently behave as ϕ. We do not need to add this to the side-condition check for transitions with subsequent behaviour of ϕ, as explained above it is not necessary to perform any checks for these transitions.

The contract is: $\mathbf{A2} = [\_]((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$

$$\curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$$
$$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle' deny[: R]\rangle}))$$

The two new transitions required are, $dock1[s = 1]$ and $dock2[s = 2]$

The contract $A2$ has the form $[\_]\phi$, if any event is enabled the transition subsequently behaves like ϕ.

$\phi = (\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$

$$\curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge$$
$$\overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle' deny[: R]\rangle})$$

The contract states that all enabled transitions subsequently behave like ϕ. The contract is replaced by a state machine with the transitions $dock1[s = 1]$ and $dock2[s = 2]$ that

subsequently behave like ϕ (see Figure C.8).

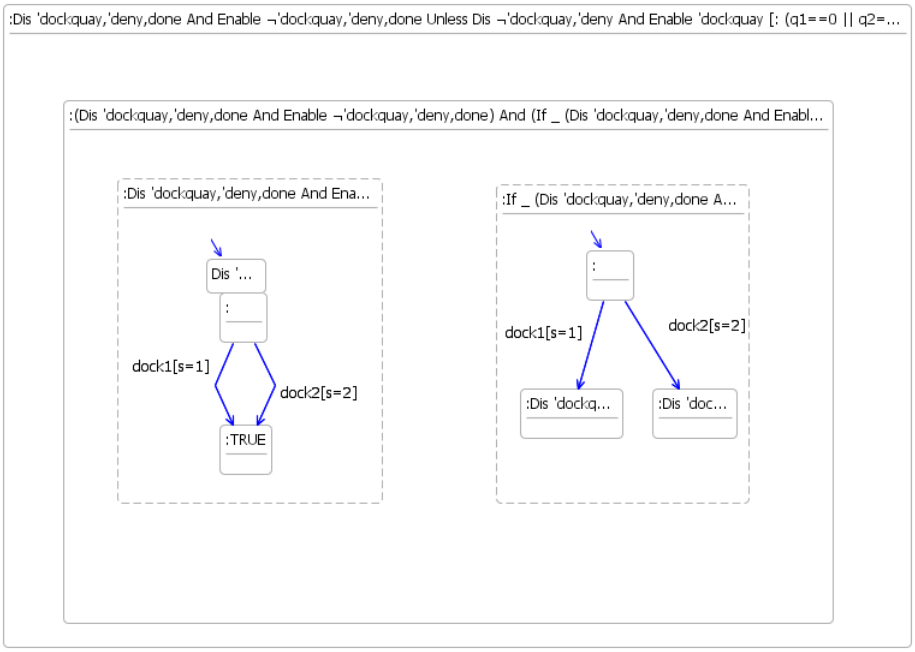

Figure C.8: If pattern applied to contract A2

12. Apply *Combine states* pattern to the new substates which both have contract A.

13. Apply the *Conjunction elimination* pattern to the state with contract $A1 \wedge A2$.

14. Apply the *Redundant hierarchy* pattern, the inner contract is assigned to the outer contract of the start state.

15. Apply *Redundant hierarchy* pattern, the inner contract A is assigned to the outer contract of the start state.

16. Further refine the substate with contract A. Apply the *Unless* pattern. The *Unless* pattern is applicable if the selected component is a state whose inner contract is an *Unless* operator expression. The pattern checks that the contract for the selected state has the form "ϕ *Unless* ψ" and in this instance the pattern refines the contract to "ψ" (see Figure C.9).

$$\mathbf{A} = ((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle -' dockquay,' deny\rangle} \wedge \langle[' dockquay[: R]]\rangle \wedge \overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle[' deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle' deny[: R]\rangle}))$$

$$\phi = ((\overline{\langle' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle)$$

$\psi = \overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$

$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$

$\overline{\langle 'deny[: R]\rangle}$

Apply the *Unless* pattern, contract A becomes $A3$.

A3 $= \overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[: R]]\rangle \wedge$

$\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$

$\overline{\langle 'deny[: R]\rangle}$

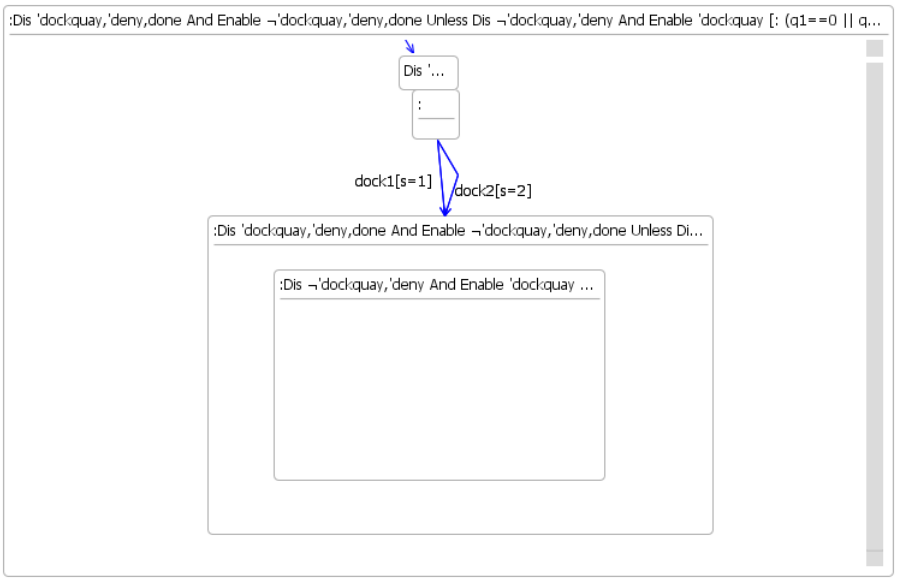

Figure C.9: Unfold Unless pattern applied to contract A

17. Apply the *Conjunction introduction* pattern to the state with contract A. This introduces two new states one with contract $(((A4 \wedge A5) \wedge A6) \wedge A7)$ and the other with contract $A8$.

A4 $= \overline{\langle -'dockquay,' deny\rangle}$

A5 $= \langle['dockquay[: R]]\rangle$

A6 $= \overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle}$

A7 $= \langle['deny[q_1, q_2, s : \neg R]]\rangle$

A8 $= \overline{\langle 'deny[: R]\rangle}$

18. Apply the *Conjunction introduction* pattern to the state with contract $((A4 \wedge A5) \wedge A6) \wedge A7$. This introduces two new states one with contract $((A4 \wedge A5) \wedge A6)$ and the other with contract $A7$.

19. Apply the *Conjunction introduction* pattern to the state with contract $(A4 \wedge A5) \wedge A6$. This introduces two new states one with contract $(A4 \wedge A5)$ and the other with contract $A6$.

20. Apply the *Conjunction introduction* pattern to the state with contract $A4 \wedge A5$. This introduces two new states with contracts $A4$ and $A5$.

21. Refine state with contract $A4$, apply the *Disable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

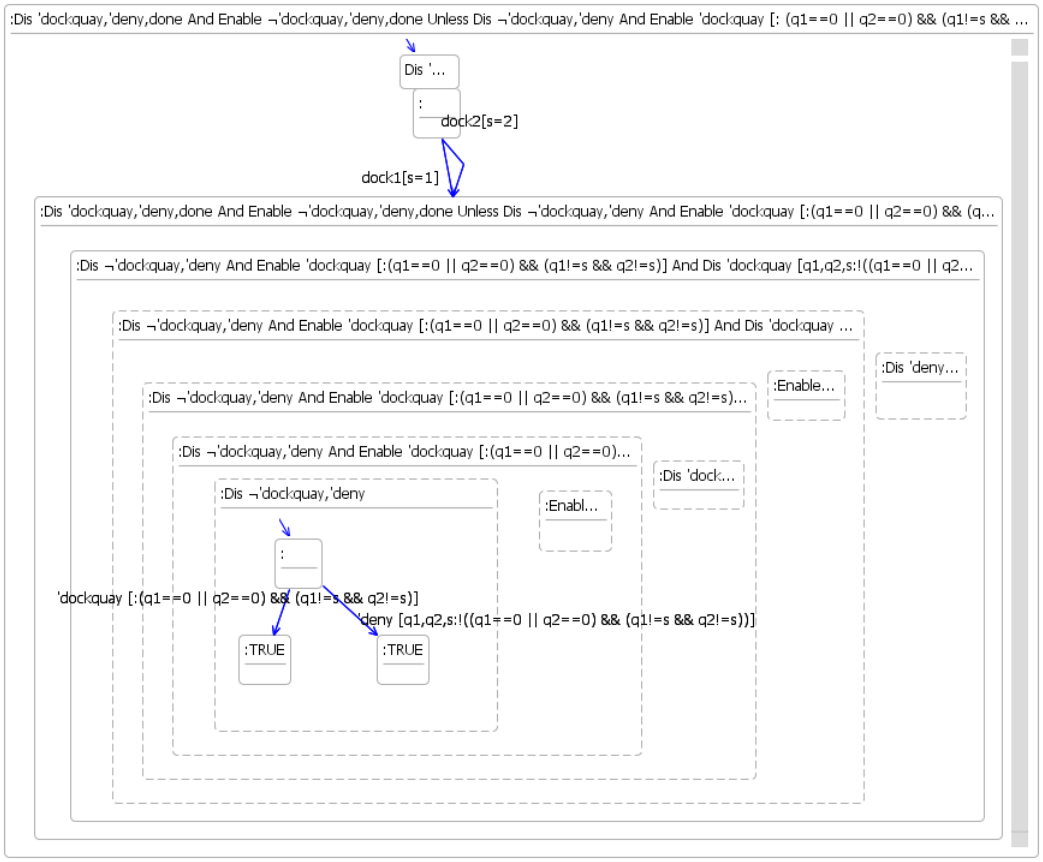

Figure C.10: Disable pattern applied to contract A4

A4 $= \overline{\langle -'dockquay,' deny\rangle}$

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*. To introduce transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, HST is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent

with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological.

The required new transitions are: '*dockquay*$[R \Longrightarrow skip]$ and '*deny*$[!R \Longrightarrow skip]$. The pattern confirms that the transition events are in the negated contract *eventlist* thus no further checks are required. The contract is replaced by a state machine with the new transitions each with subsequent behaviour *True* (see Figure C.10).

22. Refine state with contract $A5$, apply the *TotEnable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *TotEnable* pattern refines a contract to a state machine model. The *TotEnable* pattern is applicable if the selected component is a state. The side-condition checks that the state's inner contract has the form "*TotEnable eventFormula*" or "*TotEnable eventFormula [updateExpression]*". *EventFormula* can be a set of events, a negated set of events or the underscore character. The side-condition checks that at least one of the transition parameters satisfies the *TotEnable* contract.

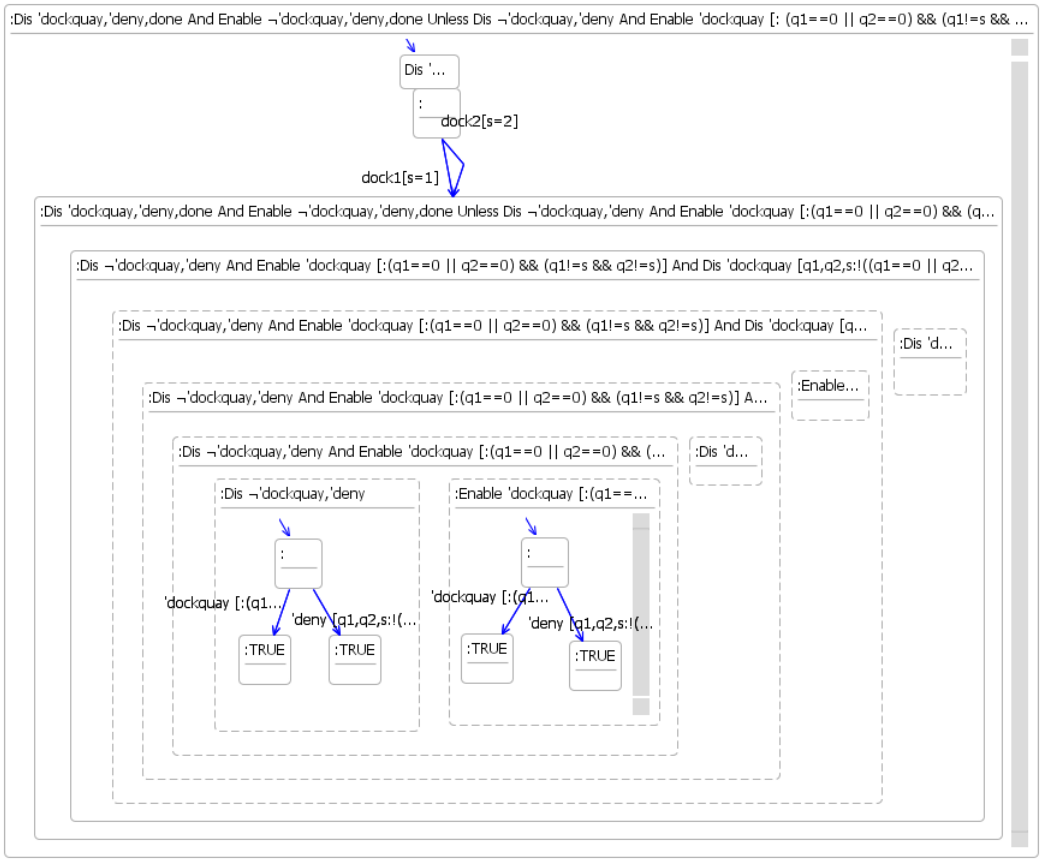

Figure C.11: TotEnable pattern applied to contract A5

Checks based on syntax are performed initially. To determine if the transition satisfies

the contract the transition event must be included in the contract's *eventlist*. If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the contract's update expression is syntactically equivalent to the transition's update expression then the transition satisfies the contract.

The contract is $\mathbf{A5} = \langle[' dockquay[: R]]\rangle$

The required new transitions are: '$dockquay[R \Longrightarrow skip]$ and '$deny[!R \Longrightarrow skip]$

The side-condition checks that at least one of the transition parameters satisfies the *TotEnable* contract. For the first transition '$dockquay[R \Longrightarrow skip]$, *'dockquay* is in the contract *eventlist* and the transition update expression is syntactically equivalent to the contract update expression. The pattern concludes that the transition satisfies the contract and the contract is replaced by a state machine with both new transitions (see Figure C.11).

23. Apply the *Conjunction elimination* pattern to the state with contract $A4 \wedge A5$.

24. Apply the *Redundant hierarchy* pattern to the state with contract $A5$. Inner contract $A4 \wedge A5$ is assigned to the outer contract of the start state.

25. Refine state with contract $A6$, apply the *Disable* pattern to introduce two new transitions *'dockquay* and *'deny*. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

$\mathbf{A6} = \overline{\langle' dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle}$

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*. To introduce transitions with subsequent behaviour *True* the pattern checks that the transition's event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, HST is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological.

The required new transitions are: '$dockquay[R \Longrightarrow skip]$ and '$deny[!R \Longrightarrow skip]$. For the first transition '$dockquay[R \Longrightarrow skip]$ the event is in the contract *eventlist* so HST is called to verify that the update expressions for the transition $[R \Longrightarrow skip]$ is not consistent with the contract update expression $[q_1, q_2, s : \neg(R \wedge skip)]$. HST confirms that $\neg((R \wedge skip) \wedge (\neg(R \wedge skip)))$ is tautological.

The transition is added with subsequent behaviour *True*. For the second transition '*deny*[$!R \Longrightarrow skip$] the pattern checks and confirms that the transition event *'deny* is not in the contract *eventlist* so no further checks are required (see Figure C.12).

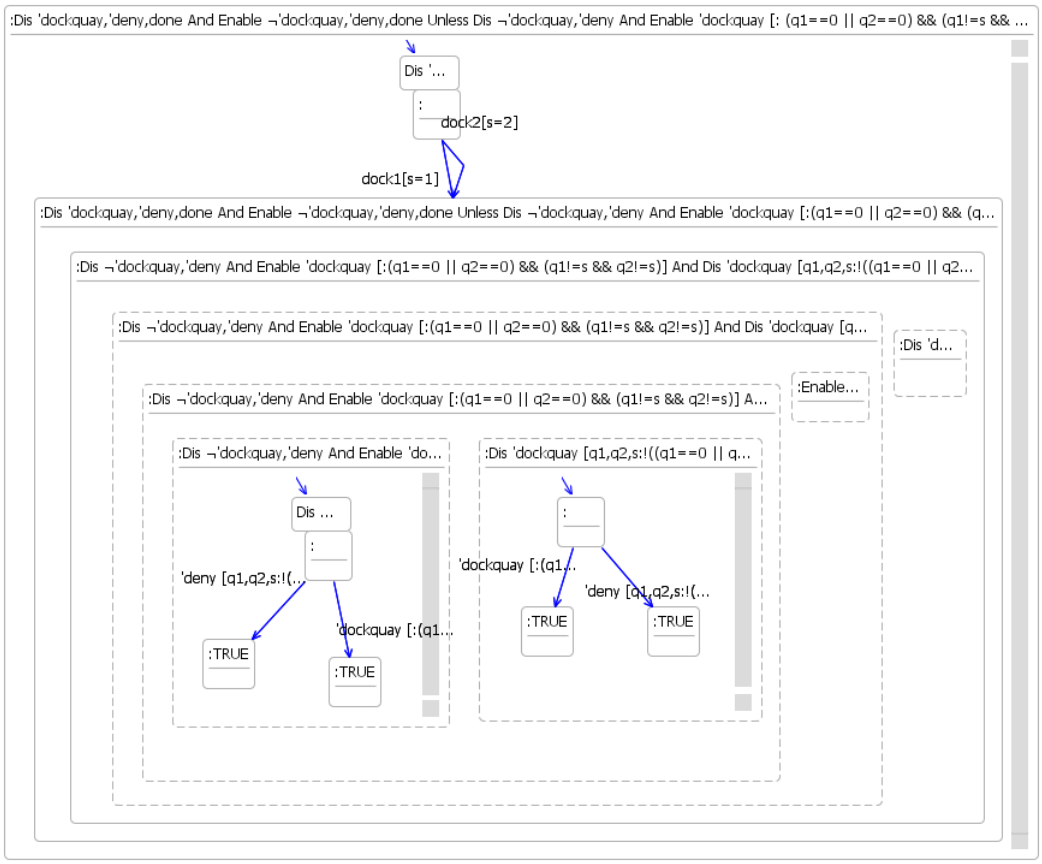

Figure C.12: Disable pattern applied to contract A6

26. Apply the *Conjunction elimination* pattern to the state with contract $(A4 \wedge A5) \wedge A6$.

27. Apply the *Redundant hierarchy* pattern to the state with contract $A6$. Inner contract $(A4 \wedge A5) \wedge A6$ is assigned to the outer contract of the start state.

28. Refine state with contract $A7$, apply the *TotEnable* pattern to introduce two new transitions *'dockquay* and *'deny*. The pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. Checks based on syntax are performed initially. To determine if the transition satisfies the contract the event must be an event included in the contract's *eventlist*.

If the *eventFormula* in the contract is a list of events, transitions with some event from the set are possible. If the *eventFormula* in the contract is a negated list of events, transitions with some event not in the set are possible. If the *eventFormula* is an underscore transitions with any event are possible. If the transition's event is in the contract's *eventlist* and the transition's update expression is consistent with the contract's update expression then the transition satisfies the contract.

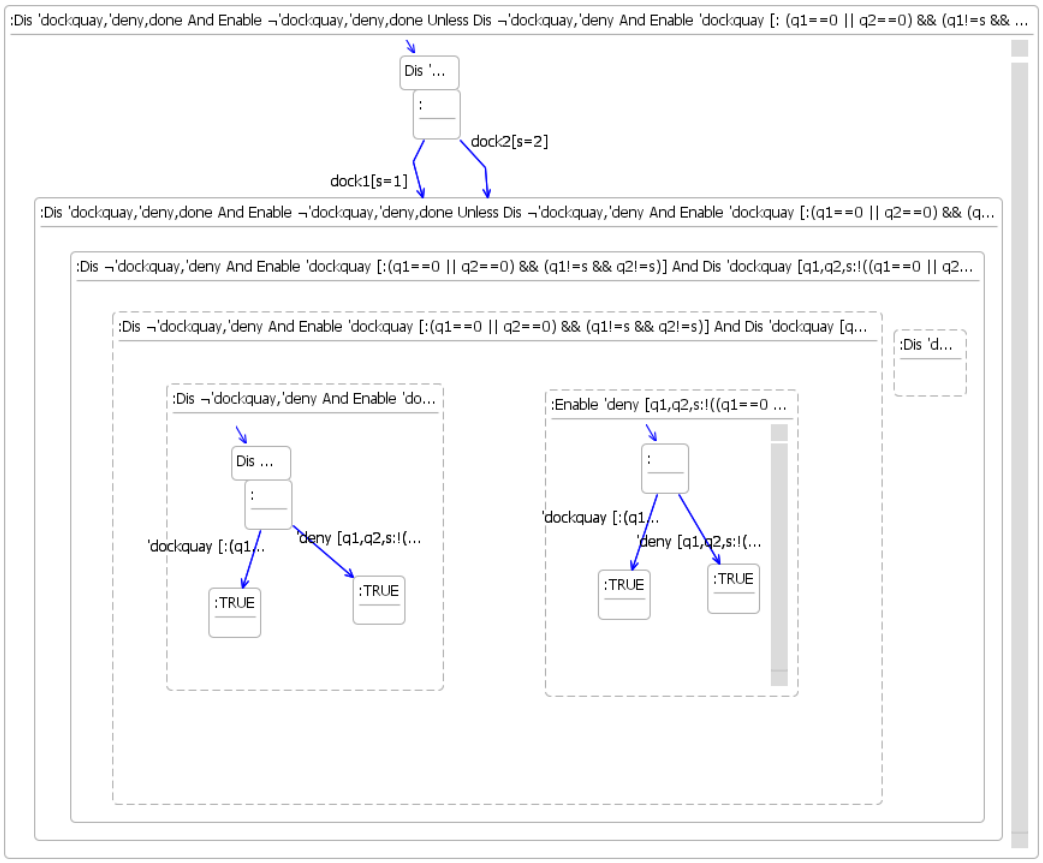

Figure C.13: TotEnable pattern applied to contract A7

It is a permitted refinement when introducing a transition to strengthen the contract's update expression within the guard. If the update expression in the contract places no restrictions on the possible after values of variables so that they could be updated in any way and the contract is of the form $\langle[eventlist[G]]\rangle$ where G is a guard and refers only to *before* values for variables, it is satisfied by transitions of the form $event[G]$, $event[G \Longrightarrow assignmentlist]$, $event[G \Longrightarrow skip]$.

A7 $= \langle['deny[q_1, q_2, s : \neg R]]\rangle$

The required new transitions are: $`dockquay[R \Longrightarrow skip]$ and $`deny[!R \Longrightarrow skip]$. The

pattern checks that at least one of the transition parameters satisfies the *TotEnable* contract. For the second transition $'deny[!R \Longrightarrow skip]$, $'deny$ is in the contract *eventlist*. The contract $A7$ says $'deny$ is enabled when $\neg R$ holds and the variables can be updated in any way. The transition says that a $'deny$ action is possible when $\neg R$ holds but the variable values remain the same. The transition's update expression strengthens the contract's update expression within the guard which is a permitted refinement. The transition satisfies the contract and the contract is replaced by a state machine with both new transitions (see Figure C.13).

29. Apply the *Conjunction elimination* pattern to the state with contract $((A4 \wedge A5) \wedge A6) \wedge A7$.

30. Apply the *Redundant hierarchy* pattern to the state with contract $A7$. Inner contract $((A4 \wedge A5) \wedge A6) \wedge A7$ is assigned to the outer contract of the start state.

31. Refine state with contract $A8$, apply the *Disable* pattern to introduce two new transitions $'dockquay$ and $'deny$. The *Disable* operator $\overline{\langle a_1, .., a_n[x : P]\rangle}$ is expressed in terms of the *If* operator $[a_1, .., a_n[x : P]]False$ and the *If* pattern is applied.

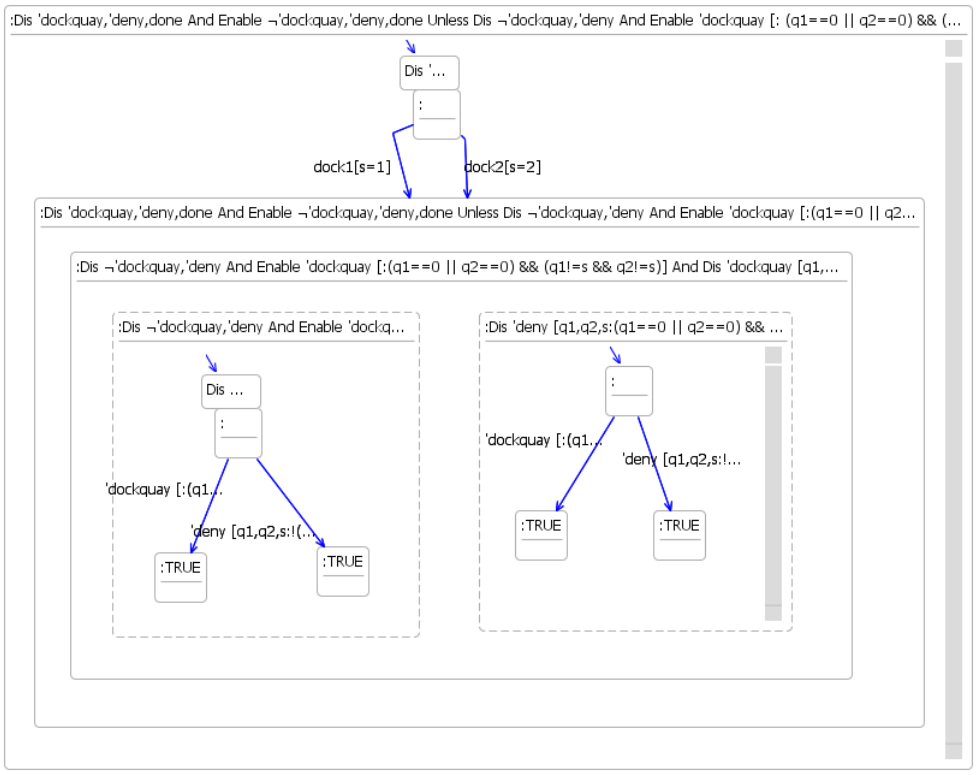

Figure C.14: Disable pattern applied to contract A8

A8 $= \overline{\langle' deny[:R]\rangle} = [' deny[:R]]False$

The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*. To introduce transitions with subsequent behaviour *True* the pattern checks that the transition's event is not in the contract *eventlist* (or it is in the negated contract *eventlist*) or if this is not the case or there is no *eventlist*, HST is used to verify that the update expression for the transition input by the user (updateExpr2) is not consistent with the contract update expression (updateExpr1), i.e. they are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological.

The required new transitions are: '$dockquay[R \Longrightarrow skip]$ and '$deny[!R \Longrightarrow skip]$. For the first transition '$dockquay[R \Longrightarrow skip]$ the pattern checks and confirms that the transition event '*dockquay* is not in the contract *eventlist* so no further checks are required. For the second transition '$deny[!R \Longrightarrow skip]$ the event is in the contract *eventlist* so HST is called to verify that the update expression for the transition $[!R \Longrightarrow skip]$ is not consistent with the contract update expression $[:R]$. HST confirms that $\neg((R \wedge skip) \wedge (\neg R \wedge skip))$ is tautological. The contract is replaced by a state machine with both transitions '$dockquay[R \Longrightarrow skip]$ and '$deny[!R \Longrightarrow skip]$ to states with contract *True* (see Figure C.14).

32. Apply the *Conjunction elimination* pattern to the state with contract A.

33. Apply the *Redundant hierarchy* pattern. The inner contract is assigned to the outer contract of the start state.

34. Apply the *Redundant hierarchy* pattern. Inner contract A is assigned to the outer contract of the start state (see Figure C.15).

35. Apply the *Move target down* and the *Redundant hierarchy* patterns to the state with contract A. The *Move target down* pattern shifts the target state of the incoming transition to the composite state down from the composite state to the default/initial state of the composite. The *Redundant hierarchy* pattern is applicable if the selected component is a state. The pattern checks that the state has substates and no incoming/outgoing transitions. The pattern flattens the level of hierarchy by removing the selected component but retaining its substates (see Figure C.16).

The overall design for the *ShipReq* component at this stage (see Figure C.17):

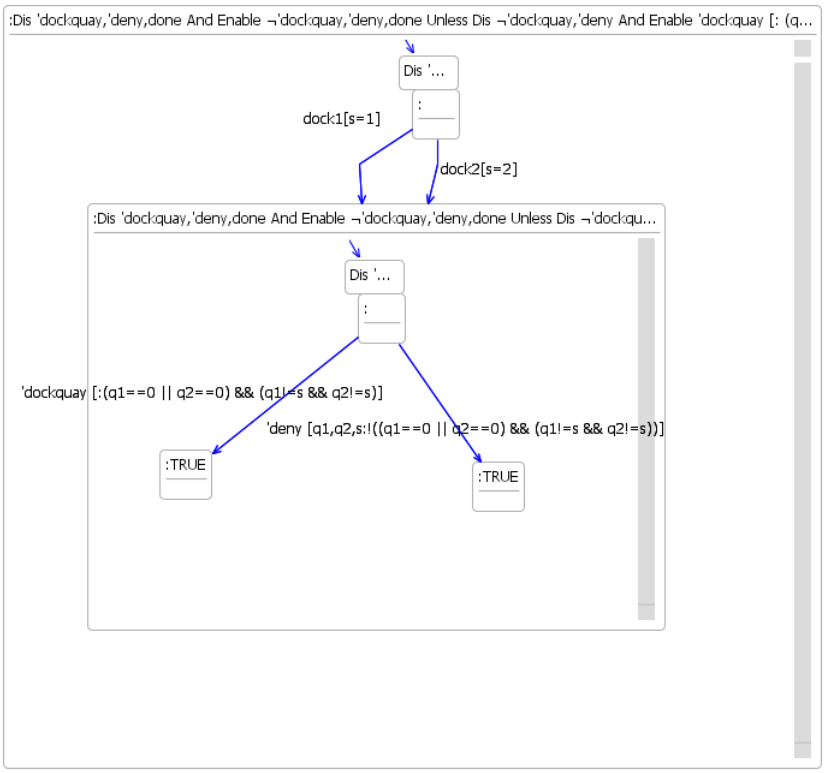

Figure C.15: Redundant hierarchy pattern applied to state with contract A3

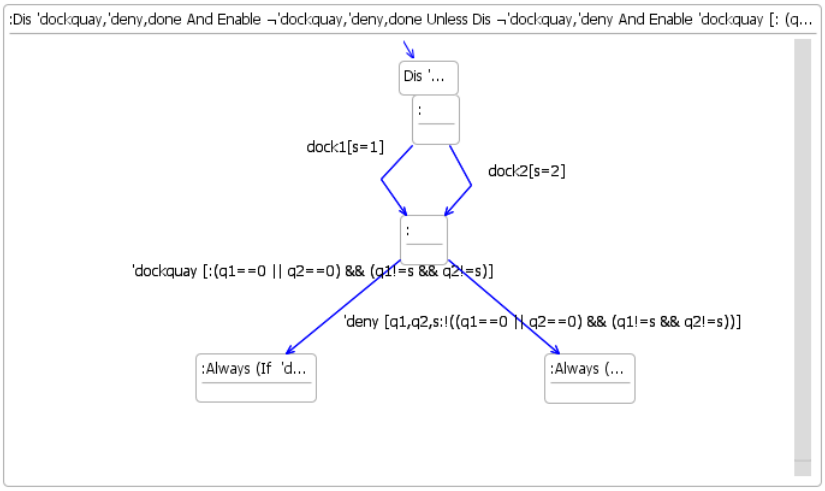

Figure C.16: Move target down and Redundant hierarchy applied to contract A

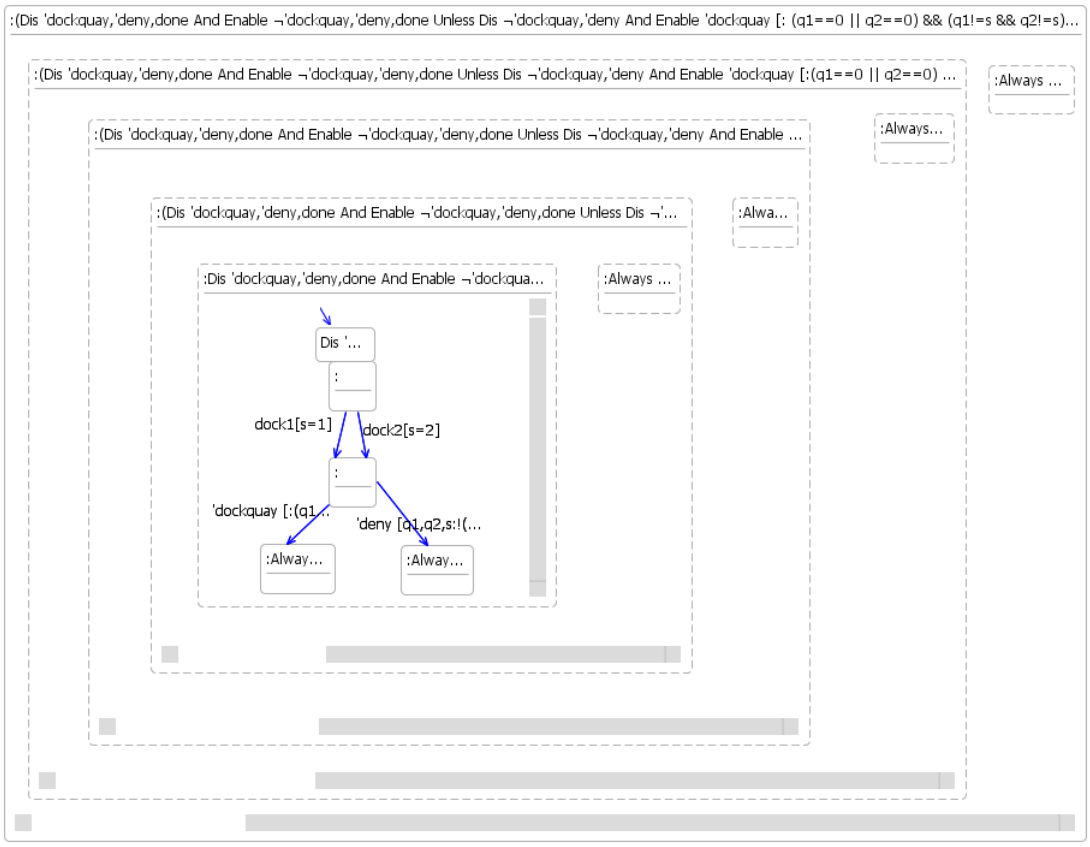

Figure C.17: The ShipReq component

36. Refine the conjunction state with contract B. Apply the *Unfold Always* pattern. The *Unfold Always* pattern is applicable if the selected component is a state whose inner contract is an *Always* operator expression. The pattern checks that the contract for the selected state has the form "*Always* ϕ" and the pattern unfolds the contract to "ϕ *And (If _ (Always* ϕ*))*".

$$\begin{aligned}\mathbf{B} = \Box[done]((&\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle'deny[:R]\rangle}))\end{aligned}$$

B becomes $B1 \wedge B2$

$$\begin{aligned}\mathbf{B1} = [done]((&\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle'deny[:R]\rangle}))\end{aligned}$$

$$\begin{aligned}\mathbf{B2} = [\_](\Box[done]((&\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle'deny[:R]\rangle})))\end{aligned}$$

37. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

38. Apply the *If* pattern to the state with contract $B1$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ". *EventFormula* can be a set of events, a negated set of events or the underscore character. For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist* the side-condition verifies that the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1). The side-condition checks that the update expressions are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological. The *If* pattern does not permit the introduction of transitions with subsequent behaviour *False*.

It is not necessary to perform any checks for transitions with subsequent behaviour "ϕ",

as if the conditions for the subsequent behaviour "ϕ" do not hold, then the subsequent behaviour is *True* and this contract can then be refined to "ϕ". A permitted refinement is to weaken the update expression.

$$\begin{aligned}\mathbf{B1} = [done]((&\overline{\langle{}'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle'deny[:R]\rangle}))\end{aligned}$$

The two new transitions required are: $dock1[s = 1]$ and $dock2[s = 2]$, their subsequent behaviour will be *True*. The pattern checks and confirms that the action for each transition is not in the contract *eventlist* (see Figure C.18).

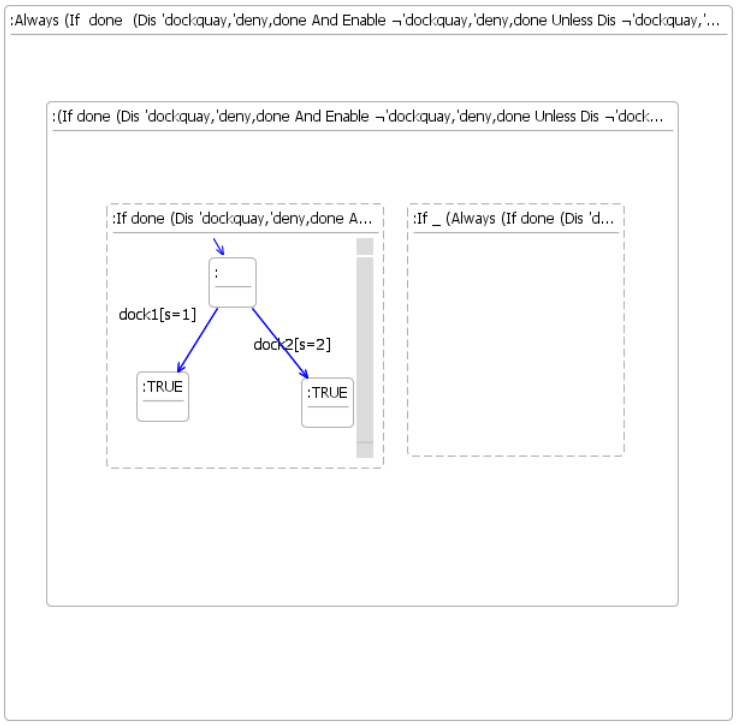

Figure C.18: If pattern applied to introduce *dock1* and *dock2*

39. Apply the *If* pattern to the state with contract $B2$ to introduce two new transitions *dock1* and *dock2*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. The contract has the form "*If eventFormula* ϕ" or "*If eventFormula [updateExpression]* ϕ". *EventFormula* can be a set of events, a negated set of events or the underscore character. For each of the transitions with subsequent behaviour *True*, the pattern checks that the

transition event is not in the contract *eventlist* (or it is in the the negated contract *eventlist*) or if this is not the case or there is no *eventlist* the side-condition verifies that the update expression for the transition (updateExpr2) is not consistent with the contract update expression (updateExpr1). The side-condition checks that the update expressions are disjoint and there is no overlap. HST is invoked to decide if !(updateExpr2 && updateExpr1) is tautological (see Figure C.19).

The contract is: $\mathbf{B2} = [\_]\Box[done]((\overline{\langle'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)$

$$\curvearrowright (\overline{\langle-'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge$$
$$\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle'deny[:R]\rangle}))$$

The two new transitions required are, $dock1[s = 1]$ and $dock2[s = 2]$.

The contract $B2$ has the form $[\_]\phi$, all enabled transitions subsequently behave like ϕ.

$\phi = \Box[done]((\overline{\langle'dockquay,'\,deny, done\rangle} \wedge \langle[-'dockquay,'\,deny, done]\rangle)$

$$\curvearrowright (\overline{\langle-'dockquay,'\,deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge$$
$$\overline{\langle'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge$$
$$\overline{\langle'deny[:R]\rangle}))$$

The contract states that all enabled transitions subsequently behave like ϕ. The contract is replaced by a state machine with transitions $dock1[s = 1]$ and $dock2[s = 2]$ that subsequently behave like ϕ.

40. Apply the *Combine states* pattern to the new substates of $B1$ with the same contracts.

41. Apply the *Combine states* pattern to the new substates of $B2$ with the same contracts.

42. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

43. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract B of the parent state to the outer contract of the start state (see Figure C.20).

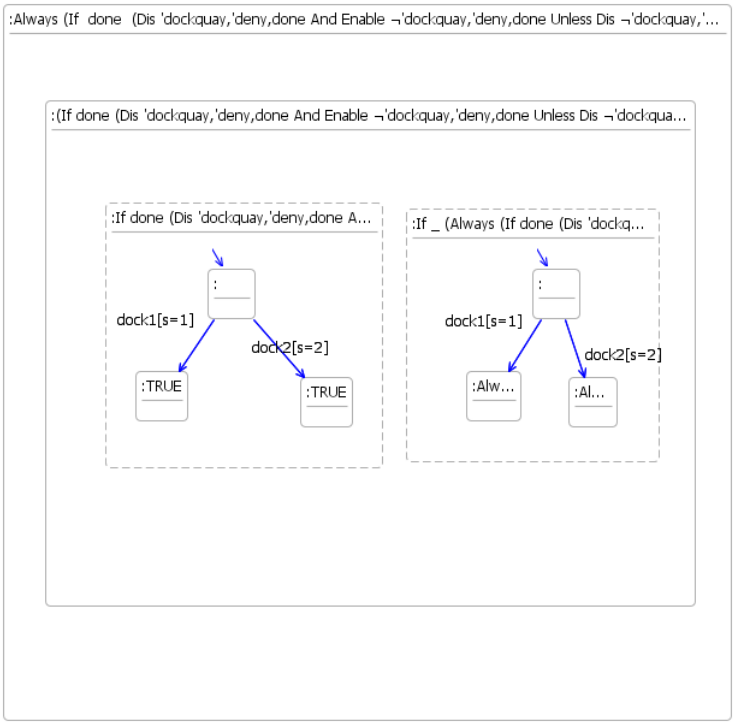

Figure C.19: If pattern applied to contract B2

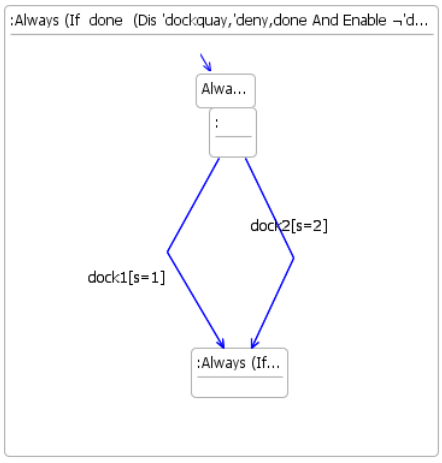

Figure C.20: Redundant hierarchy pattern applied twice

44. Further refine the substate with contract B. Apply the *Unfold Always* pattern, contract B becomes $B1 \wedge B2$.

45. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

46. Apply the *If* pattern to the state with contract $B1$ to introduce two new transitions *dockquay* and *deny*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression. For each of the transitions with subsequent behaviour *True*, the pattern checks that the transition event is not in the contract *eventlist*.

$$\begin{aligned}\mathbf{B1} = [done]((&\overline{\langle {}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle 'deny[:R]\rangle}))\end{aligned}$$

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$, their subsequent behaviour will be *True*. The pattern checks and confirms that the event for each transition is not in the contract *eventlist* (see Figure C.21).

47. Apply the *If* pattern to the state with contract $B2$ to introduce two new transitions *'dockquay* and *'deny*. The *If* pattern is applicable if the selected component is a state. The pattern checks that the inner contract of the state is an *If* operator expression.

The contract is: $\mathbf{B2} = [\_]\Box[done]((\overline{\langle 'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)$

$$\begin{aligned}&\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle 'deny[:R]\rangle}))\end{aligned}$$

The required new transitions are: $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$.

The contract $B2$ has the form $[\_]\phi$, *all* enabled transitions subsequently behave like ϕ.

$$\begin{aligned}\phi = \Box[done]((&\overline{\langle {}'dockquay,' deny, done\rangle} \wedge \langle[-'dockquay,' deny, done]\rangle)\\ &\curvearrowright (\overline{\langle -'dockquay,' deny\rangle} \wedge \langle['dockquay[:R]]\rangle \wedge\\ &\overline{\langle 'dockquay[q_1, q_2, s : \neg(R \wedge skip)]\rangle} \wedge \langle['deny[q_1, q_2, s : \neg R]]\rangle \wedge\\ &\overline{\langle 'deny[:R]\rangle}))\end{aligned}$$

The contract is replaced by a state machine with the transitions $'dockquay[R \Longrightarrow skip]$ and $'deny[!R \Longrightarrow skip]$ that subsequently behave like ϕ (see Figure C.22).

48. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

49. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

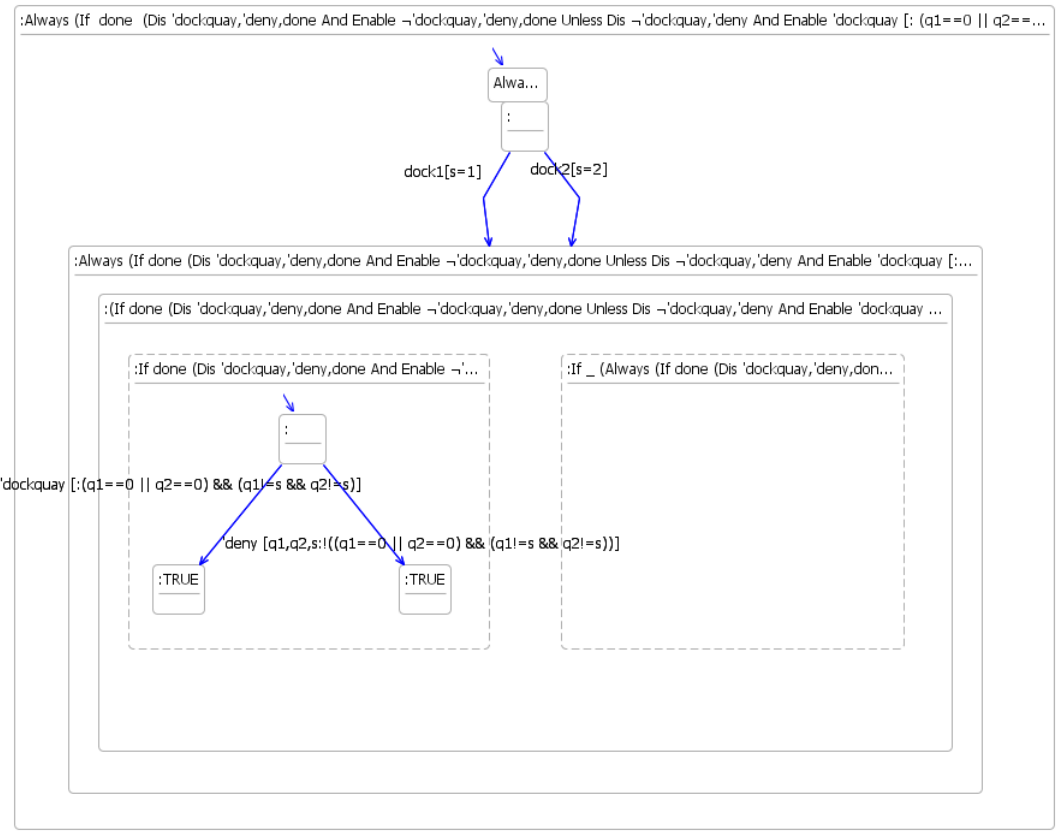

Figure C.21: If pattern applied to contract B1

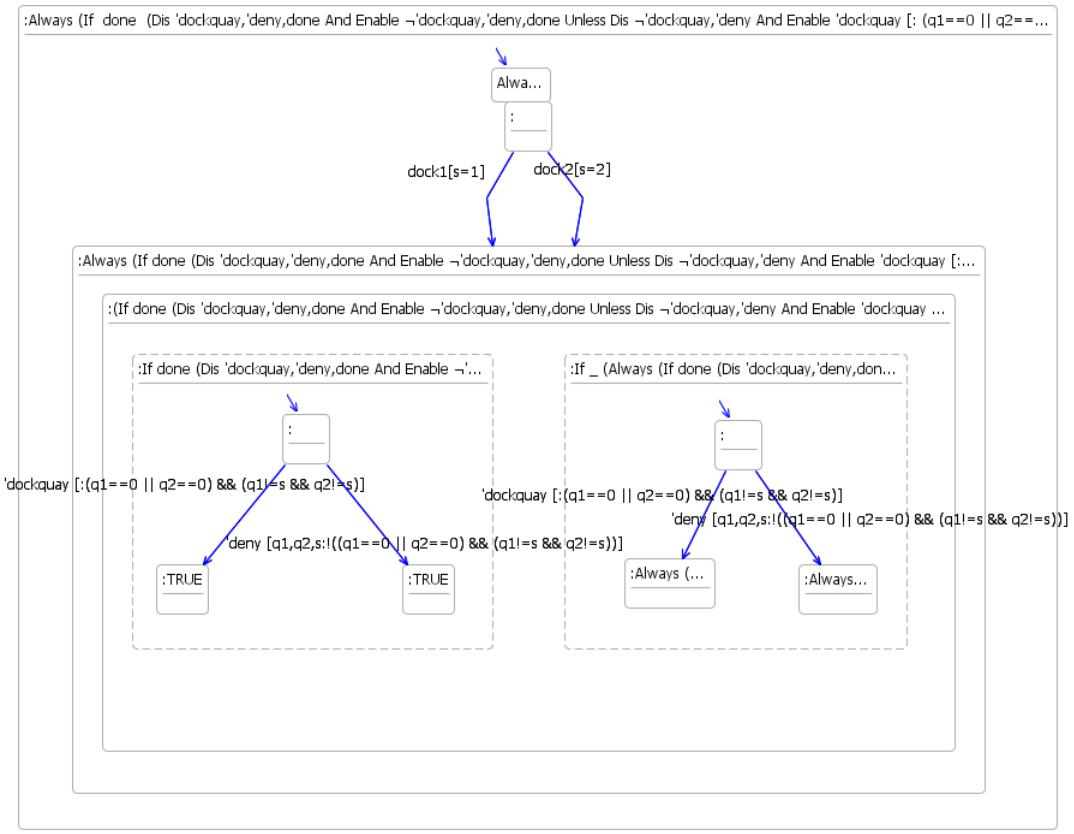

Figure C.22: If pattern applied to introduce *dockquay* and *deny*

50. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent state to the outer contract of the start state (see Figure C.23).

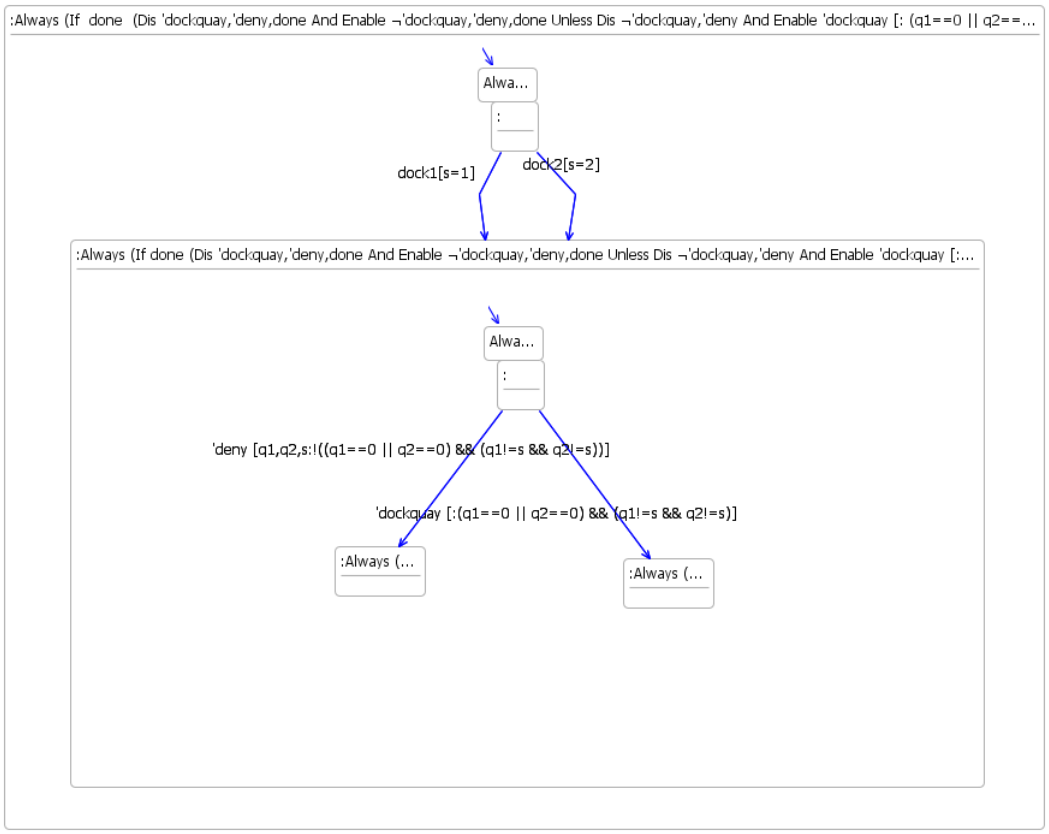

Figure C.23: Redundant hierarchy pattern applied to state with contract B1

51. Apply the *Move target down* and the *Redundant hierarchy* patterns to the state with contract B. The *Move target down* pattern shifts the target state of the incoming transition to the composite state down from the composite state to the default/initial state of the composite. The *Redundant hierarchy* pattern is applicable if the selected component is a state. The pattern checks that the state has substates and no incoming/outgoing transitions. The pattern flattens the level of hierarchy by removing the selected component but retaining its substates (see Figure C.24).

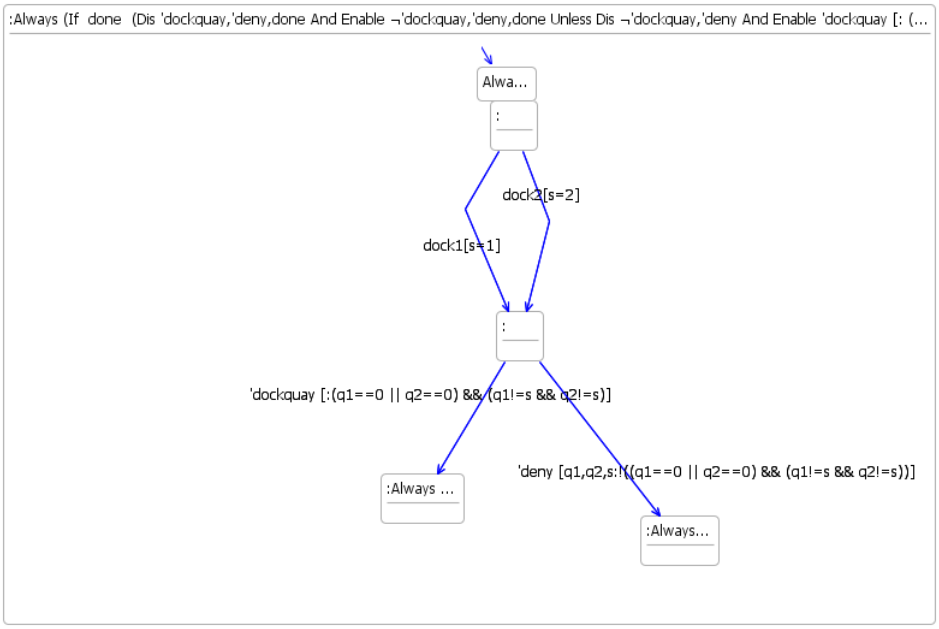

Figure C.24: Move target down and Redundant hierarchy patterns applied

The overall design for the *ShipReq* component at this stage (see Figure C.25):

52. Apply the *Conjunction elimination* pattern to the state with contract $A \wedge B$.

53. Apply the *Redundant hierarchy* pattern. Assigns the inner contract $A \wedge B$ of the parent state to the outer contract of the start state (see Figure C.26).

54. Substate with contract B and incoming transition *'dockquay* is further refined. Apply the *Unfold Always* pattern, B becomes $B1 \wedge B2$

55. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

56. Apply the *If* pattern to the state with contract $B1$ to introduce a new transition, *done*.

57. Apply the *If* pattern to the state with contract $B2$ to introduce a new transition *done*.

58. Further refine the substate of $B1$ with contract A. Apply the *Unfold Unless* pattern. Contract A becomes $A1 \wedge A2$.

59. Apply the *Conjunction introduction* pattern to the state with contract $A1 \wedge A2$.

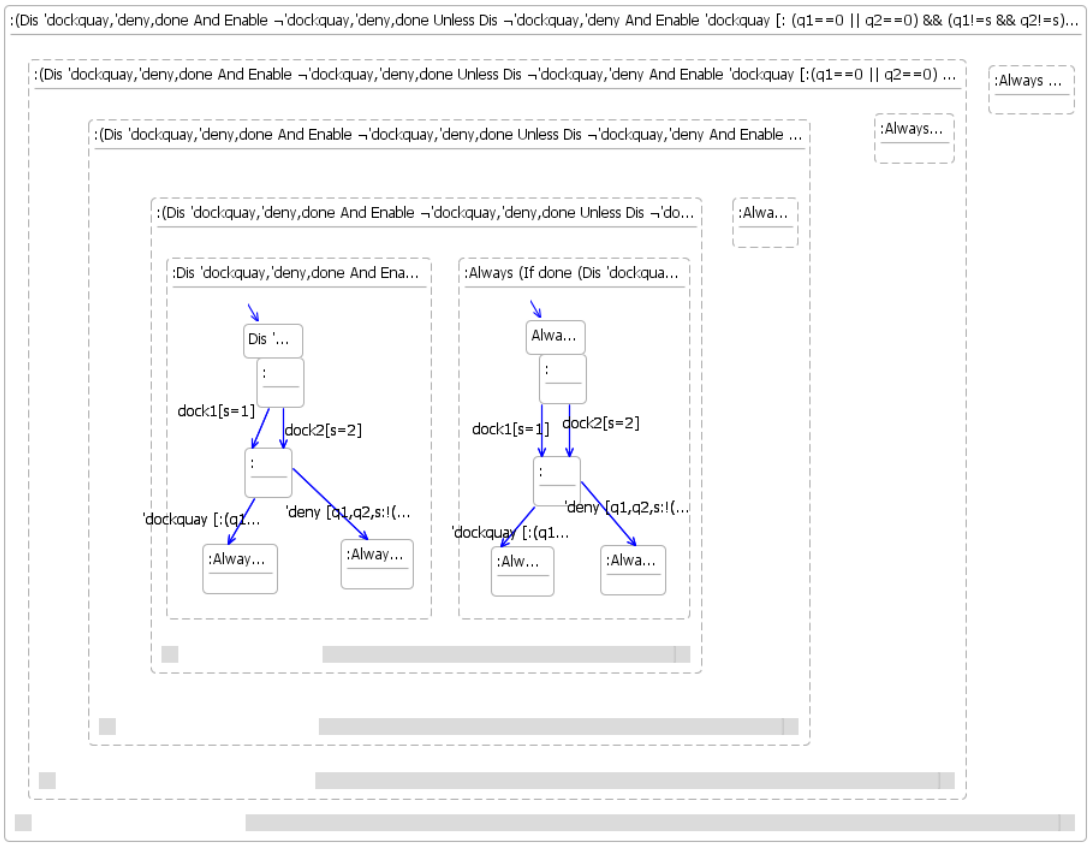

Figure C.25: The ShipReq component

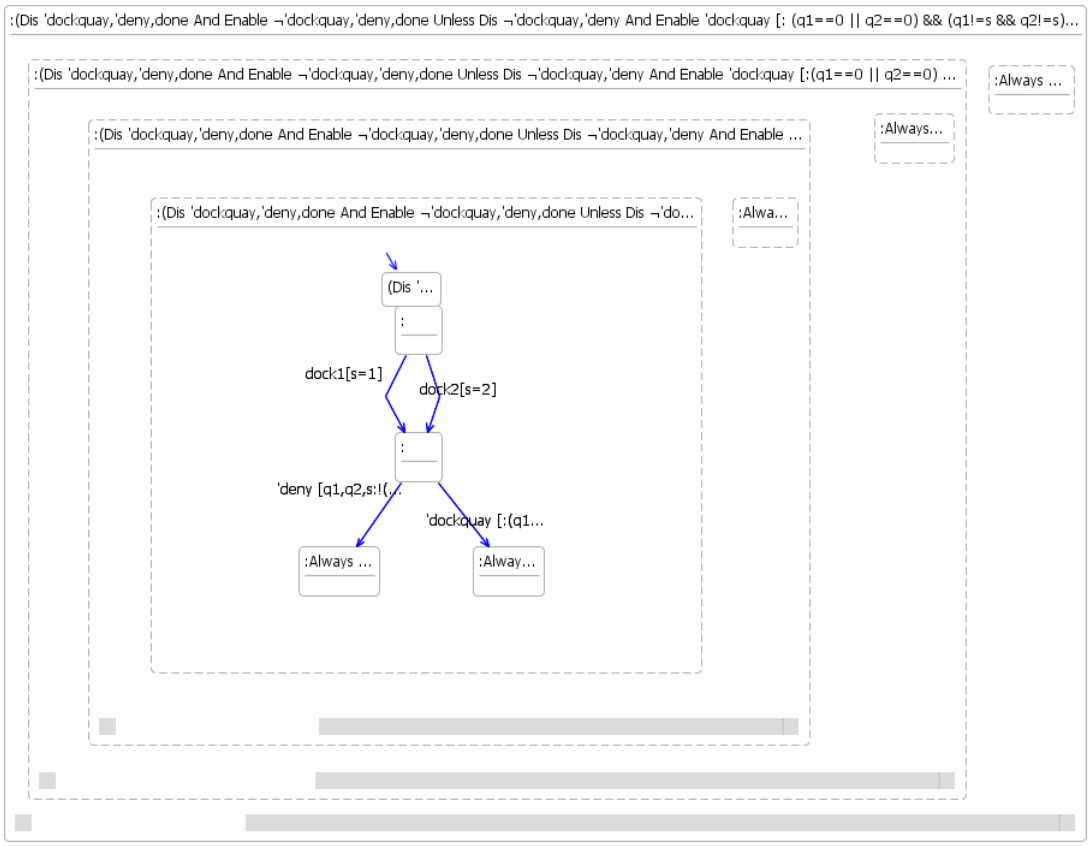

Figure C.26: Conjunction elimination and Redundant hierarchy patterns applied

60. Apply the *Conjunction introduction* pattern to the state with contract $A1$. Introduces two new states with contracts $A9$ and $A10$.

61. Apply the *Disable* pattern to the state with contract $A9$ to introduce the *docked* transition.

62. Apply the *TotEnable* pattern to the state with contract $A10$ to introduce the *docked* transition.

63. Apply the *Conjunction elimination* pattern to the state with contract $A1$.

64. Apply pattern *Redundant hierarchy*, inner contract $A1$ is assigned to outer contract of start state.

65. Apply the *If* pattern to state with contract $A2$ to introduce the *docked* transition.

66. Apply the *Conjunction elimination* pattern to the state with contract $A1 \wedge A2$.

67. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

68. Apply the *Redundant hierarchy* pattern. Assigns the inner contract A of the parent state to the outer contract of the start state.

69. Further refine the substate of $B2$ that is the target of the *done* transition with contract B. Apply the *Unfold Always* pattern, contract B becomes $B1 \wedge B2$.

70. Apply the *Conjunction introduction* pattern to the state with contract $B1 \wedge B2$.

71. Apply the *If* pattern to the state with contract $B1$ to introduce the *docked* transition.

72. Apply the *If* pattern to the state with contract $B2$ to introduce the *docked* transition.

73. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

74. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

75. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent state to the outer contract of the start state.

76. Apply the *Move target down* and the *Redundant hierarchy* patterns to both the target states of the *done* transitions.

77. Apply the *Conjunction elimination* pattern to the state with contract $B1 \wedge B2$.

78. Apply the *Redundant hierarchy* pattern. Assigns the inner contract of the parent state to the outer contract of the start state.

79. Apply the *Redundant hierarchy* pattern. Assigns the inner contract B of the parent state to the outer contract of the start state (see Figure C.27).

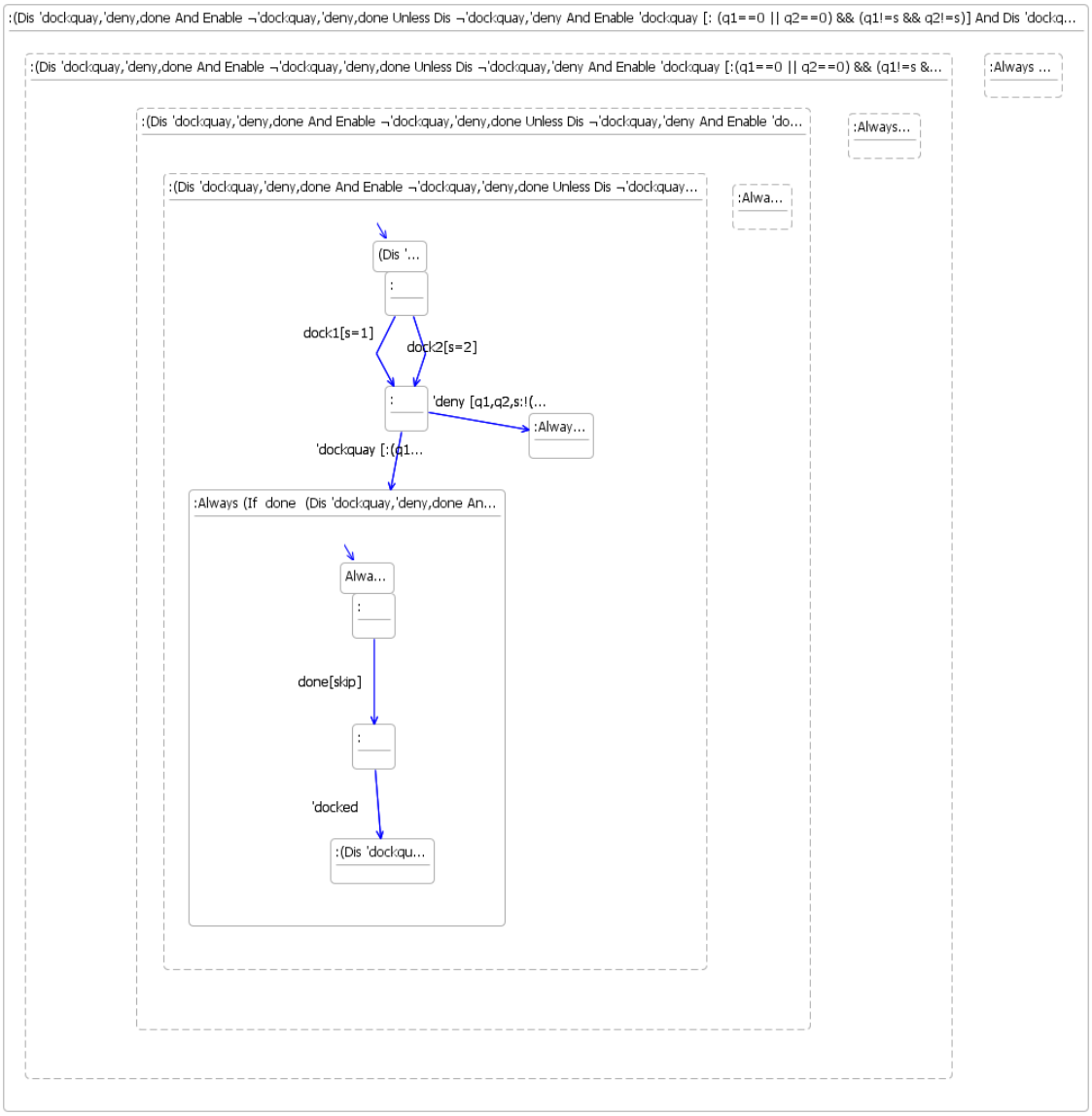

Figure C.27: Conjunction elimination followed by Redundant hierarchy

80. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target state of the *dockquay* transition (see Figure C.28).

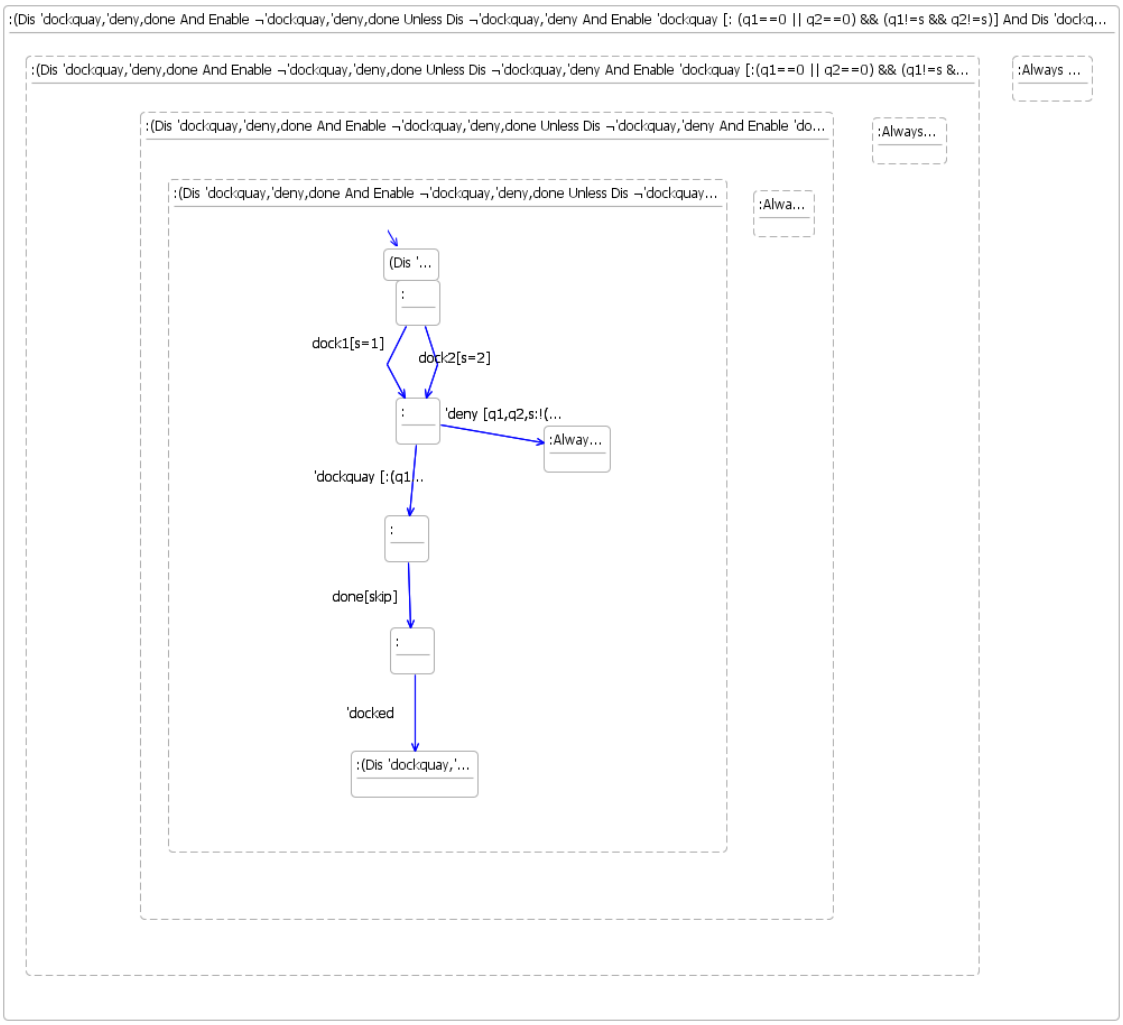

Figure C.28: Move target down and the Redundant hierarchy patterns applied

81. Apply the *Reroute* pattern to the target of the *docked* transition. This reroutes the *docked* transition to the start state which has the same contract. The *Reroute* pattern is applicable if the selected component is a state. The pattern checks that the state has incoming transitions but no outgoing transitions or substates. The pattern removes the selected state and reroutes its incoming transitions to another state, chosen by the user, whose outer contract is equivalent to the inner contract of the selected state. The pattern performs a syntactic check to determine if the contracts are equivalent (see Figure C.29).

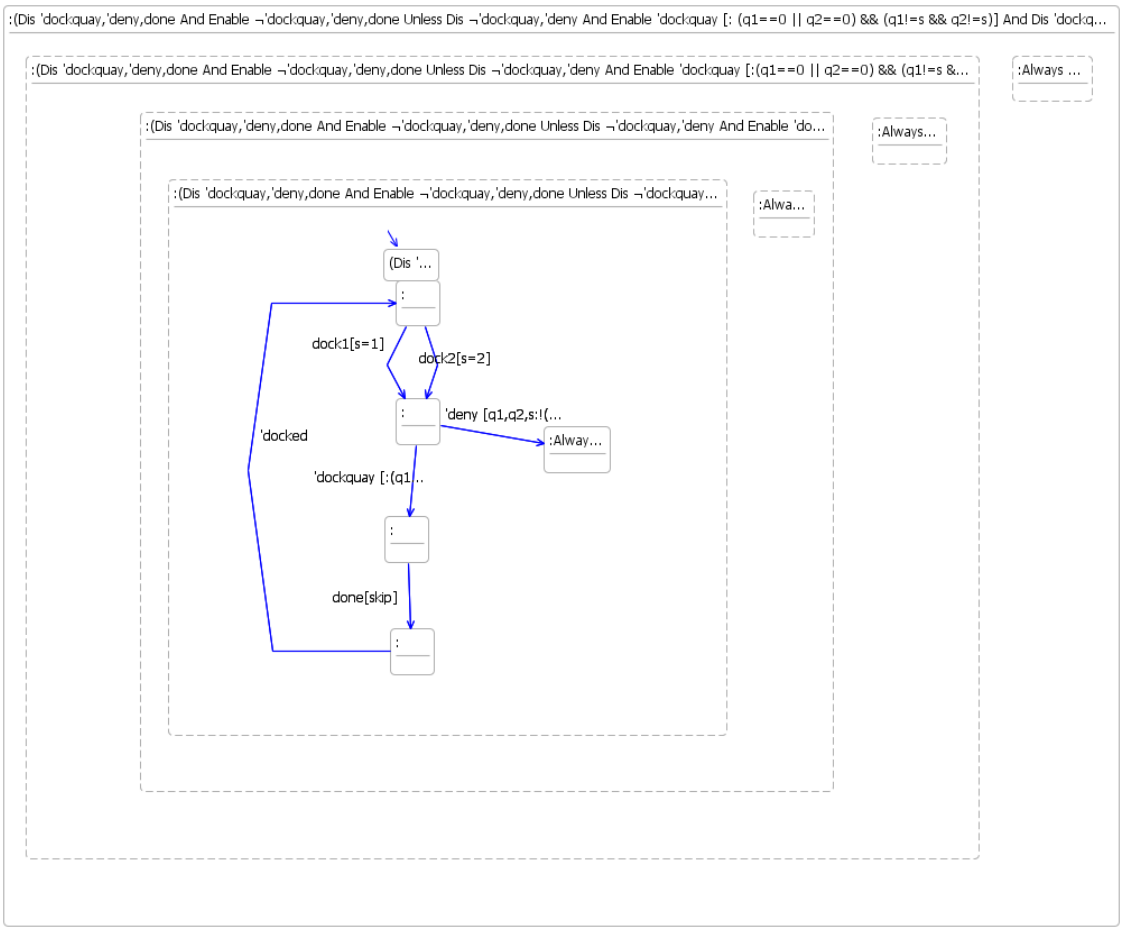

Figure C.29: Reroute pattern applied to target of *docked* transition

82. Refine state with contract C. Apply pattern *Unfold Always*, contract C becomes $C1 \wedge C2$.

$$\mathbf{C} = \Box[']deny](((\overline{\langle']dockquay,' deny, done\rangle} \wedge \langle[-']dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle-']dockquay,' deny\rangle} \wedge \langle[']dockquay[: R]]\rangle \wedge \overline{\langle']dockquay[q_1, q_2, s : \neg(R \wedge skip)]]\rangle} \wedge \langle[']deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle']deny[: R]]\rangle}))$$

$$\mathbf{C1} = [']deny](((\overline{\langle']dockquay,' deny, done\rangle} \wedge \langle[-']dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle-']dockquay,' deny\rangle} \wedge \langle[']dockquay[: R]]\rangle \wedge \overline{\langle']dockquay[q_1, q_2, s : \neg(R \wedge skip)]]\rangle} \wedge \langle[']deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle']deny[: R]]\rangle}))$$

$$\mathbf{C2} = [\_]\Box[']deny](((\overline{\langle']dockquay,' deny, done\rangle} \wedge \langle[-']dockquay,' deny, done]\rangle) \curvearrowright (\overline{\langle-']dockquay,' deny\rangle} \wedge \langle[']dockquay[: R]]\rangle \wedge \overline{\langle']dockquay[q_1, q_2, s : \neg(R \wedge skip)]]\rangle} \wedge \langle[']deny[q_1, q_2, s : \neg R]]\rangle \wedge \overline{\langle']deny[: R]]\rangle}))$$

83. Apply the *Conjunction introduction* pattern to state with contract $C1 \wedge C2$. Creates two new substates one with contract $C1$ and the other with contract $C2$.

84. Apply the *If* pattern to the state with contract $C1$ to introduce two new transitions *dock1* and *dock2*.

85. Apply the *Combine states* pattern to combine the substates of $C1$ with contract *True*.

86. Apply the *If* pattern to the state with contract $C2$ to introduce two new transitions *dock1* and *dock2*.

87. Apply the *Combine states* pattern to combine the substates of $C2$ with contract *True*.

88. Apply the *Conjunction elimination* pattern to the state with contract $C1 \wedge C2$.

89. Apply pattern *Redundant hierarchy* twice, inner contract C is assigned to the outer contract of the start state.

90. Further refine the substate of C with contract C. Apply the *Unfold Always* pattern, contract C becomes $C1 \wedge C2$.

91. Apply the *Conjunction introduction* pattern to the state with contract $C1 \wedge C2$. Introduces two new states with contracts $C1$ and $C2$.

92. Apply the *If* pattern to the state with contract $C1$ to introduce two new transitions *dockquay* and *deny*.

93. Apply the *If* pattern to the state with contract $C2$ to introduce two new transitions *dockquay* and *deny*.

94. Apply the *Conjunction elimination* pattern to the state with contract $C1 \wedge C2$.

95. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract C of the parent state to the outer contract of the start state.

96. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dock1* transition.

97. Further refine the target state of the *dockquay* transition with contract C. Apply the *Unfold Always* pattern, contract C becomes $C1 \wedge C2$.

98. Apply the *Conjunction introduction* pattern to state with contract $C1 \wedge C2$. Creates two new substates one with contract $C1$ and the other with contract $C2$.

99. Apply the *If* pattern to the state with contract $C1$ to introduce the transition *done*.

100. Apply the *If* pattern to the state with contract $C2$ to introduce the transition *done*.

101. Apply the *Conjunction elimination* pattern to the state with contract $C1 \wedge C2$.

102. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract C of the parent state to the outer contract of the start state.

103. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dockquay* transition.

104. Further refine the target state of the *done* transition with contract C. Apply the *Unfold Always* pattern, contract C becomes $C1 \wedge C2$.

105. Apply the *Conjunction introduction* pattern to state with contract $C1 \wedge C2$. Creates two new substates one with contract $C1$ and the other with contract $C2$.

106. Apply the *If* pattern to the state with contract $C1$ to introduce the transition *docked*.

107. Apply the *If* pattern to the state with contract $C2$ to introduce the transition *docked*.

108. Apply the *Conjunction elimination* pattern to the state with contract $C1 \wedge C2$.

109. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract C of the parent state to the outer contract of the start state.

110. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *done* transition.

111. Apply the *Reroute* pattern to the target of the *docked* transition. This reroutes the *docked* transition to the start state which has the same contract (see Figure C.30).

112. Apply the *Conjunction elimination* pattern to the state with contract $A \wedge B \wedge C$ and *Redundant hierarchy*. Assigns the inner contract $A \wedge B \wedge C$ of the parent state to the outer contract of the start state (see Figure C.31).

113. Refine state with contract D. Apply pattern *Unfold Always*, contract D becomes

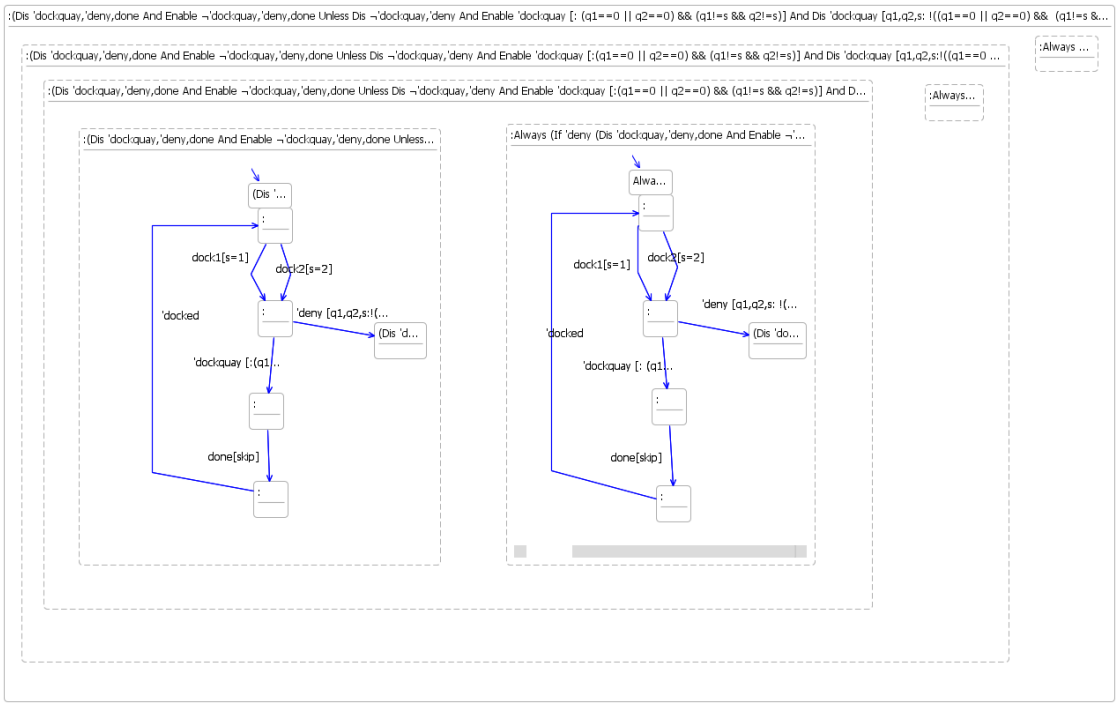

Figure C.30: Reroute pattern applied

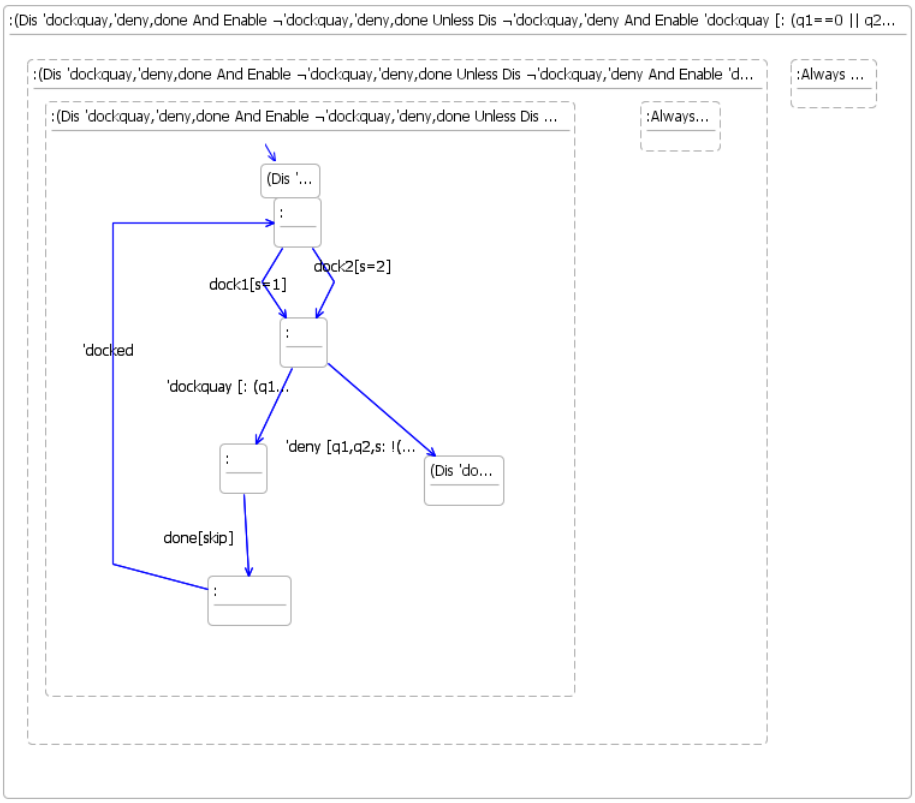

Figure C.31: Conjunction elimination and Redundant hierarchy patterns applied

$D1 \wedge D2$.

$$\mathbf{D} = \Box[' dockquay](((\overline{\langle ' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[s : s! = s']\rangle} \curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$$

$$\mathbf{D1} = [' dockquay](((\overline{\langle ' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[s : s! = s']\rangle} \curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$$

$$\mathbf{D2} = [\_]\Box[' dockquay](((\overline{\langle ' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[s : s! = s']\rangle} \curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$$

114. Apply the *Conjunction introduction* pattern to state with contract $D1 \wedge D2$. Creates two new substates one with contract $D1$ and the other with contract $D2$.

115. Apply the *If* pattern to the state with contract $D1$ to introduce two new transitions *dock1* and *dock2*.

116. Apply the *Combine states* pattern to combine the substates of $D1$ with contract *True*.

117. Apply the *If* pattern to the state with contract $D2$ to introduce two new transitions *dock1* and *dock2*.

118. Apply the *Combine states* pattern to combine the substates of $D2$ with contract D.

119. Apply the *Conjunction elimination* pattern to the state with contract $D1 \wedge D2$.

120. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract D of the parent state to the outer contract of the start state.

121. Further refine the target state of the *dock1* transition with contract D. Apply the *Unfold Always* pattern, contract D becomes $D1 \wedge D2$.

122. Apply the *Conjunction introduction* pattern to state with contract $D1 \wedge D2$. Creates two new substates one with contract $D1$ and the other with contract $D2$.

123. Apply the *If* pattern to the state with contract $D1$ to introduce two new transitions *dockquay* and *deny*.

124. Apply the *If* pattern to the state with contract $D2$ to introduce two new transitions *dockquay* and *deny*.

125. Further refine the substate of $D1$ that is the target of the *dockquay* transition with contract $D3$. Apply the *Unless* pattern. Contract $D3$ is refined to $D4$.

$$\mathbf{D3} = ((\overline{\langle ' dockquay,' deny, done\rangle} \wedge \langle[-' dockquay,' deny, done]\rangle) \wedge \overline{\langle \_[s : s! = s']\rangle} \curvearrowright (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle}))$$

$$\mathbf{D4} = (\overline{\langle -done\rangle} \wedge \langle[done[skip]]\rangle \wedge \overline{\langle done[\neg skip]\rangle})$$

126. Apply the *Conjunction introduction* pattern to the state with contract $D4$. This creates two new substates one with contract $D5 \wedge D6$ and the other with contract $D7$.

D5 = $\overline{\langle -done \rangle}$

D6 = $\langle [done[skip]] \rangle$

D7 = $\overline{\langle done[\neg skip] \rangle}$

127. Apply the *Conjunction introduction* pattern to the state with contract $D5 \wedge D6$. This creates two new substates one with contract $D5$ and the other with contract $D6$.

128. Apply the *Disable* pattern to the state with contract $D5$ to introduce transition *done*.

129. Apply the *TotEnable* pattern to the state with contract $D6$ to introduce transition *done*.

130. Apply the *Conjunction elimination* pattern to the state with contract $D5 \wedge D6$.

131. Apply *Redundant hierarchy* to the state with contract $D6$.

132. Apply the *Disable* operator to the state with contract $D7$ to introduce transition *done*.

133. Apply the *Conjunction elimination* pattern to the state with contract $D4$.

134. Apply the *Redundant hierarchy* pattern twice, the inner contract $D3$ is assigned to the outer contract of the start state.

135. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dockquay* transition.

136. Further refine the substate of $D2$ that is the target of the *dockquay* transition with contract D. Apply the *Unfold Always* pattern, contract D becomes $D1 \wedge D2$.

137. Apply the *Conjunction introduction* pattern to state with contract $D1 \wedge D2$. Creates two new substates one with contract $D1$ and the other with contract $D2$.

138. Apply the *If* pattern to the state with contract $D1$ to introduce the transition *done*.

139. Apply the *If* pattern to the state with contract $C2$ to introduce the transition *done*.

140. Apply the *Conjunction elimination* pattern to the state with contract $D1 \wedge D2$.

141. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract D of the parent state to the outer contract of the start state.

142. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dockquay* transition.

143. Apply the *Conjunction elimination* pattern to the state with contract $D1 \wedge D2$.

144. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract D of the parent state to the outer contract of the start state.

145. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dock1* transition.

146. Further refine the target state of the *done* transition with contract D. Apply the *Unfold Always* pattern, contract D becomes $D1 \wedge D2$.

147. Apply the *Conjunction introduction* pattern to state with contract $D1 \wedge D2$. Creates two new substates one with contract $D1$ and the other with contract $D2$.

148. Apply the *If* pattern to the state with contract $D1$ to introduce the transition *docked*.

149. Apply the *If* pattern to the state with contract $C2$ to introduce the transition *docked*.

150. Apply the *Conjunction elimination* pattern to the state with contract $D1 \wedge D2$.

151. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract D of the parent state to the outer contract of the start state.

152. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *done* transition.

153. Apply the *Reroute* pattern to the target of the *docked* transition. This reroutes the *docked* transition to the start state which has the same contract.

154. Apply the *Conjunction elimination* pattern to the state with contract $A \wedge B \wedge C \wedge D$.

155. Apply the *Redundant hierarchy* pattern (see Figure C.32).

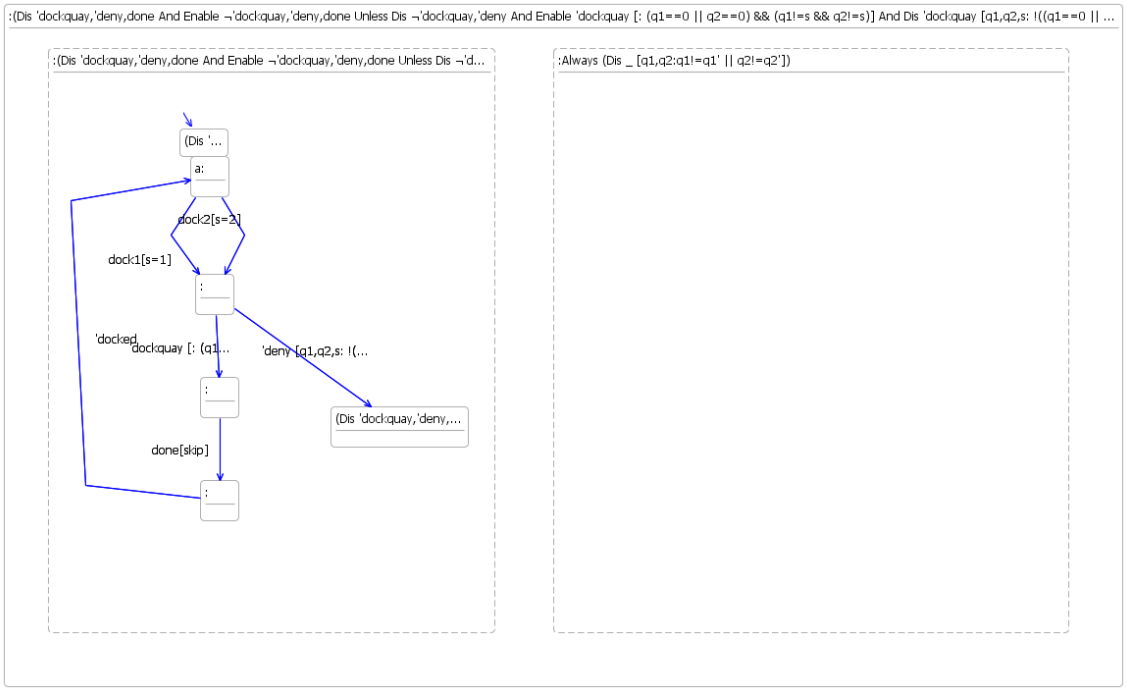

Figure C.32: Conjunction elimination followed by Redundant hierarchy

156. Refine state with contract E. Apply pattern *Unfold Always*, contract E becomes $E1 \wedge E2$.

$\mathbf{E} = \Box\overline{\langle\_[q_1, q_2 : (q_1'! = q_1)||(q_2'! = q_2)]\rangle}$

$\mathbf{E1} = \overline{\langle\_[q_1, q_2 : (q_1'! = q_1)||(q_2'! = q_2)]\rangle}$

$\mathbf{E2} = [\_]\Box\overline{\langle\_[q_1, q_2 : (q_1'! = q_1)||(q_2'! = q_2)]\rangle}$

157. Apply the *Conjunction introduction* pattern to state with contract $E1 \wedge E2$. Creates two new substates one with contract $E1$ and the other with contract $E2$.

158. Apply the *Disable* pattern to the state with contract $E1$ to introduce two new transitions *dock1* and *dock2*.

159. Apply the *Combine states* pattern to combine the substates of $E1$ with contract *True*.

160. Apply the *If* pattern to the state with contract $E2$ to introduce two new transitions *dock1* and *dock2* (see Figure C.33).

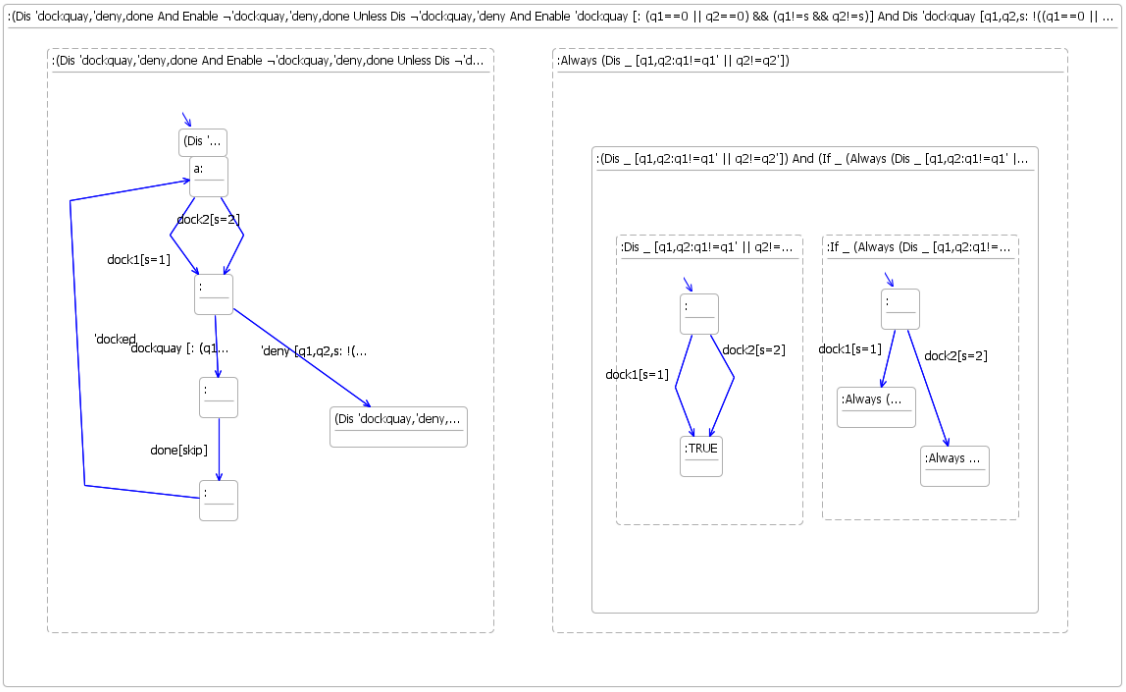

Figure C.33: If pattern applied to the state with contract E2

161. Apply the *Combine states* pattern to combine the substates of $E2$ with contract E.

162. Apply the *Conjunction elimination* pattern to the state with contract $E1 \wedge E2$.

163. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract E of the parent state to the outer contract of the start state.

164. Further refine the target of the *dock1* transition with contract E. Apply pattern *Unfold Always*, contract E becomes $E1 \wedge E2$.

165. Apply the *Conjunction introduction* pattern to state with contract $E1 \wedge E2$. Creates two new substates one with contract $E1$ and the other with contract $E2$.

166. Apply the *Disable* pattern to the state with contract $E1$ to introduce two new transitions *dockquay* and *deny* (see Figure C.34).

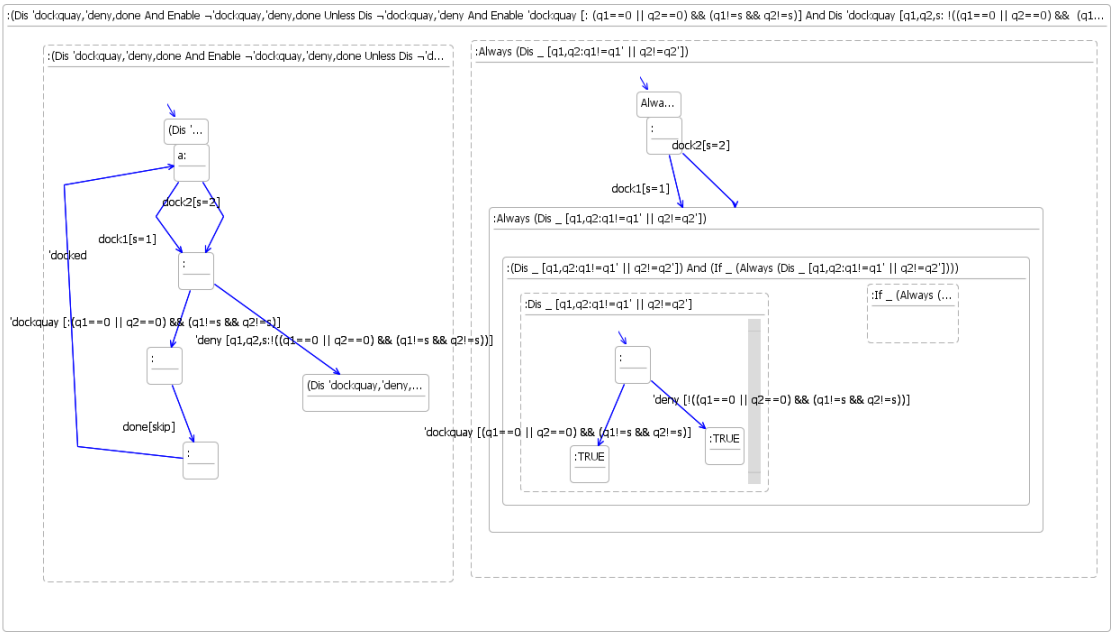

Figure C.34: Disable pattern applied to the state with contract E1

167. Apply the *If* pattern to the state with contract $E2$ to introduce two new transitions *dockquay* and *deny*.

168. Apply the *Conjunction elimination* pattern to the state with contract $E1 \wedge E2$.

169. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract E of the parent state to the outer contract of the start state.

170. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dock1* transition (see Figure C.35).

171. Further refine the target state of the *dockquay* transition with contract E. Apply the *Unfold Always* pattern, contract E becomes $E1 \wedge E2$.

172. Apply the *Conjunction introduction* pattern to state with contract $E1 \wedge E2$. Creates two new substates one with contract $E1$ and the other with contract $E2$.

173. Apply the *Disable* pattern to the state with contract $E1$ to introduce the transition

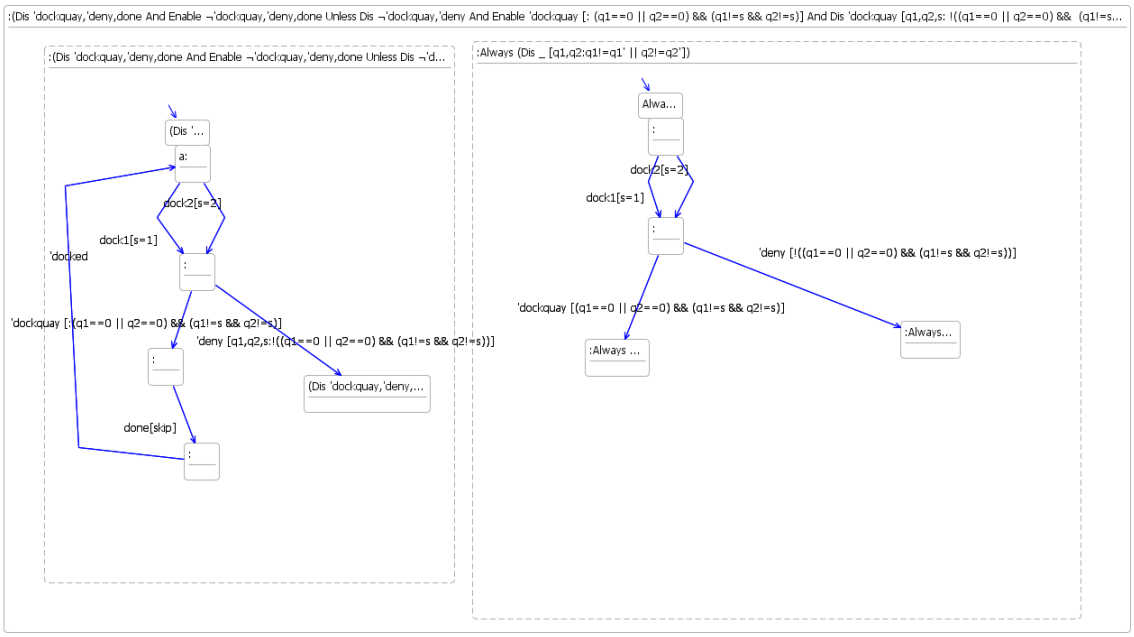

Figure C.35: Move target down and the Redundant hierarchy patterns applied

done.

174. Apply the *If* pattern to the state with contract $E2$ to introduce the transition *done* (see Figure C.36).

175. Apply the *Conjunction elimination* pattern to the state with contract $E1 \wedge E2$.

176. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract E of the parent state to the outer contract of the start state.

177. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *dockquay* transition.

178. Further refine the target state of the *done* transition with contract E. Apply the *Unfold Always* pattern, contract E becomes $E1 \wedge E2$.

179. Apply the *Conjunction introduction* pattern to state with contract $E1 \wedge E2$. Creates two new substates one with contract $E1$ and the other with contract $E2$.

180. Apply the *Disable* pattern to the state with contract $E1$ to introduce the transition *docked*.

181. Apply the *If* pattern to the state with contract $E2$ to introduce the transition *docked* (see Figure C.37).

182. Apply the Conjunction elimination pattern to the state with contract $E1 \wedge E2$.

183. Apply the *Redundant hierarchy* pattern twice. Assigns the inner contract E of the

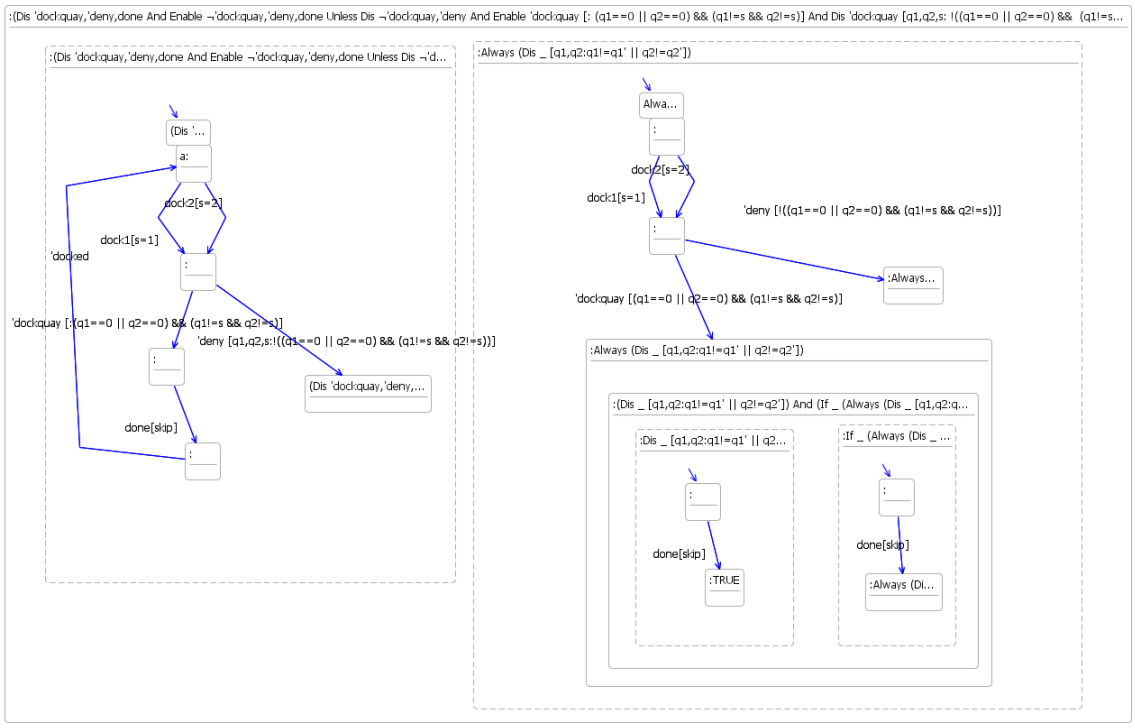

Figure C.36: If pattern applied to introduce *done* transition

parent state to the outer contract of the start state.

184. Apply the *Move target down* and the *Redundant hierarchy* patterns to the target of the *done* transition (see Figure C.38).

185. Apply the *Reroute* pattern to the target of the *docked* transition. This reroutes the *docked* transition to the start state which has the same contract.

186. Apply the *Conjunction elimination* pattern to the state with contract $A \wedge B \wedge C \wedge D \wedge E$ (see Figure C.39).

187. Apply the *Redundant hierarchy* pattern. Assigns the inner contract $A \wedge B \wedge C \wedge D \wedge E$ of the parent state to the outer contract of the start state (see Figure C.40).

188. Apply the *Reroute* pattern to the target of the *deny* transition. This reroutes the *deny* transition to the start state which has the same contract (see Figure C.41).

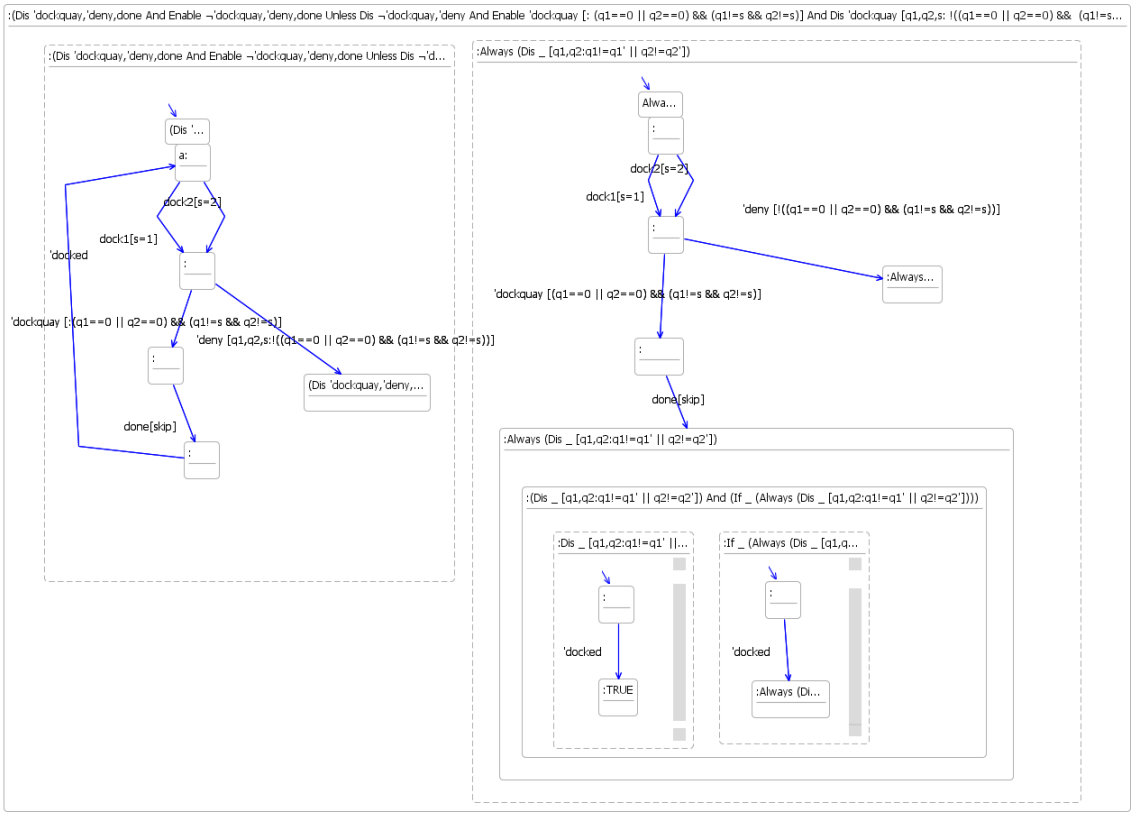

Figure C.37: If pattern applied to introduce *docked* transition

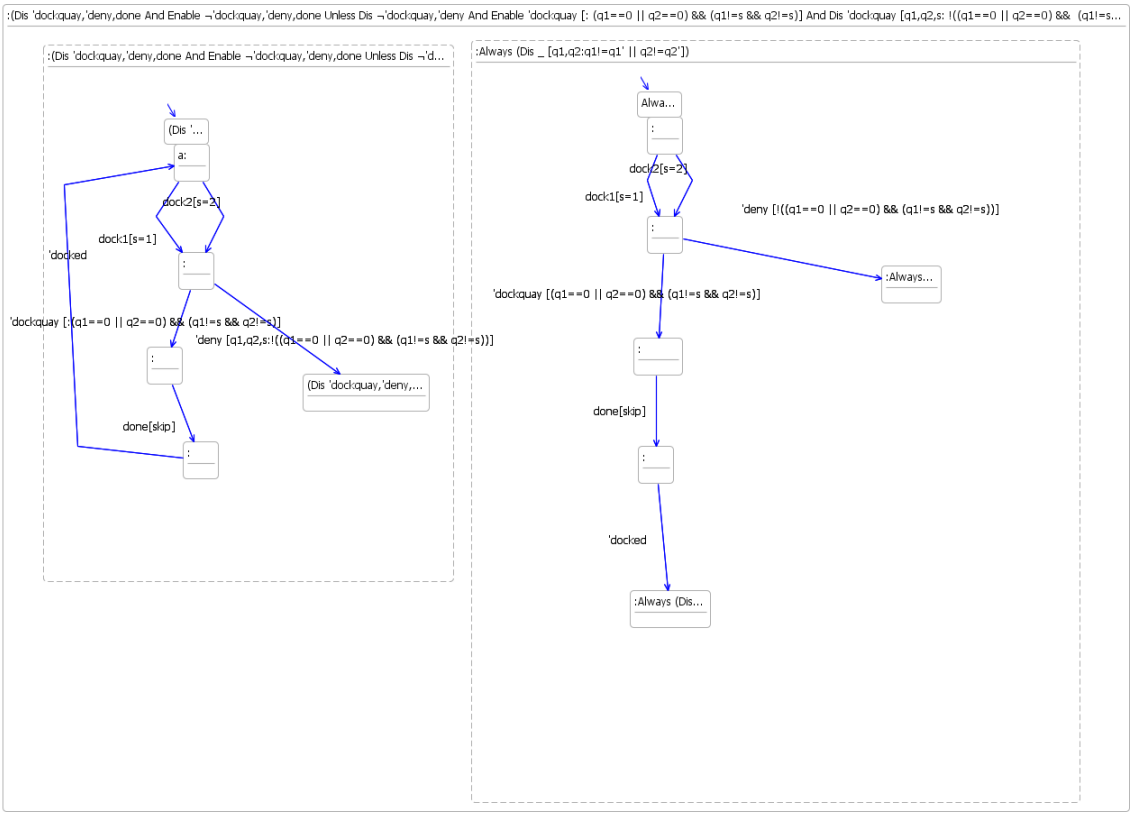

Figure C.38: Move target down and the Redundant hierarchy patterns applied

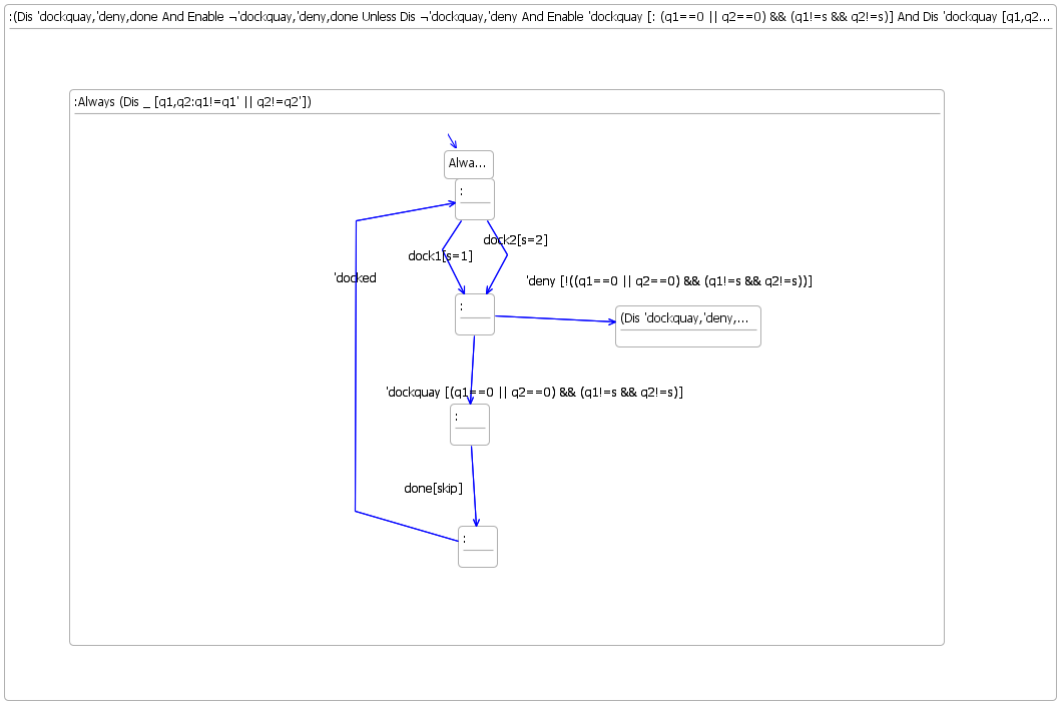

Figure C.39: Conjunction elimination pattern applied

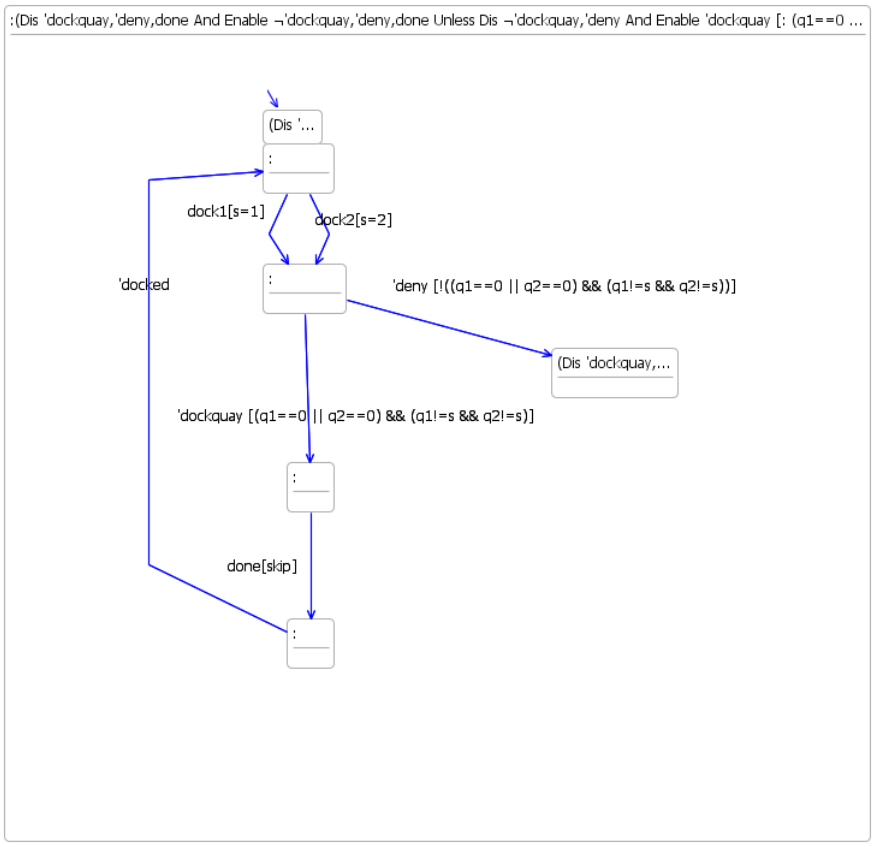

Figure C.40: Redundant hierarchy pattern applied

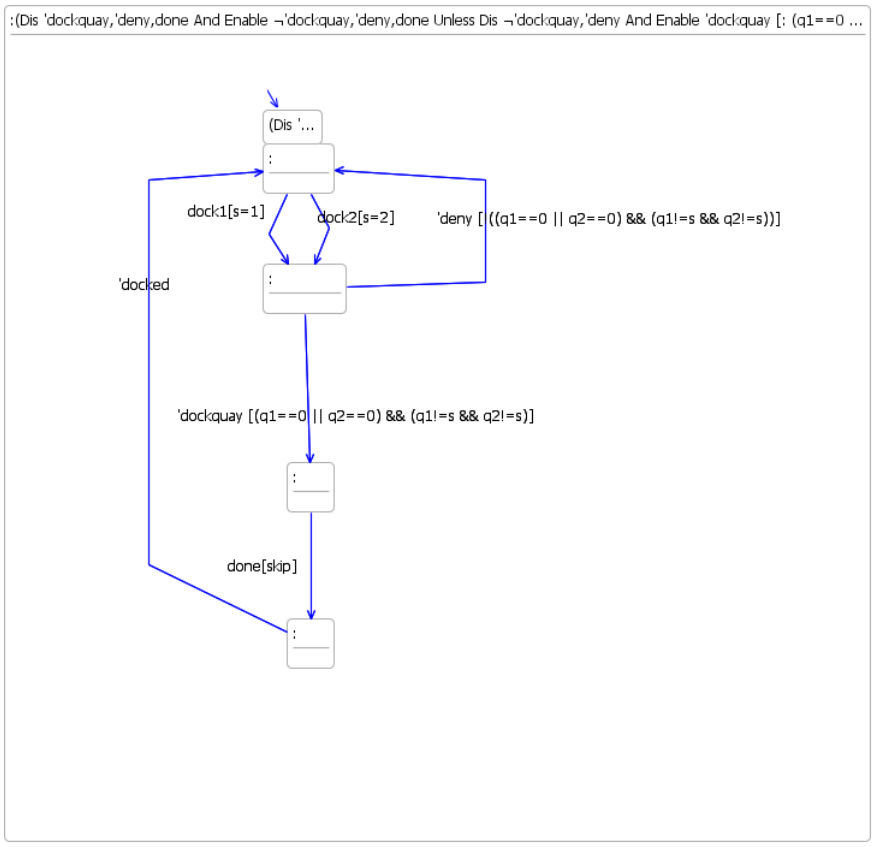

Figure C.41: Reroute pattern applied to the target of the *deny* transition

Appendix D

Additional patterns

This appendix describes patterns that are not in the core set as they are not critical or strictly necessary to achieve a deterministic solution but may be preferable for stylistic reasons offering the engineer different routes to the same solution.

D.1 Patterns for CoSta contracts

Name: *TotEnable combine actions*

Type: Refactoring

Rationale: This pattern combines two disjoined *Totalised enable* operator expressions. The inverse for this transformation *TotEnable split actions* is presented below in this section.

Constraints: The pattern constraint ensures the inner contract is a *Disjunction* operator expression of the form $C_1 \vee C_2$ where C_1 and C_2 are *Totalised enable* operator expressions. The inner contract has the form $\langle [a_1, .., a_n[x : P_1]] \rangle \vee \langle [b_1, .., b_m[x : P_2]] \rangle$. This pattern combines the sets of events together for the *Totalised enable* operator expressions. The pattern checks that the update expressions P_1 and P_2 are syntactically equivalent. The pattern could be generalised for the situation where the update expressions $[x : P_1]$ and $[x : P_2]$ are not syntactically equal. In this instance the side-condition check could be extended to verify that they are semantically equivalent $\vdash P_1 \Leftrightarrow P_2$.

Parameters: The parameter is a state whose inner contract is the disjunction of two *Totalised enable* operator expressions.

Transformation: This pattern replaces the original contract
$\langle [a_1, .., a_n[x : P_1]] \rangle \vee \langle [b_1, .., b_m[x : P_2]] \rangle$ with the new contract $\langle [a_1, .., a_n, b_1, .., b_m[x : P_1]] \rangle$

Diagram:

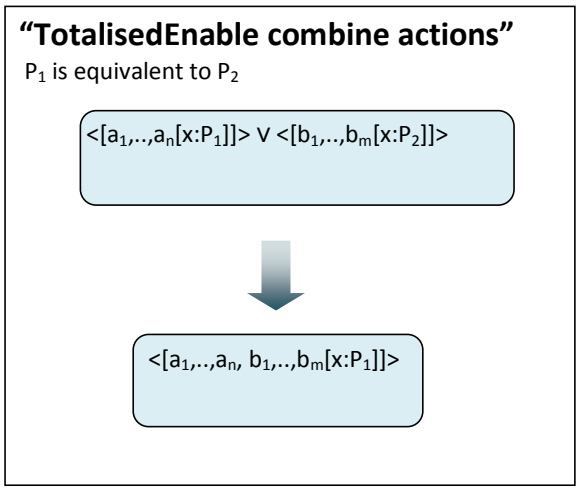

Figure D.1: TotEnable combine actions

Name: *TotEnable split actions*

Type: Refactoring

Rationale: This pattern splits a *Totalised enable* operator expression on its set of actions into the disjunction of two *Totalised enable* operator expressions. The inverse for this transformation *TotEnable combine actions* is presented above in this section.

Constraints: The pattern constraint ensures the contract is a *Totalised enable* operator expression.

Parameters: The parameters are a state whose inner contract is a *Totalised enable* operator expression and its set of actions split into two groups.

Transformation: A *Totalised enable* contract $\langle [a_1, .., a_k, a_{k+1}, .., a_n[x : P]] \rangle$ is split into $\langle [a_1, .., a_k[x : P]] \rangle \vee \langle [a_{k+1}, .., a_n[x : P]] \rangle$.

Diagram:

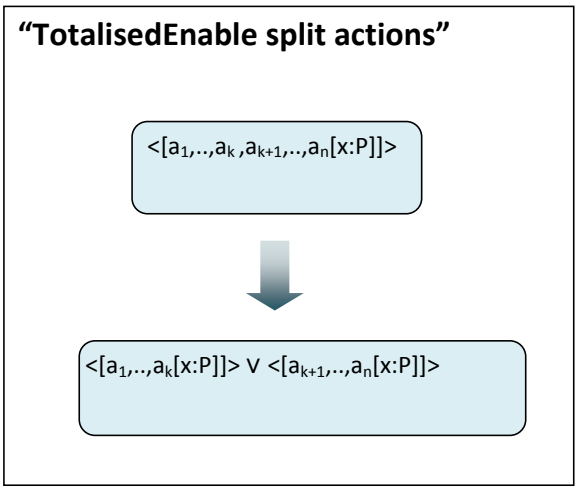

Figure D.2: TotEnable split actions

Name: *If split actions*

Type: Refactoring

Rationale: This pattern splits an *If* operator expression into the conjunction of two *If* operators. The inverse transformation for this pattern *If combine actions* is presented in Chapter 4, section 4.3.1.2.

Constraints: The pattern is applicable to a state with an inner contract that is an *If* operator expression.

Parameters: The parameters are a state whose inner contract is an *If* operator expression and its set of actions split into two groups.

Transformation: An *If* contract $[a_1, .., a_k, a_{k+1}, .., a_n[x : P]]L$ is split into $[a_1, .., a_k[x : P]]L \wedge [a_{k+1}, .., a_n[x : P]]L$.

Diagram:

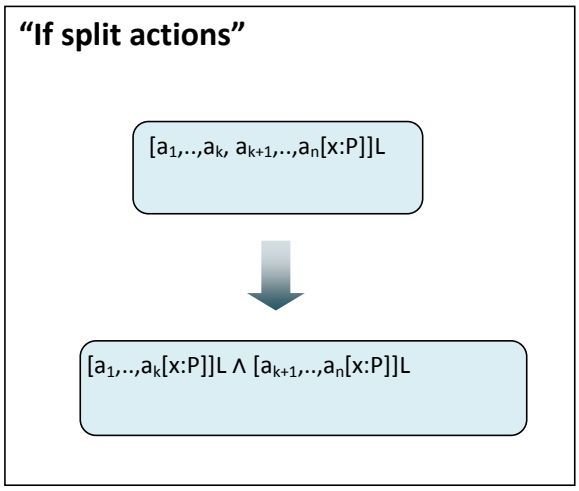

Figure D.3: If split actions

D.2 Patterns for mixed designs

Name: *Split state*

Type: Refactoring

Rationale: This pattern splits a single state into two, this then permits each of the states to be refined differently. This pattern may be needed, for example, when splitting a transition along its guard or update and refining each branch differently. (It is possible to split a transition along its guard or update and then refine each branch differently with the *TotalisedEnable* and *If* patterns). The inverse of this pattern, *Combine states*, is presented in Chapter 4, section 4.3.3.1.

Constraints: The pattern is applicable to a state with an unelaborated inner contract.

Parameter: A basic state.

Transformation: This pattern replaces a state with two new states. The new states are given the same contract as the original state and belong to the same parent state as the original state. The incoming transitions to the original state are split between the new states.

Diagram:

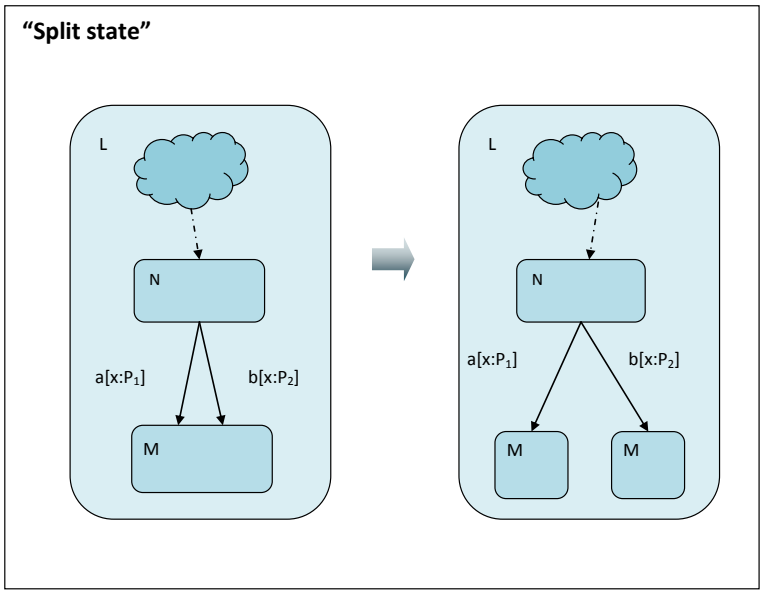

Figure D.4: Split state

Abbreviations

ACP Algebra of Communicating Processes

ACSR Algebra of Communicating Shared Resources

ATL Atlas Transformation Language

CCS Calculus of Communicating Systems

CoSta Contractual State Machines

CSP Communicating Sequential Processes

CWB Edinburgh Concurrency Workbench

ECL Epsilon Comparison Language

EGL Epsilon Generation Language

EMF Eclipse Modelling Framework

EML Epsilon Merging Language

EOL Epsilon Object Language

Epsilon Extensible Platform of Integrated Languages for Model Management

ETL Epsilon Transformation Language

EVL Epsilon Validation Language

EWL Epsilon Wizard Language

FDR Failures-Divergence Refinement

FSM Finite State Machine

GEF Graphical Editor Framework

GME Generic Modeling Environment

GMF Graphical Modelling Framework

HML Hennessey-Milner Logic

HST Heterogeneous Specification Tool

JML Java Modelling Language

LTL Linear Temporal Logic

LTS Labelled Transition System

MDA Model-Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MOF Meta-Object Facility

oAW Open ArchitectureWare

OCL Object Constraint Language

OCLE Object Constraint Language Environment

OMG Object Management Group

PFS Practical Formal Specification

PIM Platform Independent Model

PSM Platform Specific Model

QVT Queries/Views/Transformations

RCP Rich Client Platform

SSA Simulink/Stateflow Analyser

SSM Safe State Machine

STGA Symbolic Transition Graph with Assignment

SVRS Shared Variable Ready Simulation

TCOZ Timed Communicating Object-Z

UML Unified Modelling Language

VHDL VHSIC Hardware Description Language

XMI XML Metadata Interchange

XML eXtensible Markup Language

XP eXtreme Programming

Bibliography

[1] Jean-Raymond Abrial. Toward a Synthesis between Z and B. In Didier Bert, Jonathan Bowen, Steve King, and Marina Waldän, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 629–629. Springer Berlin / Heidelberg, 2003.

[2] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[3] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *ICSE*, pages 761–768, 2006.

[4] Jean-Raymond Abrial. Formal Methods: Theory Becoming Practice. *J. UCS*, 13(5):619–628, 2007.

[5] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[6] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[7] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The B-Method. In *VDM Europe (2)*, pages 398–405, 1991.

[8] Luca Aceto, Anna Ing'olfsd'ottir, Kim G. Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.

[9] Airbus. Airbus380 Passenger Aircraft [online]. [Accessed 02 December 2010] Available at: `http://www.airbus.com/aircraftfamilies/passengeraircraft/a380family/`, 2011.

[10] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, 1977.

[11] IBM Alphaworks. Emfatic Language for EMF Development[online]. [Accessed 08 October 2009] Available at: `http://www.alphaworks.ibm.com/tech/emfatic/`, 2011.

[12] antlr.org. ANTLR Parser Generator[online]. [Accessed 08 October 2009] Available at: `http://www.antlr.org/`, 2011.

[13] Hugo Arboleda, R. Cassallas, and J-C Royer. Comparing two Implementations of an Approach for Managing Variability in Product Line Construction Using the GMF and GME Frameworks. *5th Nordic Workshop on Model Driven Software Engineering, Ronneby, Sweden*, pages 67–82, 2007.

[14] Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A Framework for Evaluating Specification Methods for Reactive Systems Experience Report. *IEEE Trans. Software Eng.*, 22(6):378–389, 1996.

[15] Mohsen Asadi and Raman Ramsin. Mda-based methodologies: An analytical survey. In *ECMDA-FA*, pages 419–431, 2008.

[16] Colin Atkinson and Thomas Kuhne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20:36–41, 2003.

[17] Ralph-Johan Back. A Calculus of Refinements for Program Derivations. *Acta Inf.*, 25(6):593–624, 1988.

[18] Ralph-Johan Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, 1989.

[19] Ralph-Johan Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In *REX Workshop*, pages 67–93, 1989.

[20] Ralph-Johan Back and Kaisa Sere. Stepwise Refinement of Parallel Algorithms. *Sci. Comput. Program.*, 13(1):133–180, 1989.

[21] Ralph-Johan Back and Kaisa Sere. Stepwise Refinement of Action Systems. *Structured Programming*, 12(1):17–30, 1991.

[22] Ralph-Johan Back and Joakim von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In *REX Workshop*, pages 42–66, 1989.

[23] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[24] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335:131–146, May 2005.

[25] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge University Press, New York, NY, USA, 1990.

[26] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha S. Gokhale, and Douglas C. Schmidt. Applying model-driven development to distributed real-time and embedded avionics systems. *IJES*, 2(3/4):142–155, 2006.

[27] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[28] Michael von der Beeck. A Comparison of Statecharts Variants. In *Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[29] J. A. Bergstra. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001.

[30] Jan A. Bergstra and Jan Willem Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109–137, 1984.

[31] G Berry. A hardware implementation of pure ESTEREL. *Sadhana*, 17:95–130, 1992.

[32] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, November 1992.

[33] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32:38–45, July 1999.

[34] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[35] Juan Bicarregui, John S. Fitzgerald, Peter Gorm Larsen, and J. C. P. Woodcock. Industrial Practice in Formal Methods: A Review. In *FM*, pages 810–813, 2009.

[36] Enrico Biermann, Karsten Ehrig, Claudia Ermel, Christian Köhler, and Gabriele Taentzer. The EMF Model Transformation Framework. *AGTIVE*, pages 566–567, 2007.

[37] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST*, 3, 2006.

[38] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *MoDELS*, pages 425–439, 2006.

[39] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 53–67, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Margot Bittner and Florian Kammüller. Translating Fusion/UML to Object-Z. In *MEMOCODE*, pages 49–50, 2003.

[41] Bard Bloom. Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages, PhD thesis, MIT, 1990.

[42] James Blow and Andy Galloway. Generalised Substitution Language and Differentials. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 396–415. Springer, 2002.

[43] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[44] Francis Bordeleau. On the need for "state machine implementation design patterns". Scenarios and state machines: models, algorithms. In *ICSE2002 Workshop*, 2002.

[45] J. P. Bowen and V. Stavridou. Formal Methods And Software Safety. pages 93–98. Pergamon Press, 1992.

[46] Jonathan P. Bowen and Michael G Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12:34–41, 1995.

[47] Julian C. Bradfield. Introduction to Modal and Temporal Mu-Calculi (Abstract). In *CONCUR*, page 98, 2002.

[48] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31:560–599, June 1984.

[49] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[50] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages (Wiley Software Patterns Series)*. John Wiley & Sons, 2007.

[51] Michael J. Butler. Stepwise refinement of communicating systems. *Sci. Comput. Program.*, 27:139–173, September 1996.

[52] Michael J. Butler. csp2B: A Practical Approach to Combining CSP and B. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 490–508. Springer, 1999.

[53] Michael J. Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.

[54] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Tool for Developing Correct Programs By Refinement. In *PROC. BCS 7TH REFINEMENT WORKSHOP*. Springer, 1996.

[55] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement of actions in Circus. *Electr. Notes Theor. Comput. Sci.*, 70(3), 2002.

[56] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A Refinement Strategy for Circus. *Formal Asp. Comput.*, 15(2-3):146–181, 2003.

[57] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement: An overview. In *PSSE*, pages 1–17, 2004.

[58] Ana Cavalcanti and Jim Woodcock. ZRC - A Refinement Calculus for Z. *Formal Asp. Comput.*, 10(3):267–289, 1998.

[59] Kenneth H. W. Chan. *The Application of Formal Methods in Safety Analysis for Safety Critical Software Systems*. PhD thesis, The University of Teesside, 2000.

[60] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2031*, pages 328–342. Springer-Verlag, 2001.

[61] Tony Clark, Andy Evans, Paul Sammut, and James S. Willans. Applied Metamodelling. *www.xactium.com*, 2004.

[62] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[63] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996.

[64] Rance Cleaveland and Gerald Lüttgen. A Logical Process Calculus. *Electr. Notes Theor. Comput. Sci.*, 68(2), 2002.

[65] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15:36–72, January 1993.

[66] Rance Cleaveland and Steve Sims. The Concurrency Workbench of North Carolina, Version 1.11 - User Manual, 1998.

[67] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Picco, and Stefanos Zachariadis. Reconfigurable Component-based Middleware for Networked Embedded Systems. *International Journal of Wireless Information Networks*, 14:149–162, 2007.

[68] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[69] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[70] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *SIGSOFT Softw. Eng. Notes*, 21:179–190, October 1996.

[71] Weijia Deng and Huimin Lin. Extended Symbolic Transition Graphs with Assignment. In *Proceedings of the 29th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '05, pages 227–232, Washington, DC, USA, 2005. IEEE Computer Society.

[72] Bruce Powel Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[73] Desmond D'Souza. Interface Specification, Refinement, and Design with UML/-Catalysis. *JOOP*, 11(3):12–18, 1998.

[74] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[75] Eclipse. Eclipse GMF Tooling [online]. [Accessed 07 June 2008] Available at: `http://www.eclipse.org/modeling/gmp/`, 2011.

[76] Eclipse. Eclipse Graphical Editing Framework [online]. [Accessed 07 June 2008] Available at: `http://www.eclipse.org/gef/`, 2011.

[77] Eclipse. Eclipse Modelling Framework Project [online]. [Accessed 07 June 2008] Available at: http://www.eclipse.org/modeling/emf/, 2011.

[78] Eclipse. The Eclipse Foundation [online]. [Accessed 07 June 2008] Available at: http://www.eclipse.org/, 2011.

[79] Eclipse. Viatra2 [online]. [Accessed 08 October 2009] Available at: http://wiki.eclipse.org/VIATRA2/, 2011.

[80] Eclipse. Xtext 2.1 [online]. [Accessed 08 October 2009] Available at: http://www.eclipse.org/Xtext/, 2011.

[81] E. Allen Emerson. Handbook of theoretical computer science (vol. B). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[82] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33:151–178, January 1986.

[83] Epsilon. EuGENia [online]. [Accessed 24 July 2010] Available at: http://www.eclipse.org/gmt/epsilon/doc/eugenia/, 2011.

[84] Esterel. Esterel Technologies [online]. [Accessed 25 October 2008] Available at: http://www.esterel-technologies.com, 2011.

[85] Eurofighter. Eurofighter Typhoon [online]. [Accessed 02 December 2010] Available at: http://www.eurofighter.com/, 2011.

[86] Neil Evans, Helen Treharne, Régine Laleau, and Marc Frappier. Applying CSP || B to information systems. *Software and System Modeling*, 7(1):85–102, 2008.

[87] Harald Fecher, Michael Huth, Heiko Schmidt, and Jens Schönborn. Refinement Sensitive Formal Semantics of State Machines With Persistent Choice. *Electr. Notes Theor. Comput. Sci.*, 250(1):71–86, 2009.

[88] Harald Fecher, Marcel Kyas, Willem P. de Roever, and Frank S. de Boer. Compositional Operational Semantics of a UML-Kernel-Model Language. *Electr. Notes Theor. Comput. Sci.*, 156(1):79–96, 2006.

[89] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *ICFEM*, pages 52–65, 2005.

[90] Institute for Software Integrated Systems. Generic modelling Environment (GME) [online]. [Accessed 08 October 2009] Available at: `http://www.isis.vanderbilt.edu/Projects/gme/`, 2008.

[91] The Eclipse Foundation. Epsilon Validation Language [online]. [Accessed 09 December 2010] Available at: `http://www.eclipse.org/gmt/epsilon/doc/evl/`, 2011.

[92] Martin Fowler. *Analysis Patterns: Reusable Object Models (Addison-Wesley Series in Object-Oriented Software Engineering)*. Addison-Wesley Longman, Amsterdam, 1996.

[93] Martin Fowler. Refactoring: Improving the Design of Existing Code. In *XP/Agile Universe*, page 256, 2002.

[94] Heinz Frank and Johann Eder. Equivalence Transformations on Statecharts. In *Proc. 12th International Conf. on Software and Knowledge Eng.*, pages 150–158, 2000.

[95] Andy Galloway, J Cockram, and John Mcdermid. Experiences with the application of discrete formal methods to the development of engine control software. *IFAC Workshop on Distributed Computer Control Systems. (15th 1998 Como, Italy).*, 1999.

[96] Andy Galloway, Frantz Iwu, John Mcdermid, and Ian Toyn. Verified Software: Theories, Tools, Experiments. chapter On the Formal Development of Safety-Critical Software, pages 362–373. Springer-Verlag, Berlin, Heidelberg, 2008.

[97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[98] Hassan Gomaa. Designing concurrent, distributed, and real-time applications with UML. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 1059–1060, New York, NY, USA, 2006. ACM.

[99] Gr'egoire Hamon and John Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):447–456, October 2007.

[100] Lishan Harbird, Andy Galloway, and Richard F. Paige. Towards a Model-Based Refinement Process for Contractual State Machines. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ISORCW '10, pages 108–115, Washington, DC, USA, 2010. IEEE Computer Society.

[101] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[102] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.

[103] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5:293–333, October 1996.

[104] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *LICS*, pages 54–64, 1987.

[105] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[106] Reiko Heckel. Tutorial Introduction to Graph Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 458–459. Springer Berlin / Heidelberg, 2008.

[107] Eric C.R. Hehner. *The logic of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[108] Constance L. Heitmeyer. On the Need for Practical Formal Methods. In *FTRTFT*, pages 18–26, 1998.

[109] Constance L. Heitmeyer. On the Utility of Formal Methods in the Development and Certification of Software. In *TPHOLs*, pages 1–2, 2007.

[110] Constance L. Heitmeyer. On the Role of Formal Methods in Software Certification: An Experience Report. *Electr. Notes Theor. Comput. Sci.*, 238(4):3–9, 2009.

[111] Constance L. Heitmeyer and Nancy A. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 120–131, 1994.

[112] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32(1):137–161, 1985.

[113] Michael G. Hinchey and J. P. Bowen. *High-Integrity System Specification and Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[114] Michael G. Hinchey and Stephen A. Jarvis. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1995.

[115] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[116] Sören Holmström. A refinement calculus for specifications in Hennessy-Milner logic with recursion. *Formal Aspects of Computing*, 1:242–272, 1989.

[117] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[118] IBM. IBM Rational Rose Real-Time [online]. [Accessed 04 May 2008] Available at: `http://www-01.ibm.com/software/awdtools/developer/technical/`, 2011.

[119] IBM. Rational Rhapsody 7.6 [online]. [Accessed 14 February 2010] Available at: `http://www.ibm.com/developerworks/rational/library/rhapsody_release-v7.6/release.html/`, 2011.

[120] IBM. Rational Statemate [online]. [Accessed 14 February 2010] Available at: `http://www-01.ibm.com/software/awdtools/statemate/`, 2011.

[121] Alexei Iliasov. Refinement patterns for rapid development of dependable systems. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, EFTS '07, New York, NY, USA, 2007. ACM.

[122] Alexei Iliasov, Alexander Romanovsky, Budi Arief, Linas Laibinis, and Elena Troubitsyna. On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 141–145, Washington, DC, USA, 2007. IEEE Computer Society.

[123] F. Iwu, A. Galloway, I. Toyn, and J. McDermid. Practical Formal Specification for Embedded Control Systems. In *Proceedings of the 11th IFAC Symposium on Information Control Problems in Manufacturing*, Bahia Brazil, 2004.

[124] Frantz Iwu, Andy Galloway, John Mcdermid, and Ian Toyn. Integrating safety and formal analyses using UML and PFS. *Reliability Engineering & System Safety*, 92(2):156–170, February 2007.

[125] Jean-Marc Jezequel, Michel Train, and Christine Mingins. *Design Patterns with Contracts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[126] Cliff B. Jones. The META-Language: A Reference Manual. In *The Vienna Development Method: The Meta-Language*, pages 218–277, 1978.

[127] Cliff B. Jones, Peter W. O'Hearn, and Jim Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, 2006.

[128] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[129] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, pages 128–138, 2005.

[130] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.

[131] Jan Jürjens, Eduardo B. Fernández, Robert B. France, Bernhard Rumpe, and Constance L. Heitmeyer. Critical Systems Development Using Modeling Languages (CSDUML-04): Current Developments and Future Challenges (Report on the Third International Workshop). In *UML Satellite Activities*, pages 76–84, 2004.

[132] Gábor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *J. UCS*, 9(11):1296–1321, 2003.

[133] Ismaïl Khriss, Rudolf K. Keller, and Issam A. Hamid. Pattern-based refinement schemas for design knowledge transfer. *Knowl.-Based Syst.*, 13(6):403–415, 2000.

[134] Anneke Kleppe. MCC: A Model Transformation Environment. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture - Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin / Heidelberg, 2006.

[135] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[136] D. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [online]. [Accessed 14 May 2010] Available at: `http://www.eclipse.org/gmt/epsilon/`, 2011.

[137] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin / Heidelberg, 2006.

[138] Dimitrios S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2009.

[139] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *ECMDA-FA*, pages 128–142, 2006.

[140] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[141] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. Polack. The Epsilon Transformation Language. In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[142] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Eclipse Development Tools for Epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[143] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Novel features in languages of the Epsilon model management platform. In *Proceedings of the 2008 international workshop on Models in software engineering*, MiSE '08, pages 69–73, New York, NY, USA, 2008. ACM.

[144] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A. C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In *Modeling Symposium, Eclipse Summit Europe*, 2007.

[145] Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Rigorous Development of Fault-Tolerant Agent Systems. In Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2006.

[146] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[147] Leslie Lamport. Verification and Specifications of Concurrent Programs. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 347–374, London, UK, 1994. Springer-Verlag.

[148] K. Lano and J. Bicarregui. UML refinement and abstraction transformations. In *Second Workshop on Rigorous Object Orientated Methods: ROOM 2*, 1998.

[149] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[150] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[151] LCI. OCLE 2.0 Object Constraint Language Environment 2.0 [online]. [Accessed 17 March 2008] Available at: `http://lci.cs.ubbcluj.ro/ocle/`, 2003.

[152] Gérard Le Lann. An analysis of the Ariane 5 flight 501 failure - a system engineering perspective. In *Proceedings of the 1997 international conference on Engineering of computer-based systems*, ECBS'97, pages 339–346, Washington, DC, USA, 1997. IEEE Computer Society.

[153] Ji-Hyun Lee, Cheol-Jung Yoo, and Ok-Bae Chang. Component Contract-Based Interface Specification Technique Using Z. *International Journal of Software Engineering and Knowledge Engineering*, 12(4):453–469, 2002.

[154] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26:18–41, July 1993.

[155] Huimin Lin. Symbolic Transition Graph with Assignment. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 50–65, London, UK, 1996. Springer-Verlag.

[156] Formal Systems Europe Ltd. FDR2 User Manual [online]. [Accessed 17 April 2010] Available at: `http://www.fsel.com/documentation/fdr2/html/`, 2010.

[157] Gerald Lüttgen and Walter Vogler. Safe Reasoning with Logic LTS. In *SOFSEM*, pages 376–387, 2009.

[158] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to statecharts semantics. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, SIGSOFT '00/FSE-8, pages 120–129, New York, NY, USA, 2000. ACM.

[159] Brendan Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *THE 20TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE-98*, pages 95–104. IEEE Press, 1997.

[160] Vincent Mah, Benot Combemale, Juan Cadavid, Inria Rennes, and Bretagne Atlantique. Crossing model driven engineering and agility preliminary thought on benefits and challenges.

[161] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[162] Florence Maraninchi and Lionel Morel. Logical-Time Contracts for Reactive Embedded Components. In *EUROMICRO*, pages 48–55. IEEE Computer Society, 2004.

[163] John McDermid, Andy Galloway, Simon Burton, John Clark, Ian Toyn, Nigel Tracey, and Sam Valentine. Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap. *International Conference on Formal Engineering Methods*, 0:76, 1998.

[164] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. In *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, OOIS '02, pages 290–297, London, UK, 2002. Springer-Verlag.

[165] Stephen J. Mellor and Leon Starr. Six Lessons Learned Using MDA. In Nuno Nunes, Bran Selic, Alberto Rodrigues da Silva, and Ambrosio Toval Alvarez, editors, *UML Modeling Languages and Applications*, volume 3297 of *Lecture Notes in Computer Science*, pages 198–202. Springer Berlin / Heidelberg, 2005.

[166] Sun Meng, Zhang Naixiao, and Luis S. Barbosa. On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 164–173, Washington, DC, USA, 2004. IEEE Computer Society.

[167] Tom Mens. On the Use of Graph Transformations for Model Refactoring. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer Berlin / Heidelberg, 2006.

[168] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations: Research Articles. *J. Softw. Maint. Evol.*, 17:247–276, July 2005.

[169] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[170] MetaCase. Domain-Specific Modelling with MetaEdit+ [online]. [Accessed 08 October 2009] Available at: `http://www.metacase.com/`, 2011.

[171] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25:40–51, October 1992.

[172] Bertrand Meyer. Design by Contract: The Eiffel Method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.

[173] Bertrand Meyer. Contract-Driven Development. In Matthew B. Dwyer and Antónia Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, page 11. Springer, 2007.

[174] Microsoft. Microsoft Domain-Specific Language (DSL) Tools [online]. [Accessed 08 October 2009] Available at: `http://www.microsoft.com/download/en/`, 2011.

[175] Granville Miller, Scott Ambler, Steve Cook, Stephen Mellor, Karl Frank, and Jon Kern. Model driven architecture: the realities, a year later. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 138–140, New York, NY, USA, 2004. ACM.

[176] Granville Miller, Andy Evans, Ivar Jacobson, Henrik Jondell, Allan Kennedy, Stephen Mellor, and Dave Thomas. Model driven architecture: how far have we come how far can we go? In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 273–274, New York, NY, USA, 2003. ACM.

[177] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[178] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[179] Benjamin Morandi, Sebastian S. Bauer, and Bertrand Meyer. SCOOP - A Contract-Based Concurrent Object-Oriented Programming Model. In *LASER Summer School*, volume 6029 of *Lecture Notes in Computer Science*, pages 41–90. Springer, 2008.

[180] Carroll C. Morgan. *Programming from specifications.* Prentice Hall International Series in computer science. Prentice Hall, 1990.

[181] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[182] Piotr Nienaltowski, Bertrand Meyer, and Jonathan Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, 21:305–318, 2009.

[183] Barry Norton, Gerald Lüttgen, and Michael Mendler. A Compositional Semantic Theory for Synchronous Component-based Design. In *CONCUR*, pages 453–467, 2003.

[184] Jay F. Nunamaker, Jr., Minder Chen, and Titus D. M. Purdin. Systems development in information systems research. *J. Manage. Inf. Syst.*, 7:89–106, October 1990.

[185] Software Technology Group of Dresden University of Technology. EMFText [online]. [Accessed 07 November 2009] Available at: `http://www.emftext.org/index.php/EMFText/`, 2011.

[186] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Asp. Comput.*, 15(1):28–47, 2003.

[187] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Formal development of industrial-scale systems in Circus. *Innovations in Systems and Software Engineering*, 1:125–146, 2005.

[188] Marcel Oliveira, Frank Zeyda, and Ana Cavalcanti. A tactic language for refinement of state-rich concurrent specifications. *Sci. Comput. Program.*, 76(9):792–833, 2011.

[189] OMG. MOF QVT Final Adopted Specification [online]. [Accessed 22 July 2009] Available at: `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[190] OMG. Object Constraint Language 2.0 Specification [online]. [Accessed 17 March 2008] Available at: http://www.omg.org/technology/documents/formal/ocl.htm, 2006.

[191] OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 23 October 2008] Available at: http://www.omg.org/spec/UML/2.2/, 2007.

[192] OMG. XML Metadata Interchange 2.1.1 Specification [online]. [Accessed 24 June 2009] Available at: http://www.omg.org/technology/documents/formal/xmi.htm, 2007.

[193] OMG. Meta-Object Facility [online]. [Accessed 10 January 2008] Available at: http://www.omg.org/mof, 2008.

[194] OMG. Model Driven Architecture [online]. [Accessed 09 September 2009] Available at: http://www.omg.org/mda/, 2008.

[195] OMG. UML Profile Specifications [online]. [Accessed 22 July 2010] Available at: http://www.omg.org/technology/documents/profile_catalog.htm, 2010.

[196] openArchitectureWare. openArchitectureWare [online]. [Accessed 09 December 2010] Available at: http://www.openarchitectureware.org/, 2008.

[197] Manuel Oriol and Bertrand Meyer, editors. *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*, volume 33 of *Lecture Notes in Business Information Processing*. Springer, 2009.

[198] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19:279–285, May 1976.

[199] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975.

[200] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. Refinement via Consistency Checking in MDA. *Electron. Notes Theor. Comput. Sci.*, 137:151–161, July 2005.

[201] Myung-Hwan Park, Ki-Seok Bang, Jin-Young Choi, and Inhye Kang. Equivalence Checking of Two Statechart Specifications. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, RSP '00, pages 46–, Washington, DC, USA, 2000. IEEE Computer Society.

[202] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[203] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, 1986.

[204] Amir Pnueli. System Specification and Refinement in Temporal Logic. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 1–38, London, UK, 1992. Springer-Verlag.

[205] Amir Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[206] Ivan Porres, Turku Centre, and Computer Science. Model Refactorings as Rule-Based Update Transformations. In *Proceedings of UML 2003 Conference, Springer-Verlag LNCS 2863*, pages 159–174. Springer, 2003.

[207] Ben Potter, Jane Sinclair, and David Till. *An introduction to formal specification and Z.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[208] W. Pree and H. Sikora. Design Patterns for Object-Oriented Software Development. In *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on*, pages 663 –664, May 1997.

[209] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[210] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 42–68, London, UK, 2002. Springer-Verlag.

[211] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[212] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. In *ECMDA-FA*, pages 1–16, 2008.

[213] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[214] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, 1999.

[215] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a Standard ADL? In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, ICECCS '99, pages 43–, Washington, DC, USA, 1999. IEEE Computer Society.

[216] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In *Object-Oriented Behavioral Specifications*, pages 265–286. 1996.

[217] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in Circus. In *FME*, pages 451–470, 2002.

[218] H. Schmidt. *On the role of nondeterminism and refinement in model-driven top-down development of software systems*. PhD thesis, University of Kiel, 2009.

[219] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[220] Steve A. Schneider, Helen Treharne, and Neil Evans. Chunks: Component Verification in CSP||B. In *IFM*, pages 89–108, 2005.

[221] Peter Scholz. A refinement calculus for statecharts. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 285–301. Springer Berlin / Heidelberg, 1998.

[222] Jens Schönborn and Marcel Kyas. Refinement Patterns for Hierarchical UML State Machines. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 5961 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 2009.

[223] Kendall Scott. *UML explained*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[224] Emil Sekerinski and Kaisa Sere. *Program Development by Refinement: Case Studies Using the B Method*. Springer-Verlag, London, UK, 1st edition, 1999.

[225] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[226] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20:42–45, 2003.

[227] Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, FME '97, pages 62–81, London, UK, 1997. Springer-Verlag.

[228] Graeme Smith and John Derrick. Specification, Refinement and Verification of Concurrent Systems-An Integration of Object-Z and CSP. *Form. Methods Syst. Des.*, 18:249–284, May 2001.

[229] Colin Snook and Marina Waldén. Refinement of Statemachines Using Event B Semantics. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin / Heidelberg, 2006.

[230] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010.

[231] SourceForge.net. The Java Modeling Language (JML) [online]. [Accessed 21 January 2010] Available at: `http://www.eecs.ucf.edu/~leavens/JML//index.shtml/`, 2009.

[232] SourceForge.net. Octopus : OCL Tool for Precise UML Specifications [online]. [Accessed 17 March 2008] Available at: `http://sourceforge.net/projects/octopus/`, 2011.

[233] Arcot Sowmya and S. Ramesh. Extending Statecharts with Temporal Logic. *IEEE Trans. Software Eng.*, 24(3):216–231, 1998.

[234] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[235] Susan Stepney, Fiona Polack, and Ian Toyn. A Z Patterns Catalogue: I specification and refactorings, v0.1, 2003.

[236] Susan Stepney, Fiona Polack, and Ian Toyn. An outline pattern language for Z: five illustrations and two tables. In *Proceedings of the 3rd international conference on Formal specification and development in Z and B*, ZB'03, pages 2–19, Berlin, Heidelberg, 2003. Springer-Verlag.

[237] Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to Guide Practical Refactoring: Examples Targetting Promotion in Z. In Didier Bert, Jonathan Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 627–627. Springer Berlin / Heidelberg, 2003.

[238] Colin Stirling. An introduction to modal and temporal logics for CCS. In *Proceedings of the UK/Japan workshop on Concurrency : theory, language, and architecture: theory, language, and architecture*, pages 2–20, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[239] Colin Stirling. Modal and temporal logics for processes. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata: structure versus automata*, pages 149–237, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[240] Canada STL Queen's University. UML 2 Semantics Project [online]. [Accessed 26 August 2009] Available at: `http://www.research.cs.queensu.ca/~stl/internal/uml2/`, 2006.

[241] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[242] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In Martin Gogolla and Cris Kobryn, editors, *UML*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2001.

[243] S.Yacoub and H.Ammar. A Pattern Language of Statecharts. *PLOP '98*, WUCS-98-25 of Techn. Rep. Series, Washington Univ., Dept. Computer Science, 1998.

[244] SysML.org. SysML Open Source Specification Project [online]. [Accessed 02 November 2010] Available at: `http://www.sysml.org/`, 2010.

[245] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 481–488, London, UK, 2000. Springer-Verlag.

[246] Gabriele Taentzer, André Crema, René Schmutzler, and Claudia Ermel. Applications of Graph Transformations with Industrial Relevance. chapter Generating Domain-Specific Model Editors with Complex Editing Commands, pages 98–103. Springer-Verlag, Berlin, Heidelberg, 2008.

[247] L.Schneider G.Taentzer T.Arendt, F.Mantz. Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. In *Joint MoDSE-MCCM 2009 Workshop: Models and Evolution*, 2009.

[248] Inc. The MathWorks. Matlab/Simulink/Stateflow tool suite [online]. [Accessed 11 July 2009] Available at: `http://www.mathworks.com/`, 2010.

[249] Ian Toyn and Andy Galloway. Proving Properties of Stateflow Models Using ISO Standard Z and CADiZ. In *ZB*, pages 104–123, 2005.

[250] Ian Toyn and Andy Galloway. Formal Validation of Hierarchical State Machines against Expectations. *Software Engineering Conference, Australian*, 0:181–190, 2007.

[251] Samuel H. Valentine, Susan Stepney, Ian Toyn, and I Background. A Z Patterns Catalogue II - definitions and laws, v0.1, 2004.

[252] Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 6:139–162, 2007.

[253] Marc van Kempen, Michel Chaudron, Derrick Kourie, and Andrew Boake. Towards proving preservation of behaviour of refactoring of UML models. In *Proceedings of the*

2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, SAICSIT '05, pages 252–259, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[254] Hans Vandierendonck and Tom Mens. Averting the Next Software Crisis. *IEEE Computer*, 44(4):88–90, 2011.

[255] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14:221–227, April 1971.

[256] Jim Woodcock. An Introduction to Refinement in Z. In *VDM Europe (2)*, pages 96–117, 1991.

[257] Jim Woodcock. Using Circus for Safety-critical Applications. *Electr. Notes Theor. Comput. Sci.*, 95:3–22, 2004.

[258] Jim Woodcock and Ana Cavalcanti. A Concurrent Language for Refinement. In *IWFM*, 2001.

[259] Jim Woodcock and Ana Cavalcanti. The Steam Boiler in a Unified Theory of Z and CSP. In *APSEC*, pages 291–298, 2001.

[260] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[261] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[262] Jim Woodcock and Carroll Morgan. Refinement of State-Based Concurrent Systems. In *VDM Europe*, pages 340–351, 1990.

[263] Xactium. Xactium XMF-Mosaic[online]. [Accessed 08 October 2009] Available at: `http://www.xactium.com/`, 2011.

[264] Dezhuang Zhang and Rance Cleaveland. Efficient temporal-logic query checking for presburger systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 24–33, New York, NY, USA, 2005. ACM.

[265] Dezhuang Zhang and Rance Cleaveland. Fast Generic Model-Checking for Data-Based Systems. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, Volume 3731 of Lecture Notes in Computer Science*, pages 83–97. Selected Key Publications, 2005.

[266] Dezhuang Zhang and Rance Cleaveland. Fast On-the-Fly Parametric Real-Time Model Checking. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 157–166, Washington, DC, USA, 2005. IEEE Computer Society.

[267] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 351–357. Springer, 2003.