A Framework for Multi-Notation, Model-Oriented Requirements Analysis

by

Nancy Ann Day

M.Sc., University of British Columbia, 1993

B.Sc.(Hon), University of Western Ontario, 1991

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

October 1998

Abstract

This dissertation addresses the problem of how to bring the benefits of formal analysis to the changing world of multi-notation requirements specifications. We show that direct use of the formal operational semantics for notations in higher-order logic produces an extensible, systematic and rigorous approach to solving this problem. Our approach achieves the desired qualities without requiring theorem proving infrastructure. We concentrate on model-oriented notations that use uninterpreted constants to filter non-essential details. A key contribution is the de-coupling of notation from analysis technique.

We use type checking to regulate combinations of notations. Specifications are represented using embeddings that package the meaning of the notation with its syntax. We demonstrate our approach using combinations of the following three notations: statecharts, decision tables, and higher-order logic.

We introduce a new automatic technique called symbolic functional evaluation (SFE) to evaluate semantic functions outside of a theorem proving environment. SFE produces the meaning of a specification. Direct use of the semantics ensures that the same meaning for a specification is used by all types of analysis. SFE extends the technique of lazy evaluation to handle uninterpreted constants.

We focus on the binary decision diagram (BDD)-based analysis techniques of completeness and consistency checking of tables, simulation, and symbolic model checking. To bridge the gap between higher-order logic and automated analysis techniques, we create a toolkit of common techniques, such as Boolean abstraction.

We show that information contained in the structure of a specification can be used to supplement BDD-based analysis approaches by producing a more precise abstraction of the specification. The partition of a numeric value specified by entries in a row of a table provides information on how to encode numeric values in a BDD.

Our approach is demonstrated by the specification and analysis of two large real-world systems: a separation minima for aircraft and the Aeronautical Telecommunications Network (ATN). In the separation minima example, analysis discovered inconsistencies previously unknown to domain experts. In the ATN example, several errors in the formalisation process were exposed. In both cases, we achieved the benefits of multiple notations for specification without sacrificing automation in analysis.

Contents

List of Tables

List of Figures

Acknowledgements

The completion of this work brings to a close a long journey. Many have helped along the way and I could not have succeeded without them.

I begin by acknowledging the support provided through the FormalWare research project. Raytheon Systems Canada, MacDonald Dettwiler, and the Advanced Systems Institute all contributed to this project. The opportunity to work on real systems had considerable impact on my research. I particularly thank Gerry Pelletier of Raytheon who worked with me on the separation minima example. His interest and patience provided me with a great deal of encouragement.

My supervisor Jeff Joyce has guided and supported me throughout this effort. He has continually challenged me with his ideas of what can be achieved. I thank him for all his extra effort that allowed him to live in the two worlds of academia and industry. Because of his encouragement to tackle real problems, I am left with a great sense of satisfaction from this work. Jeff has taught me a great deal, not only about computer science, but also about consideration for others and making courageous decisions. I am extremely thankful for the kindness of his family (Lorie, Matthew, Nicholas, and Julie) during my years in Vancouver.

I thank my colleagues in the FormalWare research group and the Integrated Systems Design Laboratory, together with others at the Department of Computer Science at the University of British Columbia. I thank my supervisory committee, Paul Gilmore, Mark Greenstreet, Alan Hu, Nancy Leveson, and Gail Murphy, for their helpful comments. I also thank the members of my examining committee, Jo Atlee, Peter Lawrence, Jim Little, and Josi Marti for their encouraging words at my defense. Special thanks go to Kendra Cooper, Brian Edmonds, Karen and Scott Flinn, Andy Martin, Michael McAllister, and Jennifer Shore.

I am fortunate to have a wonderful family who encourage and support me throughout all my endeavours. My parents Richard and Dorothy Day, my sisters Kathleen and Linda, and my brother Bill were my first teachers and they continue, with a gentle touch, to help me work through the ups and downs of life. It is my pleasure to dedicate this work to my parents partly for the value they place on education but mostly for their love and their dedication to their children.

My final word of thanks goes to my best friend, Mark Aagaard. He has an unlimited supply of enthusiasm, faith, and patience that he has shared with me. His presence in my life is a precious blessing.

NANCY DAY

The University of British Columbia

October 1998

For my parents, Richard and Dorothy Day

Chapter 1

Introduction

As the complexity of automated systems grows, the challenges in their development increase. The cost of mistakes can be immeasurable. The first step in developing a system is gaining an understanding of its functionality. Faulk states, "Experience suggests that requirements are the biggest software engineering problem for developers of large, complex systems" [Fau95]. Our ability to assure the accuracy of requirements specifications may soon limit the size and complexity of systems attempted. All stakeholders need to agree that a requirements specification adequately describes a system's intended behaviour. Research on requirements analysis addresses the question of how to present and analyse a requirements specification to achieve this agreement. This dissertation is a contribution to this area of study.

Formal methods have tremendous potential for requirements analysis. A formal notation has a formal syntax and semantics. Each sentence written in the notation has exactly one meaning. Formal notations eliminate ambiguities and make it possible to analyse specifications automatically. Research has already shown that automated analysis techniques, such as model checking, can aid in requirements analysis [AG93, ABB$^+$96, HL96, HJL96, DJJ96].

Many organisations and individuals use different notations for expressing requirements for different aspects of a system. We use the term *language* to describe a set of notations used in a specification. The methodology of structured analysis by DeMarco [DeM79] is an early example of an integrated collection of informal notations. Some formal languages, such as the Requirements State Machine Language (RSML) [LHHR94] and the Software Cost Reduction (SCR) language [Hen80], can be viewed as a collection of notations. Both of these examples were developed from experience with specifying real systems (Traffic Alert and Collision Avoidance System II for RSML and the U.S. Navy's A-7 aircraft for SCR). Different notations within the collection are used for specifying different parts of a system. The Unified Modeling Language (UML) [Rat] is a recent example of a collection of notations for specification in response to people's desire to write multi-notation specifications. A key initiative underway at NASA Jet Propulsion Laboratory (JPL) is the adoption of model-driven design [Dur98]. This initiative involves the integrated use of multiple notations, such as statecharts [Har87], for requirements specification.

Parnas and Madey [PM95] note, "A trap, into which some 'formal methods' groups have fallen is to attempt to define a universal notation for the definition of functions". Furthermore, they predict that new and improved notations will continue to be developed in response to new applications.

The problem is how to bring the benefits of analysis to the changing world of notations and multi-notation requirements specifications. There are a variety of difficulties that are encountered in existing approaches to this problem.

Choosing a fixed set of notations and building analysis tools for this set is a common approach [HLCM92, HBGL95, LBH$^+$95]. With a fixed set of notations the way that the notations can be used in combination is established once and does not change. The disadvantage of this approach is that the fixed combination may slant the requirements engineering effort towards parts of the system most easily expressed in the chosen combination of notations.

Work on requirements specifications often results in a new addition to an existing language (e.g., [HC95, BH97b]) or tailors an existing notation (such as the plethora of statecharts variants [vdB94]). Extensions and variants usually require effort to build new analysis tools or change existing ones.

Even with a fixed set of notations, a different analysis tool is often used independently for each component of the specification. For instance, a statecharts-based tool is used for analysis of the statecharts parts while a different tool is used for analysis of decision tables within a specification. A degree of integration can be achieved through a shared database, as in the approach taken by JPL in their Develop New Products project [Dur98]. But this approach is still a matter of a specification being analysed in parts rather than as a whole. Because some analysis properties rely on information found in parts of the system described in multiple notations, there is a need for techniques that allow analysis to span multiple notations. For example, an invariant may depend on the reachability of states in a state-transition diagram. The transition guards may be expressed by decision tables.

Translation into the input notation of an existing analysis tool is a common approach to analysis of either single language ([AG93, AB96, BH97a]) or multi-notation ([ZJ93, ACD97, PY97]) specifications. Translation bridges the gap between notations developed for their readability and understandability, and notations developed because they can be analysed. There are three disadvantages to the translation approach. First, there is no assurance that the translator correctly implements the semantics of the notation. Second, results are not presented to the specifier in the terms of the original specification. Third, the destination notation for the translation is often geared towards a particular type of analysis. Therefore, a new translator usually has to be written for each new type of analysis. In particular, translation often must include an abstraction step, because the destination notation is unable to capture non-finite values or values of uninterpreted types. The form of abstraction then matches only a particular type of analysis.

The translation approach of Zave and M. Jackson [ZJ93] is notable because their destination notation is one-sorted first-order logic. Abstraction is not required in their technique and multiple kinds of analysis can be applied to the same notation. However, their approach still has the first two disadvantages of translation. They deal with each notation individually rather than offering guidelines for how notations fit together. Consequently, they are faced with the question of whether the requirements specification has a well-defined meaning before analysis can proceed.

A shortcoming of current approaches is that the use of uninterpreted constants is usually not supported. Uninterpreted constants allow the specification to maintain a high level of abstraction for analysis. Section 1.1.2 elaborates on the concept of uninterpreted constants.

In response to these difficulties and shortcomings, we seek an extensible, systematic, and rigorous approach to solve these problems.

An *extensible* approach is one where new notations can be added and used with the existing set of notations without changing the existing notations. An extensible approach should also allow new analysis techniques to be added without changing the notations. Extensibility is a desirable property because new notations are likely to be created and used in combinations. Also, progress is continually being made in the development of new kinds of analysis.

A *systematic* approach is one that has a well-defined methodology for the creation of new combinations of notations in a language. Systematisation makes the process repeatable and accessible to the practitioner. As a form of codification, systematisation is a key step in making software development an engineering discipline [Sha90]. For the problem of analysing multi-notation specifications, systematisation involves creating and enforcing regulations on what combinations of notations have well-defined meanings.

A *rigorous* approach is one with a high degree of assurance that the process is correct. In making a distinction between "formal" and "rigorous", Ghezzi et al. re-mark, “Paradoxically, rigor is an intuitive quality that cannot be defined in a rigorous way” [GJM91]. For the analysis process, rigour means having a high degree of assurance that the analysis has been implemented correctly. It should be possible for an independent expert to scrutinise the process and agree that it is correct. The translation approach often lacks rigour because there is a “leap of faith” in believing a translator adequately matches the semantics of a notation. Rigour can be accomplished by making the process more “obviously correct”.

1.1 Thesis statement

The main thesis of this work is that direct use of formal operational semantics for notations in higher-order logic produces an extensible, systematic, and rigorous approach to analysing multi-notation, model-oriented specifications. Type checking of higher-order logic provides a means of regulating combinations of notations through join points. Furthermore, the information contained in the structure of a specification can be used to supplement binary decision diagram (BDD)-based [Bry86] approaches to analysis by producing a better abstraction of the specification. Our approach achieves the desired qualities without requiring theorem proving infrastructure.

Section 1.1.1 discusses our choice to not use a theorem proving environment for higher-order logic to support our framework. The use of uninterpreted constants is among the more novel aspects of our approach, especially when using BDD-based analysis techniques. Our choice of higher-order logic as the base formalism allows specifications in the framework to include uninterpreted constants. Section 1.1.2 describes uninterpreted constants. Model-oriented notations are described in Section 1.1.3.

1.1.1 Why not use a theorem prover ?

Owre, Rushby, and Shankar [ORS96] demonstrated that the approach of embedding a language in a general-purpose base formalism, such as higher-order logic, makes analysis techniques accessible to a specification. Their work was carried out in the theorem prover PVS [ORS92] for a fixed set of notations. We take their approach further. First, we create a systematic and extensible process for analysis of multi-notation specifications. Second, we show that the approach of embedding notations in higher-order logic does not require theorem proving support. Our approach is novel in linking automated analysis techniques directly with a general-purpose logic without the infrastructure of a theorem prover.

Three observations from our experience with HOL-Voss [JS93] suggest that combining theorem provers with automated techniques is not the optimal approach for automated analysis.

First, the infrastructure of the theorem prover is unnecessary for automated analysis and makes the approach clumsy and intimidating to the novice specifier. For example, rewriting by means of tactic application was used for expansion of definitions in HOL-Voss. This step was different for each specification analysed. We demonstrate that a new automatic technique, called symbolic functional evaluation, is sufficient for this task and requires little user expertise.

Second, theorem provers are verification-based analysis tools. The output of a theorem prover is confirmation of a conjecture. Often a more useful result of requirements analysis is evidence that refutes an interpretation of the requirements. Refutation-based techniques produce a variety of results other than just theorems. For example, when analysing a table for inconsistency, refutation-based techniques can clearly isolate the source of the inconsistency. Consequently, it is easier to interpret the result of a successful refutation attempt than a failed verification attempt. In HOL-Voss, to access the refutation-based results of the Voss decision procedure it was necessary to study low-level information that is passed between the theorem prover and the decision procedure. Access to this information was only possible because the tool existed in two separate parts that exchanged information through internal files.

Third, in HOL-Voss the internal files containing the results of the refutation-based analysis are not expressed in terms of the original specification and therefore are difficult to interpret.

In response to these observations about HOL-Voss, we avoid using a theorem proving environment to access a greater range of analysis techniques. Our framework allows the results of analysis to be presented in terms of the original specification.

We believe that our approach contributes to a "second generation" of formal methods-based analysis. A key characteristic of a second generation approach is the de-coupling of notations from analysis techniques. We choose higher-order logic as a base formalism for its generality and expressibility. It allows specifiers to choose a collection of notations for specification. A range of analysis procedures can be applied to the specification. Our goal is to bring the power of automated analysis to specifiers without sacrificing suitability and expressiveness of notation.

1.1.2 Uninterpreted constants and types

Uninterpreted constants help maintain a high level of abstraction in requirements specifications for analysis[1]. Uninterpreted constants are terms that are declared but not defined, i.e., they have a type but no definition. They represent an unspecified value of the type. An uninterpreted constant represents the same value in every use in a specification. An uninterpreted type is a type for which no details are provided about the members of the type.

Uninterpreted constants can be used to represent elements that have meaning to

[1]The word "constant" used here does not have the same meaning as constants in a programming language. Also, a constant of function type, i.e., "typeA $\rightarrow$ typeB", will often be referred to as a "function".

domain experts but whose definition is irrelevant for analysis of a requirements specification. For example, many air traffic control specifications depend on the "flight level" of an aircraft. The details of how the flight level is determined for an application may be irrelevant for analysis of some aspects of the system. The calculation of the "flight level" can be captured by an uninterpreted constant. Analysis results produced for a specification using uninterpreted constants hold for any interpretation of the uninterpreted constants.

Joyce has called uninterpreted constants, "a modern-day Occam's razor"[2] and points out their value in filtering non-essential details and in improving the readability of the specification [Joy89]. They provide a means for specifiers to guide analysis based on their knowledge of the system rather than knowledge of, for example, theorem proving techniques or binary decision diagram behaviour. User guidance is particularly important when analysis techniques reach capacity limits. A complex part of the specification that is unnecessary for a particular analysis query can be replaced by an uninterpreted constant. We use this technique in the Aeronautical Telecommunications Network example presented in this dissertation (Section 8.2.2).

1.1.3 Model-oriented notations

M. Jackson considers a "model" to be an analogue to the real entity [Jac95]. In this work, we are interested primarily in abstract models of computer-based systems. Our abstract models consist of a relationship between elements of a (possibly infinite) set of "configurations"[3]. A configuration is a mapping from a set of *names* to a set of values. Mathematically, the relationship between configurations may be viewed simply as a set of pairs (A, B) where A and B are both configurations. Although we can regard this

[2]The idea that the simplest approach should be used; essentially not adding details when they are unnecessary. (William of Occam, 1280? - 1347?)

[3]This concept is often called a state. Because the word "state" is overloaded when dealing with notations we adopt the word "configuration" for this concept. Similarly, what is commonly called the "state space" will be referred to as the "configuration space". A configuration has also been called a status [i-L91] and an interpretation [AB96].

mathematical relationship in purely static terms as a set of pairs, it is meant to represent the dynamic behaviour of the system. In particular, this relationship is the "next configuration" relationship between a configuration and its immediate successor(s). This next configuration relation is the transition relation of an automaton. We use the term "model-oriented specifications" to describe any kind of specification that denotes a relationship on a collection of configurations. We extend the meaning of the term model-oriented notations to include notations that describe parts of model-oriented specifications, such as decision tables. Many existing languages, such as RSML and SCR, consist of a fixed collection of model-oriented notations.

We focus on the integration of model-oriented notations making up one model. For example, for the statecharts notation [Har87], we consider an extensible set of notations that can be used for the triggers and actions of transitions. Our approach extends to multiple models in a specification by relying on the standard technique of conjunction ([AL93, ZJ93]). By considering a more limited range of notations than Zave and Jackson [ZJ93], we are able to apply automated analysis techniques and to produce guidelines for how notations fit together.

Our concentration on model-oriented notations is motivated by two factors. First, there have been successful efforts to use these notations for specifying real systems [Hen80, LHHR94, CGR95]. Second, there are automated analysis techniques developed for model-oriented specifications [CES86, HL96, HJL96, DJJ96]. These techniques usually rely on a model describing a finite set of reachable configurations. System requirements often do not describe a finite set of configurations. Our work is a contribution to "lifting up" these techniques to apply them to specifications that may contain values of infinite or uninterpreted types.

1.2 Approach: our framework

This dissertation describes a framework for

- combining multiple notational styles in one specification, possibly including uninterpreted constants, and
- applying multiple kinds of analysis to specifications.

The term "framework" is used because it provides a skeleton on which other (possibly not yet invented) notations and analysis techniques can be brought together.

This section is an introduction to our framework, which creates an extensible, systematic, and rigorous approach to analysing multi-notation specifications. The development of our approach has required careful choices in foundations and combinations of techniques. Our work draws on the areas of higher-order logic (Chapters 3, 4, and 5), requirements engineering (Chapters 4, 7, and 8), formal definitions of the semantics of notations (Chapters 4 and 5), implementation of functional programming languages (Chapters 3 and 6), and BDD-based analysis methods (Chapter 7). The reader is expected to have a working knowledge of each of these areas. We introduce our framework in a progression of ten sub-problems encountered in its development. These problems are technical issues that arise in trying to solve the previously mentioned problems with current approaches.

1.2.1 Integrating combinations of notations

The first problem is how to systematise the process of fitting together multiple notations in a specification. Current approaches either 1) use a fixed set of notations or 2) place no limits on the ways in which notations can be combined. The second approach encounters the difficulty of determining whether the combination of notations has a well-defined meaning before analysis can proceed. Our solution to this problem is to determine categories of notations and to let individual notations indicate well-defined ways that they interface to notations in other categories.

Many model-oriented languages consist of elements that fall into one of four categories: models, events, actions, and expressions. For example, a statechart falls into the model category. A decision table is a form of expression. We call these categories *semantic categories*.

A notation that belongs to one category can rely on an element in another category to produce a specification. For example, the transitions of a statechart are labelled with events and actions. The statechart notation can be described with two "slots" for these parts of the notation. These slots may be filled with any event notation for the first slot, and any action notation for the second slot. Drawing from terminology in subject-oriented programming, we call these slots join points[4]. A *join point* is a point at which one notation can be combined with another notation. Another example of a join point is the guard on a transition in RSML. In the fixed set of notations used in RSML, the guard is described by an AND/OR table. In our framework, the guard field is a join point to any notation in the expression category. Join points are labelled by a semantic category. The label indicates that a notation from that category can be used at this point. The semantic categories provide a systematic basis to consider new combinations of notations. Any notation resulting in an expression can be plugged into this join point as a guard. Instantiating all join points in a set of notations creates a language.

Semantic categories and join points are discussed in Section 4.2.

[4]Personal communication with G. Murphy, October 1996. Murphy reported that the term "join point" was first used by W. Harrison and H. Ossher at an aspect-oriented programming workshop in October 1996. They use the term to describe constructs in source code that could be used as the basis for integrating subjects in subject-oriented programming [HO93].

1.2.2 Representation of notations

The second problem is how to represent notations in a common format such that analysis can be carried out on a specification. The translation approach requires multiple (possibly integrated) editors or parsers to create each part of the specification in a different notation. Our solution is to represent the notations textually in higher-order logic. This solution means that one parser and one type checker support all notations and require no extensions for new notations. This approach limits us to notations that use the same type checking rules as higher-order logic. This restriction is not severe as witnessed by the many examples of notations embedded in higher-order logic (e.g., a subset of the VHDL hardware description language [Tas93], Z [BG94], the process calculus value-passing CCS [Nes93], and a subset of the programming language SML [VG93]). While not necessary for the framework, special-purpose editors or renderers could be used and adapted to produce or to display the textual representation of the notation. In one example carried out for this work [DJP97a, DJP97b], we took this approach by creating a tool to produce HTML tables to display a tabular specification.

This section provides a brief introduction to how notations are represented in higher-order logic. A notation is a collection of keywords that can be combined using well-defined rules. The keywords of the notation are represented as functions. These functions take as arguments the arguments to the keyword. For example, a transition between states is a tuple with fields for the source state, event, action, and destination state. In this case, the "comma" operator is a function creating the transition construct. A second example is the assignment action. In our framework, the keyword is `Asn` and the arguments to the keyword function are the left-hand side and the right-hand side of the assignment action. We call the representation of a notation in higher-order logic a *notational style*[5]. In this dissertation, we demonstrate notational styles based on statecharts,

[5]This terminology is borrowed from Zave and M. Jackson [ZJ93]. They do not define the term but rather refer to specifiers choosing different notational styles for different parts of a specification.

decision tables, the event and action notations of statecharts, and others. Higher-order logic itself can also be used as a notation.

Notational styles are discussed in Chapter 4.

1.2.3 Regulating combinations of notations

The next problem is how to ensure that a combination of notations used in a specification has a well-defined meaning. We cannot ensure that a specification is sensible, but regulation helps avoid errors resulting from inappropriate combinations of notations, such as using a model as the trigger of a transition. Pezzè and Young [PY97] deal with this problem by requiring special additions to indicate how particular notations can be used together. Our solution is to use the type system of higher-order logic to regulate combinations based on the semantic categories. This regulation can then be enforced by the type checking algorithm of Milner [Mil78]. By using the notational styles in higher-order logic, regulation requires no additions for particular combinations.

Each semantic category is associated with a type in higher-order logic. A keyword in a notational style indicates a join point by having a parameter with the type of a category. For example, one event notation consists of an `EvCond` keyword to describe the event of another event occurring when a condition is true. The type of this keyword is `event -> exp -> event`, where the `->` indicates a function. The meaning of this type expression is that `EvCond` takes an event and an expression and returns an event. Its first argument has the type of the event category and is therefore a join point to any event notation. Its second argument has the type of the expression category and is therefore a join point to any expression notation, such as an AND/OR table. This keyword returns an event and is a member of the event category.

To add a new notation to a semantic category, keywords of the new notation must return elements with the type of that category. The new notation can then be used at all relevant join points of existing notations.

Type checking and representation of notational styles in higher-order logic creates a systematic and extensible way to describe specifications in combinations of notations. The regulation provided by the type system of higher-order logic is described in Sections 4.2 and 5.3.

1.2.4 Expressing dynamic behaviour

The next problem is how to capture dynamic behaviour in higher-order logic. Model-oriented notations describe dynamic behaviour. The values of the names (data-items) can change between different configurations. Representation of names in higher-order logic requires a means of expressing the value of the name in different configurations. We introduce the concept of a configuration formally as a type. Names are functions that take a configuration and return the value of the name in that configuration. We call this technique *configuration lifting*. Sections 4.4 and 4.6.1 discusses this issue further. Section 6.9.3 discusses the implications of this approach for analysis.

1.2.5 Determining the meaning of a notation

The fifth problem is how to state the semantics for each notation. Analysis is based on the meaning of a specification. There are a variety of ways of writing the semantics of a notation, such as denotational, operational, and axiomatic semantics. A next configuration relation is required by many configuration space analysis techniques such as model checking and simulation. Operational semantics define the meaning of a model-oriented notation as a next configuration relation. Therefore, we use operational semantics to describe the meaning of a notation in our framework. Formal operational semantics can be written in a machine-readable form in higher-order logic just as the specifications are written in notational styles of higher-order logic.

Chapter 5 presents the formal operational semantics for our example notations.

1.2.6 Associating meaning with representation

The sixth problem is how to link the specification with the semantics for the notations to determine the specification's meaning. The notational styles are embedded in higher-order logic. Boulton et al. [BGG$^+$92] describe a classification of embeddings in higher-order logic as either shallow or deep. In a deep embedding, the concrete syntax is represented in the logic as a type. In a shallow embedding the syntax is represented as functions in the logic, which are defined by the meaning of the notation. For our framework to have the property of extensibility, we want the meaning to be packaged with the notation.

A shallow embedding is often suitable for the packaging required by our framework. In a shallow embedding, the functions representing the keywords are defined by the meaning of the keyword. For example, the `Asn` keyword, which takes two parameters that are the left-hand side and right-hand side of an assignment, is defined to equate the value of the left-hand side in the next configuration with the value of the right-hand side in the previous configuration. The use of a notation can be viewed as a "lego-like" building block for creating a specification. A block is a package of keywords and the definition of the meaning of the keywords. Join points are the interfaces between blocks. Packaging creates an extensible way to write a multi-notation specification with a well-defined meaning.

For notations whose meaning cannot be represented compositionally (notably statecharts) a deep embedding is used. If the meaning of a notation cannot be expressed compositionally based on its syntax, then a shallow embedding must be an expanded version of the original notation. The representation in higher-order logic then loses its resemblance to the original specification. A deep embedding maintains the resemblance to the original notation. To retain the packaging necessary for our framework using a deep embedding, we include keywords in the notation that take as arguments the representation of the syntax of the notation. For example, the structure of a statechart specification is represented with the type constructors `AND_STATE`, `OR_STATE`, and `BASIC_STATE`. As a notation in our framework, the use of a statechart consists of the keyword `Sc` applied to a statechart structure. The join points to other notations are still regulated by type consistency.

We call our approach *packaged embeddings*. Chapter 5 describes the packaging of notational styles with their semantics.

1.2.7 Determining the meaning of a specification

The seventh problem is how to ensure that the analysis procedures correctly interpret the meaning of the specification. We also want to ensure that all analysis procedures understand the same meaning for the specification. There is a danger that implementations of analysis procedures will not correctly interpret the meaning of the specification. This danger is multiplied when working with multiple notations and new notations. Furthermore, we want to avoid the extra work of modifying the analysis procedure when new notations are added.

Translation is one approach to this problem. A translator transforms the specification into a notation understood by the analysis procedure. Translation is notation-specific. It also often has the disadvantage of lack of rigour. A second approach is to use rewriting in a theorem prover to expand the semantic definitions. Rewriting relies on a unification step, which searches for matches between expressions and theorems of equality. This step is not necessary to expand the definitions of the semantic functions. To avoid the need for theorem proving infrastructure, we developed a technique specifically for definitional expansion.

We introduce a new technique called *symbolic functional evaluation* (SFE) to use the semantic definitions directly to expand the meaning of the specification. Analysis is then applied to the meaning of the specification in higher-order logic. Symbolic functional evaluation removes the need for theorem proving infrastructure without loss of rigour.

The semantics are definitions in higher-order logic. Function definitions in higher-order logic are represented in the lambda calculus. Definitional expansion replaces the use of a definition with the body of the definition. Evaluation involves definitional expansion and beta-reduction of lambda abstractions. The lambda calculus is the basis for functional programming languages. Consequently there is a large body of knowledge on how to carry out evaluation efficiently and compactly, including techniques such as "lazy evaluation" and "evaluation in place". We draw on this knowledge and provide extensions to handle uninterpreted constants in a specification. The name for our technique is chosen because it carries out functional evaluation in the programming language sense, but the results contain "symbols" (the uninterpreted constants).

The presence of uninterpreted constants means that a specification might not evaluate to a concrete term, as in functional programs. Uninterpreted constants in a specification mean that the result of evaluation can contain the application of uninterpreted constants to expressions. Many automated analysis techniques can only be applied to a finite specification. An abstraction of a potentially infinite specification must be created to apply these techniques. Consequently, further evaluation of arguments to the uninterpreted constants exposes details about the specification that will be lost in the abstraction process. For efficiency, we define several levels of evaluation that are useful for automated analysis. These levels are based on how much evaluation is carried out on the arguments of uninterpreted functions. These levels are implemented as different modes, which can be chosen by the specifier.

SFE ensures that the meaning of the specification understood by the analysis tool is the same as the meaning defined by its semantics. Moreover, all analysis techniques assume the same meaning for the specification. Also, the burden of implementing a new analysis function is lessened since it need not deal with the extensive range of notational styles of our framework.

Chapter 6 describes symbolic functional evaluation.

1.2.8 Abstraction and automated analysis procedures

The eighth problem is how to attach the analysis procedures to our framework. Most previous approaches to requirements analysis have either used theorem provers to handle the non-finite elements of a specification or else restricted specifications to be in notations describing a finite set of configurations. An expert is required either to use a theorem prover or determine a finite abstraction of a specification for analysis.

In our framework, analysis procedures interface with the expanded meaning of a specification in higher-order logic produced by SFE. Many automated techniques can only be applied to a specification that describes a finite number of configurations. To apply these techniques in our framework, we have included a toolkit of methods used by multiple BDD-based analysis techniques. The toolkit includes an abstraction technique for creating a specification with a finite number of configurations from one with a possibly infinite configuration space. This abstraction is *conservative* in that it allows the system to have more behaviours than the original specification. The use of a conservative abstraction can produce results that describe behaviours not found in the original specification. An abstraction that has fewer behaviours than the original specification is called *liberal*.

The abstraction produced by this method may not be sufficient to prove a particular property, but it does provide a simple means for practitioners to begin applying analysis to a specification. Section 1.2.10 describes a way that we have improved upon existing Boolean abstraction mechanisms. Also, the addition of small amounts of "domain knowledge", either specialised to the system or common knowledge such as theorems about numbers, can help analysis produce more accurate results. We call these additions *environmental constraints*.

Our toolkit of techniques common to multiple BDD-based analysis approaches lessens the burden of adding new techniques. A toolkit of re-usable techniques adds to the rigour of our approach because it reduces what can go wrong in the implementation of a new technique.

Forms of simulation, model checking, and completeness and consistency checking have been implemented to demonstrate our framework. Simulation and model checking can be applied to any specification that is a model in our framework. Completeness and consistency checking are specific to tabular specifications.

If a specification is "executable", it can be simulated. In simulation, the specifier plays the role of the environment by providing inputs and by stepping through the resulting behaviour of the system. Simulation is an effective means of investigating whether a specification matches the specifier's intent.

More exhaustive techniques than simulation for analysing the specification can be categorised as application-dependent or application-independent. An application-dependent analysis technique requires the input of some system specific information. For example, model checking is an automatic, application-dependent technique that tests whether a specification satisfies a property in temporal logic [CES86]. Application-independent properties can be formulated without reference to particulars of a specification.

Two examples of application-independent techniques are completeness and consistency checks. Jaffe et al. define completeness of a requirements specification in an engineering sense as including sufficient detail to "distinguish the behaviour of the desired software from that of any other, undesired program that might be designed" [JLHM91]. Techniques called domain completeness [HL96] or coverage [HJL96] have been developed for examining the completeness of outputs specified for possible combinations of inputs in tabular specifications. A consistent specification is one where the requirements do not conflict. Parnas points out that these types of checks are very tedious for reviewers to carry out manually [Par93b].

The C programming language is used to implement our toolkit and analysis procedures. Analysis procedures used in the framework are described in Chapter 7.

1.2.9 Reporting analysis results

Because analysis procedures often require an abstraction of the specification, the results of analysis are produced in terms of the abstraction. The ninth problem is how to return results in terms of the original specification. In a translation approach, the tool returning the results does not have access to the original specification and therefore can only produce results in terms of the abstraction. For example, many BDD-based analysis tools return results in terms of Boolean variables. The translation process may have abstracted a Boolean expression to be represented by a Boolean variable.

We address this problem in two ways in our approach. First, the abstraction carried out for analysis is reversed before returning results. Second, the symbolic functional evaluation phase maintains the unexpanded version of expressions. This ability allows results to be presented at an abstract level that matches the original input. We call this *restructuring* of the results. For example, the output of completeness checking of a table is presented with the entries in the rows in their unevaluated forms as in the original specification.

Section 7.3.3 discusses reversing the abstraction process. Section 6.5 describes restructuring. Section 8.1.2 provides an example of its use in returning analysis results.

1.2.10 Exploiting structure

The last problem is how to exploit structure to aid in analysis. Automated analysis techniques usually ignore structure and only work with the meaning of a specification. We use the term *structure* to mean the arrangement of items in the specification chosen by an individual. Different notations provide different structures. For example, the rows and columns of a table comprise a structure. We show that this structure can be exploited in the analysis of tabular specifications to produce a more precise abstraction than strictly BDD-based techniques and to reduce the size of the configuration space. Because our framework provides access to the parse tree representation of a specification, the loss of distinction between the syntax and semantics in a shallow embedding does not affect our ability to exploit structure in analysis.

A specifier's arrangement of items in a row may contain a partition of a set of numeric values. We show how the arrangement can be checked to determine if it is a non-overlapping partition for a numeric value. If so, the partition is encoded in a BDD and the results produced by analysis are less conservative than treating each Boolean constraint as an independent Boolean variable. If the partition does not include all ranges, the extra ranges are added to the encoding. Furthermore, the size of the configuration space is reduced.

The techniques of our framework, such as symbolic functional evaluation, limit the use of the framework to analysis of model-oriented notations of fixed structure. For example, this limitation means that there must be a fixed number of rows in a table, or a fixed number of states in a statechart. Fixed structure does not imply a finite configuration space.

The use of structure in analysis is discussed in Section 7.3.1.

1.3 Examples

Three examples are used to demonstrate our claims. The first example is a specification of a heating system, loosely based on an example found in Booch [Boo91]. It is used throughout the dissertation to illustrate aspects of the work. This example uses a combination of notational styles, namely higher-order logic, tables and statecharts. All types of analysis currently implemented in our framework except symmetry checking[6] are carried out on the heating system specification.

The second example is a specification of the separation minima for aircraft in the

[6]Symmetry checking is a new analysis technique that we introduce for determining if a tabular specification is symmetric in its parameters. It will be described in Section 7.5.3.

North Atlantic Region. It uses a combination of the tabular style of specification and higher-order logic. It makes use of many uninterpreted types and constants. This specification is analysed for the application-independent properties of completeness, consistency, and symmetry. The ability to restructure results of this analysis to present them to the specifier in the form of the specification is particularly valuable. The use of structure found in rows of a table reduces the number of errors in the results due to the choice of abstraction. Inconsistencies are found in the original informal specification through our analysis.

The third example is the Aeronautical Telecommunications Network (ATN). The formal specification uses higher-order logic and statecharts and was created by a team of specifiers. The ATN is a global telecommunications network for air traffic control systems. It will allow aircraft and ground stations to exchange data. This specification is analysed in our framework using simulation and model checking. It makes extensive use of the parameterisation achieved through the combination of higher-order logic and statecharts. The size of this formal specification (43 pages) demonstrates the scalability of the symbolic functional evaluation technique. The analysis discovers several errors in the formalisation process and uncovered behaviour unknown to several of the specifiers and to the author of this dissertation.

1.4 Validation

We claim that the direct use of formal operational semantics of notations in higher-order logic produces an extensible, systematic, and rigorous approach to analysing multi-notation specifications that improves upon existing approaches. Furthermore, we claim that information contained in the structure of a specification can be used to supplement BDD-based approaches to analysis by producing a better abstraction of the specification. In this section we describe how these claims have been validated in this dissertation.

Our framework has been implemented and used to analyse the three examples described in the previous section. The examples each use multiple notational styles in their specification and multiple kinds of analysis are carried out on the specifications. For example, the heating system specification uses statecharts, decision tables, and higher-order logic. Model checking, simulation, completeness and consistency checking are all carried out on the heating system specification in the framework (Chapter 7).

The example notations of statecharts and decision tables are chosen because of the prevalence of well-developed requirements specification techniques that rely on state-transition and tabular notations (e.g., Parnas tables [Par92], the Software Cost Reduction (SCR) method [Hen80], the Requirements State Machine Language (RSML) [LHHR94] and the Object Modeling Technique (OMT) [R$^+$91]). Higher-order logic was chosen as a notation for specification because it includes uninterpreted constants.

Our framework is extensible in its ability to create new combinations of notations without changes to the existing set of notations. Extensibility is accomplished by packaging the notation with its meaning. Extensibility is demonstrated through the use of combinations of notations not previously seen in specifications. Also, new notations are introduced and used in combinations without requiring changes to the existing set of notations. For example, a function table is used to describe the action of a statechart transition in the heating system specification (Section 4.10). Neither notation required any changes to use them in combination. A second example demonstrated in the separation minima specification is that tables can be used as entries or results to other tables interchangeably with other forms of expressions (Section 8.1.1). This integration requires no changes to the tabular notation. A third example is that the combination of higher-order logic and statecharts allows the statecharts to be parameterised with no changes to the statecharts notation. This combination is demonstrated in the ATN example (Section 8.2.1). A fourth example is that the ATN specification requires directed communication between components. The original event and action notations for statecharts include only broadcast communication. We extend the set of notations with event and action notations, called CommEvent and CommAction respectively, for this directed communication without any changes to the statecharts notation (Sections 4.15 and 4.14). These new notations are used in the ATN specification (Section 8.2.1).

Our framework is also extensible in its ability to add new analysis techniques without changing the notations. Our choice of higher-order logic as the base formalism ensures that new analysis techniques are not limited by notation. The creation of a reusable toolkit of core techniques facilitates extensions in analysis. We demonstrate this extensibility by creating and implementing two new techniques, which are both based on BDDs. First, we create a technique for checking for symmetry in the parameters of tabular specifications (Section 7.5.3). Symmetry checking is carried out on the separation minima example (Section 8.1.2). Second, we create a technique for simulating a specification with uninterpreted constants (Section 7.8). Simulation is carried out on the heating system example (Section 7.8) and the ATN (Section 8.2.2). The specifications of the ATN and the separation minima were completed prior to these new analysis procedures being implemented and are not changed to apply these techniques.

Our framework is systematic in that there is a methodology for the creation of new combinations of notations in a language. The categories of notational styles and join points guide and limit the specifier to ensure that they create a well-defined specification. To extend the framework with a new notation, it is only necessary to write a packaged embedding of the semantics for the new notation in higher-order logic. Symbolic functional evaluation is used for all notations and replaces the step of having to write a separate translator for each notation. Our approach is more accessible to a practitioner than previous approaches in two ways. First, SFE makes it possible to restructure the results of analysis. For example, we present the missing cases in a table in the same terms as the input. Second, we remove the need for any theorem proving knowledge on the part of the specifier to carry out analysis.

Achieving systematisation in writing the semantics of a notation is extremely difficult. An expert is needed to write the semantics to extend the set of notations. Therefore, rather than outlining a method, we provide examples for representative notations, namely statecharts and decision tables. Consequently, considerable portions of the dissertation are devoted to explaining the packaged embeddings in higher-order logic for the notational styles used in the examples (Chapters 4 and 5), which serve as guidelines for extending our framework to similar notations. For example, statecharts is chosen as an example notation because it is representative of a large collection of model-oriented requirements notations including RSML and SCR.

Our framework is rigorous in the direct use of the semantic definitions to determine the meaning of a specification. The new technique of symbolic functional evaluation evaluates the semantic definitions. This method removes the gap between a statement of the semantics of a notation and an implementation of a translator. Furthermore, as an entity in logic, the semantics themselves can be scrutinised for their correctness and other properties. For example, we use this ability to determine conditions under which the concurrent components of a statechart can be partitioned into multiple statecharts (Section 5.13). This property of statecharts aids in the configuration space explosion problem for complex statechart specifications.

The framework ensures rigour up to the point of handing the meaning of a specification to an analysis procedure. The implementations of the analysis procedures themselves could be made rigorous by writing formal specifications of their operations. A further step towards rigour would be a proof that the implementations satisfy the specifications.

Our choice of higher-order logic as the base formalism for the framework allows uninterpreted constants to be used in specifications. Many examples of their use can be found in the separation minima (Chapter 8.1.1). Also, our use of higher-order logic means that we do not require an abstraction step in the evaluation phase. The result of evaluating the semantic definitions is semantically equivalent to the original specification. Abstraction may be required for analysis, but it is not required for determining the meaning of a multi-notation specification in our framework.

We demonstrate the use of structure in the analysis of tabular specifications. Analysis of the heating system example shows the application of this technique in model checking, simulation, and completeness and consistency analysis. Using structure made the results more accurate than those produced by strictly BDD-based approaches (Section 7.5). Analysis of the separation minima also benefited from use of this technique (Section 8.1.2). Early results from using our framework on this example were unacceptable to a domain expert because the strictly BDD-based technique used did not have sufficient understanding of expressions involving numeric inequalities. Using structure addresses this problem to the satisfaction of our domain expert.

1.5 Contributions

The main contribution of this work is an extensible, systematic, and rigorous framework for analysis of multi-notation specifications, which may include uninterpreted constants. Our work provides specifiers with the means to explore new options in the combinations of notations with immediate access to well-known automated analysis techniques. Furthermore, we achieve this result without using the infrastructure of a theorem prover.

Our framework achieves the desirable qualities through the use of operational semantics in higher-order logic. The choice of this core technology led us to use type checking as a mechanism for regulating combinations of notations. It also led us to develop symbolic functional evaluation, which is a rigorous method for determining the meaning of a specification in any notation. Thus, the key ingredients for creating our framework are higher-order logic, operational semantics, type checking and symbolic functional evaluation. The choice of these general-purpose techniques avoids a multiplicity of special-purpose tools for notations and analysis.

1.6 Overview of the dissertation

Chapter 2 describes related work on both single-notation and multi-notation techniques. Chapter 3 briefly presents higher-order logic, the lambda calculus, and functional programming language implementation techniques. Chapter 4 presents notational styles. Chapter 5 describes the formal operational semantics for the example notational styles. While closely linked, the topics of these two chapters are separated because a specifier need only read Chapter 4. Chapter 5 is for an expert wishing to examine the semantics of the notations or to extend our framework to other notations. Chapter 6 describes symbolic functional evaluation. Chapter 7 presents the architecture of the tool implemented to support our framework and describes the automated analysis techniques currently available in our framework. The heating system example is used in Chapter 7 to illustrate the analysis procedures currently implemented in the framework. Chapter 8 demonstrates the use of our framework for the separation minima and the Aeronautical Telecommunications Network examples. Chapter 9 summarises the dissertation and offers thoughts on future work.

Chapter 2

Related Work

This chapter discusses related work on automated analysis of requirements specifications. The chapter is divided into three sections. The first section contains a survey of automated formal analysis of requirements specifications for particular languages. This survey includes examples of analysis procedures that will be discussed in Chapter 7 as illustrations of analysis in our framework. Our framework allows these analysis procedures to be applied to an extensible set of notations. Section 2.2 describes work directly related to our framework on analysis of multi-notation specifications. Section 2.3 describes work related to symbolic functional evaluation.

2.1 Survey of formal analysis of requirements

In this section, we present work on analysing specifications stated in requirements notations that were chosen for their suitability and expressibility. We describe work on analysing specifications in requirements notations such as RSML, SCR, modecharts, and Z. These notations are examples of languages that could be incorporated into our framework. Broadly speaking, this work can be categorised into three groups: *manual translation* from one notation (perhaps informal) into the input language of an existing analysis tool (Section 2.1.1); *automatic translation* from one formal notation into the input language of an existing analysis tool (Section 2.1.2), and *notation-specific analysis tools* (Section 2.1.3). Section 2.1.4 describes two examples of embedding languages in theorem provers for automated analysis.

2.1.1 Manual translation

This section discusses efforts that manually translate a specification in a chosen requirements notation into the interface notation of an existing tool to carry out analysis.

Model checking of SCR mode tables

The Software Cost Reduction Method (SCR) is a requirements specification methodology developed by Heninger, Parnas and Shore [Hen80] to specify the requirements for an existing flight program in the Navigation/Weapon Delivery System of the U.S. Navy's A-7 aircraft. The requirements document is about 500 pages. In its creation, the authors were forced to confront the problems encountered in applying specification techniques to real systems.

The SCR approach consists of data forms, functions, conditions, events, text macros, modes, modeclasses, condition tables, and event tables. It does not allow uninterpreted constants. Conditions are expressions that maintain a value for a non-zero amount of time. An event happens at an instant of time. Modes are very similar to states in a statechart. They "represent the history of events that have occurred in the program" [Hen80]. Functions are grouped by modes. A set of conditions (or invariants) associated with a mode should be true whenever the system is in that mode. These conditions can be specified by a mode condition table.

A modeclass is a group of related modes, which create one level of hierarchy. A system may consist of several modeclasses. The system moves between modes based on events. Mode transition tables describe this information in a tabular representation [AG93].

Atlee and Gannon [AG93] established a method for checking safety properties of SCR mode transition tables. They manually convert an SCR table into a format that can be input to the model checker MCB [CES86]. Properties, stated in Computational Tree Logic (CTL), describe invariants that should hold true in a mode. The specification was also tested for liveness properties of individual transitions. Their analysis uncovered instances of unexpected combinations of events and race conditions in a cruise control system and a water-level monitoring system.

Extra relationships between the values of conditions are manually added into the analysis. For example, conditions that the temperature is too high, too low or okay are mutually exclusive. They use this information to fill in “don't care” values in the mode transition table, rather than directly inputing it to the model checker. Two examples of relationships that they manually derived are enumeration and range enumeration. These two relationships are automatically recognised in our framework. Enumeration means encoding the elements of an enumerated type so conditions referencing these types are related. Encoding of enumerated types has been done previously by Hu [HDDY93] and is also used in our framework (Section 7.3.1). Range enumeration means partitioning the possible values of a numeric name into a finite set of ranges. This partition can then be encoded. In Section 7.3.1, we show how a form of these ranges can be automatically recognised. The other types of relationships added by Atlee and Gannon could be stated as environmental constraints in our framework to produce more accurate results from analysis.

Atlee and Gannon's analysis is carried out using a strictly Boolean representation. Each condition, such as “temp > 100”, is treated as one Boolean variable. Results from the model checker must be manually interpreted with respect to these abstractions. In our framework, we provide a means for the tool to reverse automatically the abstractions to present the results to the user in terms of the original specification (Section 7.3.3).

Model checking of RSML

The Requirements State Machine Language(RSML) [LHHR94] is a formal model-oriented language. TCAS II (Traffic Alert and Collision Avoidance System) was specified using RSML. The notation is very similar to statecharts [Har87] with the addition of AND/OR tables to represent conditions on the triggers of transitions. The use of tables allows complex sets of conditions (decision logic) to be described in a structured form.

Actions associated with transitions consist mainly of assignments and event generation. RSML does not allow uninterpreted constants. RSML specifications consist of state definitions, constant definitions, event definitions, input variable definitions, output variable definitions, interface definitions, macro definitions, function definitions, and transition definitions. RSML has a textual representation associated with the graphical representation of the state hierarchy.

Anderson et al. [ABB$^+$96] have applied model checking to one part of the TCAS II specification. They analysed approximately 30% of the TCAS II specification consisting of the requirements for a state machine called "Own-Aircraft". They manually translated the TCAS II specification in RSML into the SMV [BCM$^+$90] input language. The representation in SMV used 227 Boolean variables: 10 for events, 55 for states, and 162 for data. The semantics we present in Chapter 5, developed independently from their work, codify many of the steps they went through in the manual translation process. They made some simplifications in the translation process, such as abstracting calculated values. As a point of comparison, our ATN example presented in Chapter 8 requires 395 Boolean variables.

Anderson et al. checked for nondeterminism (where multiple transitions can be taken in the same step), mutual exclusion in functions defined by cases, and other global properties of the system. They encountered configuration space size limitations because of addition and comparison operations. They found it more efficient to do some of this manipulation outside of the model checking process. Their next configuration relation was 124,618 BDD nodes in size. Although highly dependent on variable order, size of the next configuration relation is a common measure used in the presentation of model checking efforts. Chapter 8 reports the next configuration relation size for the ATN example.

OMT and PVS

Lutz [Lut97] examines the link between formal specifications and re-use. A generic formal specification of a design for software monitors on a spacecraft was developed. The Object Modelling Technique (OMT) [R$^+$91] was used for understandability. A formal specification in the PVS theorem prover [ORS92] was then manually created from the OMT specification. PVS was used to prove safety and interface properties of the specification. Both of the OMT and PVS specifications were manually checked for conformity with the actual software design. Tracing was originally carried out between individual software monitors and the OMT specification but was abandoned in favour of tracing from the PVS specification. This work is an example where the manual translation approach suffers because the original notation chosen for its readability was abandoned for a notation that can be analysed. OMT is an example of a notation that could be integrated into our framework.

2.1.2 Automatic translation

In this section, we present two examples of automated translation approaches used for model checking analysis. In comparison to our framework, these approaches have the disadvantages of translation. For example, there may be a loss of rigour in the gap between the definitions of the semantics and the translator. Our framework removes the need to write a translator increasing the rigour of our approach and saving effort. Furthermore, we are able to return results in terms of the original specification.

Atlee and Buckley [AB96] created an automated means of producing input for

SMV from SCR mode transition tables using a "logic-model semantics". Sreemani and Atlee [SA96a, SA96b] evaluated the feasibility of this approach by model checking properties of the A-7E aircraft requirements in SMV. The specification consists of three concurrent components each with six to eighteen modes and 69 input conditions. Conditions in the mode transition tables are mapped to Boolean variables in the SMV specification. Their translator uses environmental assumptions about mutual exclusion in the SCR specification that relate conditions used in the mode transition table to produce automatically variables of enumerated types. Each row of a mode transition table turns into a case statement in SMV. The next configuration relation was represented in SMV in 64,490 BDD nodes and 117 Boolean variables.

Bharadwaj and Heitmeyer [BH97a] have added model checking capability to the SCR* toolset by creating a translator to turn SCR specifications into input for the SPIN [Hol97] model checker. The input language to SPIN, Promela, is similar to an imperative programming notation. They improve upon Atlee and Buckley's approach by allowing variables of enumerated types or integer subranges. They note the difficulty of interpreting counterexamples produced by SPIN and comment that it may be more appropriate to have a special purpose model checker for SCR.

Bharadwaj and Heitmeyer describe two techniques for statically reducing the size of the configuration space of the specification analysed. One is to eliminate irrelevant entities by examining the dependencies of the property being analysed. The resulting specification is equivalent to the original specification for the property in question. The second is to reduce the size of the specification under consideration conservatively by moving details into the environment in a more abstract form. Dependencies are determined between the structural components (i.e., the tables) of the specification. Therefore, these reductions are based on the structure of the specification.

2.1.3 Notation-specific analysis tools

This section discusses analysis tools developed for particular notations. Many of the techniques have been grouped into toolsets, which provide an editor and a range of analysis procedures to apply to a specification in the particular notation [HLCM92, HBGL95, LBH$^+$95]. Analysis techniques used in these toolsets, as well as techniques applied to other notations, are discussed in this section. These efforts have the disadvantage that a fixed set of notations is chosen and changes or additions to the notations may require changes to the analysis procedures because these procedures are notation-specific. Our framework overcomes these disadvantages by using the general-purpose base formalism of higher-order logic. This choice allows us to extend the set of notations without changing the analysis procedures.

Consistency checking of SCR tables

Heitmeyer, Jeffords and Labaw describe work on checking the consistency of tables in the SCR notation [HL93, HLK95, HJL96]. This work is part of the SCR* toolset [HBGL95].

They present a semantics for SCR in a predicate logic-like format. These semantics determine a model by associating tables with functions. Therefore, they only define the meaning of deterministic specifications in SCR. They describe a set of "consistency" checks, where consistency means that the specification is deterministic. Other types of analysis in the SCR* toolset assume the specification is deterministic and therefore these consistency checks must precede further analysis.

Two consistency checks are for coverage and disjointness. A table satisfies the property of coverage if the disjunction of the conditions in a row is a tautology. The property of disjointness means the conditions in each row are mutually exclusive. To examine these properties they use a tableaux-based decision procedure based on an algorithm described by Smullyan [Smu68][1].

[1]Not enough detail is provided to determine which tableaux-based algorithm is used although

Heitmeyer and Labaw [HL93] limited the specification to conditions ranging over Boolean values or those that have been converted by hand to Boolean variables. Expressions involving relations are also converted manually into Boolean variables. Their analysis of the condition tables for the Operational Flight Program of the U.S. Navy's A-7 aircraft found 17 legitimate errors in 36 tables having a total of 98 rows. Two false errors were found due to their strictly Boolean interpretation of the specification. They found errors such as not specifying behaviour for all of the values of variables, and ranges that did not form a partition. They were also able to determine when behaviour was not specified for some submodes. It is unclear whether the results of analysing specifications that involved encodings of enumerated types were automatically or manually mapped back to the correct level of abstraction for interpretation.

Domain completeness and consistency checking of RSML

In the TCAS II specification, a table can be used to describe the condition for taking a transition in a state machine. In the completeness and consistency analysis carried out by Heimdahl and Leveson [HL96], the specification is considered domain complete if a transition is always enabled from a state. It is consistent if the specification is deterministic, i.e., if no two transitions can be enabled at the same time.

An AND/OR table is a representation of an expression in disjunctive normal form. Each disjunct is represented by a column and the conjuncts are represented in rows. The cells in the rows can contain only true, false, or "don't care". Related conditions such as "$x < 280$" and "$x > 450$" are listed on separate rows.

In the analysis method of Heimdahl and Leveson, a Boolean variable is associated with each row label. The meaning of each cell in the row is the condition of whether this Boolean variable is true or false. Domain completeness analysis examines whether the disjunction of the columns of all the tables used to describe transitions from a state is a

because their tool is automated, we assume they use the one for propositional logic.

tautology. Consistency analysis checks that there is no overlap in the conditions between multiple tables describing transitions from the same state, i.e., that the conjunction of the meaning of two tables is a contradiction. They represent these properties as Binary Decision Diagrams (BDDs). Checking if the BDD representation of an expression is a tautology or a contradiction takes constant time.

Related information found on different rows in an AND/OR table is associated with independent Boolean variables. Therefore, the analysis method of Heimdahl and Leveson can produce spurious results. For example, it might return a result indicating that no table covers the case where both the conditions "$x < 280$" and "$x > 450$" are true. This case is impossible and distracting to reviewers of the analysis output. Their tool catches spurious results with respect to enumerated types, but not those arising from the use of mathematical functions. Inaccurate results can also be generated because the reachability of conditions is not examined, i.e., the analysis is only done on an individual state basis. Another source of spurious results is relationships between mathematical functions that cannot be captured in a BDD-based approach. Heimdahl provides a discussion of these issues [Hei96].

We have developed a slight variation of AND/OR tables that allow related conditions to be specified on the same row (Section 4.8). We found this notation appropriate for the separation minima example (Section 8.1). Furthermore, the structure of these decision tables can be exploited in analysis to eliminate some of the spurious errors produced by the analysis.

Tablewise

Tablewise [HC95] is a tool for creating and analysing decision tables. It carries out completeness and consistency checking similar to that for RSML and SCR. The output produced is also a table. However individual tables are independent and cannot be nested as in our framework. The tool automatically translates relational comparisons such as

"AC_Alt $\geq$ Acc_Alt" to an expression in terms of enumerated types: "compare(AC_Alt,Acc_Alt) ϵ { EQ, GT }". A form of "structured analysis" is used to summarise output cases from completeness checking[2]. They use finite decision diagrams (a generalisation of binary decision diagrams with a finite number of branches) in their implementation. Environmental constraints cannot yet be included to eliminate infeasible constraints.

AVAT

Dai and Scott describe a CASE tool supporting software verification and validation called AVAT [DS95]. It provides support for Program Function (PF) tables, which were developed by Parnas [Par92, Par94] and used in the verification of software for the shutdown systems of the Darlington Nuclear Reactor Power Plant [CGR95]. A PF table expresses the relationship between inputs and outputs of a program. A parser for the tables has been developed in Prolog. This parser is used to check the completeness and consistency of the tables as syntax checks. AVAT also compares PF tables that represent the same information but were prepared independently. Comparison of tables was performed manually in the Darlington project.

Nitpick

Damon, D. Jackson and Jha have developed a model checker called Nitpick for relational specifications over finite sets based on the Z notation [DJJ96]. They use BDDs and an encoding scheme for scaler types and finite sets similar to that used by others [HDDY93].

They have developed three ways to reduce the size of the configuration space based on the structure of the specification: isomorph elimination, derived variable detection, and short circuiting (identifying variables that may be unnecessary to the current analysis

[2]Not enough detail is provided to determine whether this technique relates to the Structured Analysis of DeMarco [DeM79].

because of a partial assignment). Isomorph elimination involves reducing the number of cases based on determining cases of “similar shape”. Also, by separating schemas for transitions and states, they are able to incorporate state invariants into the analysis. The technique we present based on structure (Section 7.3.1) not only reduces the size of the configuration space but also increases the accuracy of analysis results.

Timing analysis of modecharts

Modecharts [JM94] is a state-transition based graphical specification language developed by Jahanian and Mok. It was developed for expressing and examining timing requirements of a system. Transitions are labelled with delays and deadlines, which indicate when the transition can be taken. Actions have start times and stop times.

For timing analysis [JS88], a modechart specification is automatically translated into a set of formulae in Real Time Logic (RTL). RTL is essentially integer arithmetic without multiplication. RTL includes an uninterpreted occurrence function, which indicates the absolute time at which the ‘ith’ occurrence of an event occurs. Universal and existential quantification are possible only over integers. The meaning of a modechart in RTL is described entirely in terms of the following events: entering a mode, exiting a mode, starting an action, stopping an action, changing the value of a variable (which can only be done in an action), taking a transition, and external events (such as pressing a button).

Jahanian and Stuart [JS88] present an algorithm for building a computation graph based on a modechart specification. A computation is “an assignment of time values to the events on a path (perhaps infinite) from the root of the tree such that it is consistent with lower/upper bound requirements on the events” [JS88]. Two classes of timing properties are presented that can be verified using the computation graph. The first class concerns the relative ordering and time separation of events. For example, a property might state that “two successive entries to the mode M are either ten time units apart or within five time units of each other". The second class of properties relates one interval to another and can be used to state that two actions are mutually exclusive. Jahanian and Stuart limit their variables to the Boolean domain. Because the specification is described entirely in terms of occurrences of events, rather than constraints on the modes, they cannot prove invariants such as safety properties as in CTL model checking.

RTL is used both to represent the meaning of a modechart specification and to express the properties to be checked. Representing both the specification and the properties in the same notation is advantageous in that it may help to create a compositional method where large specifications can be analysed in parts. A property resulting from one analysis can be used directly in another because the property is in the same language.

Timing analysis is not currently implemented in our framework. This section is provided as an example of another types of analysis that could extend the current set of analysis techniques provided by the framework.

2.1.4 Theorem provers

The technique of embedding notations in higher-order logic for their study was pioneered by Gordon [Gor88a]. Two examples of work on doing automated analysis of model-oriented notations using theorem provers are presented in this section.

Model checking of statecharts

In previous work [Day93], we presented a semantics for statecharts [Har87] in higher-order logic using event and action notations very similar to those used in the CASE tool STATEMATE [HL$^+$90]. This effort used a deep embedding of the notations and operational semantics. The HOL-Voss [JS93] tool was used to model check the statecharts specifications. The semantics for statecharts presented in Chapter 5 are an extension of this previous work.

In this dissertation, different decisions about the means of embedding notations are made to incorporate an extensible set of notations in specifications. The specification is also allowed to include uninterpreted constants. We further provide an extensible and systematic framework for analysis of multi-notation specifications. Also, in our framework higher-order logic is directly linked with automated analysis without a theorem prover.

Embeddings in PVS

The Prototype Verification System (PVS) [ORS92] is a theorem prover for higher-order logic (which includes uninterpreted constants). Type checking is undecidable in PVS because it allows predicate subtyping.

Rajan, Shankar and Srivas [RSS95] showed how Boolean simplification using BDDs and model checking can be integrated as a decision procedure in PVS. They create a next configuration relation directly in the PVS logic. We differ from their approach in two ways. First, we create a framework that does not rely on theorem proving infrastructure. Our architecture links higher-order logic directly with automated analysis. Section 1.1.1 gave a description of the benefits of our approach in this regard. Second, we provide a distributed mechanism for grouping the elements of the configuration rather than a record structure. Our approach is more convenient for the specifier but requires an extra step in our process of determining the grouping (Section 7.6).

PVS has been used by Owre, Rushby and Shankar for completeness and consistency checking and model checking of decision tables and SCR tables [ORS96, ORS97]. They introduce a new construct into the PVS logic for tables and internally translate this construct into their existing COND and CASES constructs. This translation defines the semantics for tables in a similar manner to the semantic embedding of tables in Chapter 5. The original form of the table is remembered for output of the definition. Associated with the COND and CASES constructs are type correctness conditions of disjointness (consistency) and coverage (completeness). Thus, completeness and consistency checking is carried out through the proof that the type correctness conditions hold. If a proof of one of these conditions does not succeed, the result is an unproven sequent. They show an example of how a missing case in completeness checking would be a sequent with no assumptions and several formulae to prove. This method requires tables to be complete and consistent to pass type checking. It does not enumerate cases that fall into a "default" column of a table or present the results as a table. In using formal methods for an independent validation and verification effort, Easterbrook and Callahan abandoned the use of PVS to carry out completeness and consistency checks because of the difficulty of determining the source of an inconsistency in a failed proof [EC97]. A further difficulty with the PVS approach is obtaining counterexamples from model checking in terms of the original specification.

Model checking of a specification stated in SCR mode transition tables in PVS is carried out by grouping previous and next configuration inputs into a "state" record. Used as inputs to the table, the table then becomes a transition relation for their integrated model checker. The configuration must include only elements of finite type or a finite abstraction must be created. Mode transition tables represent concurrent components whose next configuration relations can be conjoined together. Within each component there is one level of hierarchy, so each row in the table represents a transition and the meaning of the rows can be disjoined together. Therefore, their semantic embedding within PVS is straightforward.

Statecharts allow a more hierarchical state structure and priority of transitions based on hierarchy, which complicates the semantics of statecharts in comparison to SCR mode transition tables. We have chosen statecharts as a representative notation for demonstrating the framework and providing guidelines for extending the framework to other notations.

Owre et al. [ORS96] acknowledge the superiority of lightweight special-purpose tools (such as the SCR* toolset) for particular notations and analysis techniques, partly because of their scalability. The lack of scalability could be in part because there does not appear to be any way to control the variable ordering for BDDs within PVS. Because of the importance of this factor in the scalability of many automated techniques, we address the issue directly by allowing the input of a variable order (Section 7.3.4). By providing a lightweight interface between a general purpose notation and automated analysis, this dissertation offers a middle ground between special-purpose tools and general-purpose theorem provers.

2.2 Multi-notation specification and analysis

This section discusses three efforts comparable to the one presented in this dissertation for specifying and analysing multi-notation specification.

2.2.1 Translation to first-order logic

Zave and M. Jackson [ZJ93] present algorithmic translations from a variety of notations to one-sorted first-order logic with equality. Their approach is illustrated for Z, deterministic finite automata, data flow diagrams, Petri nets and other notations. The overall system is specified by the conjunction of the parts written in different notations. Their semantics for these notations are captured by their algorithmic translations.

Similar to the reasons why we choose higher-order logic as a base formalism (Section 3.1), their choice of first-order logic is for its generality. Our prospects for generality for notations by using higher-order logic (Section 9.2.1) is supported by their statement that they are "fairly" confident that all notational styles can be represented in first-order logic. However, with a one-sorted logic they are unable to use type checking for regulation of combinations as we can in higher-order logic.

Zave and M. Jackson express the meaning of a notation in first-order logic. We capture the translation process in higher-order logic in the definitions of the semantics. Compared to our approach, the work of Zave and M. Jackson has the disadvantage that the translator lacks rigour. They do not demonstrate any analysis on their multi-notation specifications. But because they use a translation approach, the results of analysis could not be presented to the specifier in terms of the original specification. They describe the logical formula in first-order logic resulting from the translation as "large and incomprehensible".

The method of Zave and M. Jackson does not rely on the parts of the specification in different notations being independent. They deal very generally with any type of notation rather than just model-oriented notations. Thus, the first problem they face is whether the specification is consistent, where they use the word "consistent" to mean that the specification is satisfiable. They suggest that consistency properties can be considered with respect to possible overlap between notations rather than just for particular specifications. By limiting our scope to model-oriented notations, we are able to produce guidelines for how notations can be combined such that a specification has a well-defined meaning that is not inconsistent.

2.2.2 Ada, regular expressions, and GIL in hybrid automata

Avrunin, Corbett and Dillon [ACD97] describe an approach for analysing systems partially implemented in Ada. The motivation for their approach is that development and maintenance of systems do not proceed at a uniform rate. Therefore, it is useful to be able to take advantage of implementation details when available, especially for the sake of timing analysis, combined with specification information about non-implemented components. Another rational for their approach is that it is useful to specify the behaviour of the environment for the analysis tools and one would not necessarily want to construct software for this purpose. Finally they describe the value of compositional analysis where some parts are abstracted to make computations feasible.

Their approach uses the common formalism of constant slope linear hybrid automata, which are supported by the HyTech verifier [AHH96]. The specifications are stated in regular expressions (for ordering of events) and a temporal logic called the Graphical Interval Logic (GIL) for timing constraints. These specifications are translated by hand to automata. Hybrid automata are automatically constructed from the Ada programs conservatively. Specifications of environmental behaviour can be written in GIL or in regular expressions. The system is specified by means of an intersection operation on hybrid automata. Properties are also stated in hybrid automata.

For the sake of timing analysis, which is particularly important for real-time systems, they choose hybrid automata as their common representation. They point out that real-time performance may depend on details found in the implementation, thus requiring the integration of at least parts of the implementation (in this case in Ada) in the analysis.

This work falls into the category of translations from one notation to another and therefore has the previously discussed disadvantages of translation. There is a potential to express hybrid automata in higher-order logic (e.g., the formalisation of timed transition systems by Cardell-Oliver et al. [COHH92]). Analysis techniques aimed at timing verification of systems with continuous variables could be used within our framework.

2.2.3 Combining notations using hypergraphs

Pezzè and Young describe an approach to combining state transition notations using an "inframodel" of a hypergraph for configuration space exploration [PY97]. In form, a hypergraph is very similar to a labelled Petri net. A specification in a particular notation is translated to the inframodel and rules associate the semantics of the original notation with the part of the inframodel in that notation. The rules govern enabling, firing and matching of arcs in the hypergraph. Matching indicates sets of arcs that can fire together. Pezzè and Young demonstrate how Petri nets, Ada tasking, statecharts with history, and state diagrams of RSML can be translated to the inframodel. For multiple parts of the inframodel stated in different notations to operate concurrently, rules for matching arcs that belong to models in different notations are required. Their approach can be described as instantiating the hypergraph for a particular flavour of state-transition diagrams or particular combination of notations.

We differ from their approach in three ways. First, we use a base formalism of higher-order logic, which is more convenient for expressing non-state-transition based notations such as decision tables. This choice allows us to include uninterpreted constants and does not restrict us to a particular type of analysis. For example, we are able to use the semantics to reason about the notation (Section 5.13). Second, we directly use the semantics of the notation in analysis. Their approach involves translation to a hypergraph. It is unclear as to how well the results are returned to the user in terms of the original specification. Third, we define the behaviour of multiple models through conjunction and do not require additional information for particular combinations of notations. We concentrate more on the combination of notations used to build one model.

The goal of the work of Pezzè and Young is to have a general approach to building tools for reachability analysis and model checking for multiple state-transition notations, such that compact representations for particular notations are not lost. Their rule-based approach to mapping a particular notation into a common base notation could be used to help systematise the process of writing the operational semantics for notations in the semantic category of models.

2.3 Evaluation of logic

Symbolic functional evaluation involves evaluating expressions in higher-order logic. Evaluation is relevant both as a means of evaluating the semantic definitions that define the meaning of a notation and as a method for carrying out simulation or symbolic simulation of functional specifications.

To carry out simulation, a specification of a next configuration function can be iteratively applied to itself a finite number of times representing the results of that number of steps of simulation. If the specification includes uninterpreted constants or symbolic inputs, the simulation must be symbolic.

Evaluation, i.e., definitional expansion and beta-reduction, is usually carried out in theorem provers using rewriting and beta-reduction. Joyce [Joy89] and Windley [Win90] use rewriting and beta-reduction in HOL to carry out symbolic simulation of specifications of hardware circuits in higher order logic. Goosens [Goo93] and van Tassel [Tas93] carried out symbolic simulation of specifications in deep embeddings of hardware description languages using the same techniques. The semantic functions were expanded using rewriting and beta-reduction. Donat [Don98] applies rewriting to higher-order logic requirements specifications outside of a theorem prover for test case generation.

The Boyer-Moore logic used in the Nqthm theorem prover is a first-order logic without quantifiers [BM88]. Their syntax is very close to LISP. Every function in the logic has an "executable counterpart" in a Lisp procedure. A command called R-LOOP provides an execution environment. When invoked on expressions that apply a function symbol to concrete values (i.e., expression without variables), R-LOOP calls the executable counterpart of the function to return a concrete value for the expression. In a more recent version, called ACL2, the syntax is actually a subset of LISP so no separate execution environment is needed [KM97]. This choice was made for efficiency.

Camilleri [Cam88] developed a method for translating executable hardware specifications into ML [Pau91] where they could be simulated. This technique is applicable to a subset of higher-order logic and involves an initial step of translating relational specifications of hardware into functional ones. Rajan [Raj93] took a similar approach in translating a subset of higher-order logic into ML. In both of these approaches, specifications cannot include uninterpreted constants.

By using BDDs to represent Boolean variables in a functional programming language, Seger developed a method of symbolic simulation for hardware specifications [SB95]. These specifications cannot include uninterpreted constants.

Eisenbiegler and Kumar [EK94] provide a discussion of many of the issues concerning the symbolic simulation of specifications through evaluation of HOL terms.

Andrews [And97] translates specifications in S [JDD94] (a syntactic variant of higher-order logic) into the Lambda Prolog programming language. Uninterpreted constants can be used in the specification. Expressions with quantifiers can be executed using the unification and backtracking of the programming language. A drawback of this approach is that functions that do not return Boolean values cannot be represented directly in Lambda Prolog.

Our approach differs from all of these efforts in that symbolic evaluation is carried out directly on the specification using efficient techniques from implementations of functional programming languages.

2.4 Summary

This chapter has described work related to this dissertation. A variety of examples of analysis of requirements specifications were presented. We differ from other multi-notation approaches in our use of higher-order logic as a base formalism. We limit our scope to model-oriented notations to achieve systematisation, but we include notations such as decision tables. Compared to existing translation approaches, we directly use the semantics of notations in analysis to achieve rigour. As entities in logic, our semantics definitions can be analysed for properties of the notation itself. Our packaged embeddings require no changes to the existing set of notations for extensions of the framework. Symbolic functional evaluation allows us to include uninterpreted constants and maintain rigour while avoiding a theorem proving environment or translation to an executable programming language.

Chapter 3

Foundations

This chapter provides a brief introduction to the foundations of our framework, namely, higher-order logic, the lambda calculus, and techniques for the evaluation of functional programming languages. Higher-order logic is the base formalism used in the framework. Constant definitions do not require any extensions to higher-order logic.

The lambda calculus is used to represent functions and function applications in higher-order logic. It evolved out of higher-order logic as an area of study in itself and is the foundation for functional programming languages. The lambda calculus (and therefore higher-order logic) includes rules for evaluating expressions. Section 3.4 discusses an implementation for evaluation of expressions in the lambda calculus without free variables.

The reader is assumed to have some familiarity with these three foundational concepts. Higher-order logic is used in the next two chapters for representing notations and their semantics. Chapter 6 draws upon the evaluation technique to create a technique for evaluating expressions in higher-order logic including uninterpreted constants (free variables).

3.1 Higher-order logic

The base notation used in this dissertation is essentially the higher-order logic of the HOL theorem proving system [Gor87, GM93]. HOL uses a slight variant of Church's simple theory of types developed in 1940 [Chu40]. The reader need only be familiar with the notation of higher-order logic and not the HOL system. A presentation of the notation can be found in Chapters 2 and 15 of Gordon and Melham's book [GM93]. Expressions in higher-order logic are variables, constants, abstractions, or applications (although there is greater variety than this in the concrete syntax described in Chapter 4). Abstractions are functions in the lambda calculus, such as $\lambda x.x+1$ and will be discussed in more detail in the next section.

The reasons for choosing higher-order logic as the base notation for our framework are:

- *generality:* The notation does not impose any particular style of specification. It has been shown to be a useful notation for studying other notations by embedding them in the logic [Gor88a]. The introduction of a constant definition as a new axiom in higher-order logic is a conservative extension of the logic, which means it does not change the set of derivable formulae of the logic. Definitions contribute to the generality of the notation because notational styles can be created through the introduction of keywords as defined constants.

- *convenience of expression:* The ability to have functions take other functions as arguments, and to describe unnamed functions using the lambda calculus are features that increase the convenience of the notation and are exploited in the semantic functions.

- *type checking:* Using a typed notation has great benefits for early detection of errors in a specification as in typed programming languages [LP97].

- *decidable type checking:* Type checking is decidable for the variant of higher-order logic chosen [Mil78][1]. Polymorphism allows a function to represent a set of functions of various different types. The most general type can be determined for any expression in higher-order logic[2]. This choice is in keeping with our intention to create a fully automatic system.

The logic of the HOL system serves as a well-documented foundation for our approach of using higher-order logic. In this work, we have not directly used the HOL system or any other theorem prover. As opposed to an approach based on theorem proving, we do not provide support for deriving well-formed formula through interactive proof.

Because higher-order logic is undecidable, it is not possible to check automatically all properties stated in the logic. There are useful classes of expressions in the logic for which reasoning can be automated. Our framework uses techniques applicable to these expressions. The reasoning performed by the automated analysis techniques can be replicated using only the axioms and inference rules of the logic.

Section 3.4 describes one analysis technique for evaluating expressions in higher-order logic which is based on the lambda calculus. This technique provides the foundation for symbolic functional evaluation. Examples of other analysis techniques, which are less fundamental to the framework, are found in Chapter 7.

3.2 Typed lambda calculus

The typed lambda calculus is used as a notation for functions in higher-order logic. In 1941, Church began to investigate the use of the lambda calculus separately from higher-order logic. Functional programming languages have evolved from this foundation. In this

[1]Our implementation is based on the algorithm for type checking found in Hancock [Han87] which is based on Milner's work.

[2]Predicate subtyping, where type membership depends on the value of a predicate application, is not allowed because it is not decidable. Rushby [Rus97] discusses the value of predicate subtyping.

section, we present the typed lambda calculus used in higher-order logic. Our presentation of this material is drawn from Peyton Jones [Jon87] and Gordon [Gor88b].

The lambda calculus consists of expressions involving variables, function application $(f \ \ x)$ and lambda abstractions $(\lambda x.x + 1)$. Higher-order logic makes a distinction between variables and constants which is not present in the lambda calculus. Constants in higher-order logic are free variables in the lambda calculus. A free variable is one that is not bound by a lambda abstraction or quantifier. Lambda abstractions represent functions.

There are three rules that can be used to "calculate" with the notation. These rules involve symbol manipulation and produce equivalent expressions. They are:

- *alpha-conversion*

  The names of lambda variables can be changed provided the same substitution is made within the body of the lambda abstraction and this substitution does not cause any variables that were previously free in a sub-expression to become bound.

  $$\lambda V.E = \lambda V'.E[V'/V]$$

  $E[V'/V]$ is the expression resulting from substituting the variable V' for the variable V in the expression E. Table 3.1 defines substitution as found in Gordon [Gor88b]. *Name capture* (also called *variable capture*) occurs when a variable that is free in the argument becomes bound in the resulting expression. The definition of substitution in Table 3.1 ensures name capture does not happen.

- *beta-conversion*

  An expression that consists of a lambda abstraction applied to an argument can be reduced by substituting the argument as the lambda variable in the expression that is the body of the lambda abstraction.

  $$(\lambda V.E_1)E_2 = E_1[E_2/V]$$

Table 3.1: Substitution

E	$E[E'/V]$
V	E'
V' (where $V \neq V'$)	V'
$E_1 E_2$	$E_1[E'/V] E_2[E'/V]$
$\lambda V.E_1$	$\lambda V.E_1$
$\lambda V'.E_1$ (where $V \neq V'$ and V' is not free in E')	$\lambda V'.E_1[E'/V]$
$\lambda V'.E_1$ (where $V \neq V'$ and V' is free in E')	$\lambda V''.E_1[V''/V'][E'/V]$ where V'' is a variable not free in E' or E_1

For example, $(\lambda x.x + 1)\, 2$ becomes $2 + 1$. This substitution must ensure no variables that are free in the argument become bound in the resulting expression. *Beta-reduction* involves turning an expression in the form of the left-hand side of this rule into the right-hand side. Because the argument to the abstraction is not evaluated in beta-reduction, it is called *lazy evaluation*. *Beta-abstraction* is the reverse [Jon87]. Beta-reduction is one of the primitive inference rules of higher-order logic.

- *eta-conversion*

  If the only place the variable of a lambda abstraction appears is as the last argument of an expression forming the body of the abstraction, an equivalent form of the expression is to drop the abstraction variable and its usage at the end of the expression.

$$\lambda V.(E\ V) = E$$

Evaluation of an expression in the lambda calculus is the process of reducing or simplifying an expression using these rules. There are two common goals of evaluation:

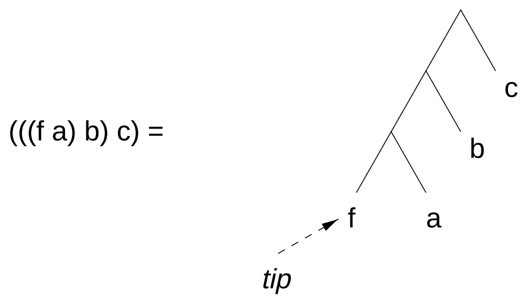

Figure 3.1: Tip of a function application

- *Normal Form*

  A *redex* (reducible expression) is an application consisting of a lambda abstraction applied to an argument. A redex can be reduced using beta-reduction. An expression is in normal form when no redexes remain in the expression.

- *Weak Head Normal Form*

  The leftmost branch of a series of function applications is called the *tip* as illustrated in Figure 3.1. An expression is in weak head normal form if 1) it is an application whose tip is a variable or 2) it is an irreducible expression (i.e., it is not a redex). Inner redexes can exist.

The first and second Church-Rosser Theorems, which are applicable to the untyped lambda calculus, are discussed in Gordon [Gor88b]. The first Church-Rosser Theorem states that there is a unique normal form for an expression if a normal form exists. The second Church-Rosser Theorem guarantees that if a terminating reduction sequence exists to simplify an expression to normal form, it can be found using *normal order reduction.* This form of reduction is where the leftmost, outermost redex is simplified first. These theorems also apply to the typed lambda calculus.

3.3 Definitions in higher-order logic

A specification or program usually contains definitions of constants. Constant definitions in higher-order logic are a conservative extension of the logic. This is of critical importance in our work because it allows notational styles to be created through the introduction of keywords without any change to the proof theory of the logic. This section discusses how defined constants are evaluated in the lambda calculus.

A constant defined using a non-recursive or non-pattern matching definition can be considered the name of a lambda expression. For example, the following is a definition:

$$\mathrm{n} := 1$$

This definition means that in subsequent description the name "n" can be used for the number "1". An example of a definition with parameters is:

$$\mathrm{add}\ \ a\ \ b := a + b$$

The parameters of the definition can be viewed as variables bound by lambda abstractions in the expression on the right-hand side of the definition. This definition is equivalent to:

$$\mathrm{add} := \lambda a.\lambda b.\ a + b$$

When a function is used in an expression, the appropriate definition body is substituted for the function name during evaluation. The following two sections discuss recursive definitions and pattern matching definitions.

3.3.1 Recursive definitions

In a recursive definition the constant being defined also appears on the right-hand side of the definition. For example, the following definition is for a constant "sum" that sums the natural numbers from 1 to n, where n is the parameter of the definition:

$$\mathrm{sum}\ \ := \lambda n.\mathrm{if}\ \ (n > 0)\ \ \mathrm{then}\ \ n + \mathrm{sum}(n-1)\ \ \mathrm{else}\ \ 0$$

A definition is treated as an axiom in higher-order logic. As shown in Peyton Jones [Jon87], applications of recursively defined functions can be evaluated the same way as applications of non-recursively defined functions.

The general form of a recursive definition is:

$$\text{recfcn} := \lambda x.\ \ldots \text{recfcn} \ldots x \ldots$$

The variable "x" is the argument to the function. (There can be multiple arguments.) Evaluating an application of this function in the same way as a non-recursive function means "recfcn n" can be reduced to "$\ldots \text{recfcn} \ldots n \ldots$". Using beta-abstraction, "recfcn" is equivalent to:

$$\text{recfcn} = (\lambda r.(\lambda x.\ \ldots r \ldots x \ldots))\text{recfcn}$$

Equality is used here, rather than the definition sign (":="), because this statement can be considered an equation to solve for the value of "recfcn". The function "recfcn" is the least fixed point (by the definition of recursive functions) for the function "$(\lambda r.(\lambda x.\ \ldots r \ldots x \ldots))$".

3.3.2 Pattern matching definitions

A special group of constants in logic, called type constructors, usually have special status in the lambda calculus. Pattern matching function definitions over constructors for a type can be very useful in writing specifications and are included in higher-order logic. An example of a pattern matching definition over the type "list", which has two type constructors, "NIL", and "CONS", is:

$$\begin{aligned}&\text{length}(\text{NIL}) := 0 \ \mid \\ &\text{length}(\text{CONS}\ e\ l) := 1 + (\text{length}\ l);\end{aligned}$$

Uncurried function definitions[3] can be seen as pattern matching definitions over the pair type.

[3]An uncurried function definition takes parameters as a tuple.

Definitions in higher-order logic based on pattern matching can have varying degrees of complexity. Our notation includes only those where

- the multiple cases do not overlap (i.e., each must have a unique type constructor as part of the pattern to match)
- only the first argument to a function can be defined using patterns
- patterns are not nested
- variables cannot be repeated in the pattern

The definition need not cover all possible constructors. These choices were made to limit the complexity of the evaluation algorithm discussed in Chapter 6. The algorithm could be extended to handle extra degrees of complexity in pattern matching definitions. In our experience, these restrictions were not limiting in writing the semantic definitions or specifications.

As shown in Peyton Jones [Jon87], evaluation of a function defined by pattern matching can be achieved by stating its meaning as a lambda abstraction. The appropriate lambda abstraction can be stated using a function called "unpack_sum". This function is so named because types defined by constructors are the union of values that can be formed from the constructors. For a definition of the form:

$$\begin{array}{l} \mathrm{c}(s1\ v_0 \ldots v_n) := E1 \ \mid \\ \mathrm{c}(s2\ v_0 \ldots v_m) := E2 \ \mid \\ \ldots \end{array}$$

The constant "c" is the function being defined, "$s1$" and "$s2$" are type constructors, and "v_i"'s are variable, "c" is semantically equivalent to:

$$\mathrm{unpack\_sum}[s1\ v_0 \ldots v_n; s2\ v_0 \ldots v_m; \ldots]$$

The definition of "unpack_sum [$s1\ v_0 \dots v_n; s2\ v_0 \dots v_m; \dots$]" is:

$$\begin{array}{l} \lambda x.\text{case } x \text{ of} \\ s1\ x_0 \dots x_n \Rightarrow (\lambda v_0 \dots v_n.E1)\ x_0 \dots x_n \\ s2\ x_0 \dots x_m \Rightarrow (\lambda v_0 \dots v_m.E2)\ x_0 \dots x_m \\ \dots \\ \text{else} \Rightarrow \text{FAIL} \end{array}$$

The "case" operator means the function returns the right-hand side of the "$\Rightarrow$" if the variable "x" matches the left-hand side of the arrow. The check for matches proceeds in the order of the list of options.

Even if the pattern matched argument (the first argument) is never used in the body of the function, it must be possible to match it to some case or else evaluation does not proceed. As a result, the first argument cannot be evaluated lazily[4]. The arguments to the constructor are not evaluated when carrying out the pattern match; rather they are treated as arguments to the function once a match has been determined.

In practise, "unpack_sum" is not used directly but its behaviour for functions such as "c" is implemented in rewriting or evaluation.

3.4 Evaluation of functional programming languages

Functional programming languages are based on the lambda calculus and methods for the efficient evaluation of expressions in the lambda calculus have been developed. This section discusses a technique for evaluation of a functional program. Our presentation of this material is drawn from Peyton Jones [Jon87].

Figure 3.2 presents an algorithm found in Peyton Jones [Jon87][5] which implements

[4]Here we do not make a distinction between pattern matching definitions with only one case (called product constructor patterns) and those with multiple cases (called sum constructor patterns). There are situations where lazy evaluation is applicable for product constructor patterns.

[5]There appears to be an error in this algorithm found on page 213 of Peyton Jones [Jon87]. In the case of a constructor, the algorithm says an error should be returned if the argument list is

normal order reduction to achieve weak head normal form for an expression. This algorithm is presented in C-like pseudo code. This algorithm is intended to work in conjunction with a function that prints the output. Since the output could be an infinite list, results are printed as soon as they are available. Therefore, this algorithm returns an expression in weak head normal form so the outermost identifier (usually a type constructor) can be printed, then the printing function calls the evaluation algorithm for the arguments to attain normal form output.

An implementation of this algorithm represents the expression as a graph, with expressions sharing pointers to common subexpressions. The principle of referential transparency (first used by Whitehead and Russell [WR27]) is that any subexpression without free variables will evaluate to the same normal form. Therefore, the algorithm carries out *evaluation in place*, i.e., replace the expression being evaluated with the result, so the expression is evaluated at most once.

This algorithm uses a technique called *spine unwinding.* A list of the arguments of a function application is kept while proceeding with the left branch of the function application. This operation is found in the `APPLICATION` case of the algorithm.

If the expression is an abstraction, beta-reduction is carried out. The arguments are substituted for the variables in the body of the lambda abstraction. Substitution proceeds by pointer substitution rather than copying the argument so that the same subexpression is not evaluated more than once. In this case, the result also takes the place of the original redex.

If a constructor is encountered at the head of the function application, the expression is in weak head normal form and evaluation stops.

For efficiency, built-in functions such as addition can be evaluated by an algorithm rather than by using their definition in the lambda calculus. The algorithm associated

not null. It is often the case that constructors such as "CONS" have arguments, so this error has been corrected in our presentation.

```
expression EvalNode(expression *exp, expressionlist arglist)

switch (formof(exp))

case APPLICATION:
    new_arglist = add argument of the application to the head of arglist
    return EvalNode(function of the application, new_arglist)

case ABSTRACTION:
    if (arglist == NULL) then
        return exp
    else
        newexp = substitute arglist for args in exp using pointers
        result = EvalNode(newexp, leftovers in arglist)
        overwrite root of redex with result
        return result

case CONSTRUCTOR:
    return exp applied to the arglist

case BUILT-IN FUNCTION:
    if too few args in arglist for this function then
        return exp applied to the arglist
    else
        evaluate the arguments
        result = evaluate the built-in function
        overwrite root of redex with result
        return result
```

Figure 3.2: Implementation of normal order reduction

with a built-in function is called a *delta-conversion*. It must preserve the properties of the lambda calculus such as the Church-Rosser Theorems. In Figure 3.2, if the expression is a built-in function with the appropriate number of arguments, evaluation proceeds using a delta-conversion. The redex is then replaced with the result.

This algorithm implements lazy evaluation, where the arguments to a function are only evaluated when they are used, and expressions are evaluated in place.

Extending this algorithm to work with function definitions involves replacing the name of the function with its definition.

Free variables

The above algorithm had no case for variables appearing in the expression. In a functional program, variables are expected to be substituted with arguments and therefore never encountered at the tip. The lambda expression representing a functional program never has any free variables [Jon87]. Consequently, in using normal order reduction, arguments being substituted for parameters never contain any free variables and therefore substitution does not need to deal with the problem of name capture. The absence of free variables makes combinator theory applicable for implementations of functional programs [Jon87].

Uninterpreted constants are free variables in a lambda expression. With free variables, the substitution function used in the previous algorithm must have the behaviour specified by Table 3.1 on page 52. Searching for free variables that could become bound can be computationally expensive. Chapter 6 discusses the value in distinguishing uninterpreted constants from other variables for efficiency and presents an algorithm for evaluation that uses this optimisation.

3.5 Summary

This chapter has presented the foundations of our framework. Higher-order logic will be used in the next two chapters both as a notation for specification and as the means of defining the semantic functions.

Chapter 6 revisits the concepts of the evaluation of functional programs to carry out evaluation of the semantic functions for different notations defined in higher-order logic. Lazy evaluation is used, but allowances are made for uninterpreted constants (free variables). The result is an evaluation technique more efficient than rewriting because general unification of terms is not needed. We further define levels of evaluation as gradients between weak head normal form and normal form that are useful for different analysis techniques.

Chapter 4

Integrating Model-Oriented Notations

This chapter elaborates on our solutions to the first four sub-problems presented in Chapter 1, namely, integrating combinations of notations (Section 1.2.1), representation of notations (Section 1.2.2), regulating combinations of notations (Section 1.2.3), and expressing dynamic behaviour (Section 1.2.4).

A formal notation is a set of keywords, user-created identifiers, and rules for combining these to produce well-formed sentences. The keywords and sentences may be textual or graphical. The keywords and rules together form the concrete syntax of the notation. Associated with each well-formed sentence is a meaning.

Formal requirements languages, such as SCR and RSML, have identified different notations that work well for different parts of specifications. For example, AND/OR tables are used to specify triggers of transitions in RSML. This chapter addresses the question of how the integration of notations can be systematised. Our approach makes new combinations of notations possible and the set of notations extensible. New combinations are demonstrated in this chapter in the heating system example. For example, a decision table is used in the action of a statechart. An expression in higher-order logic including a quantifier is the trigger of a statechart transition. We also demonstrate two new notations for events and actions for directed communication in statecharts.

The term "notation" is used to represent a collection of keywords. A notation can be as small as a single keyword. The chapter begins with a definition of model-oriented notations in Section 4.1.

To systematise the way in which notations can be combined, we begin by identifying four categories of notations common in model-oriented specifications. These categories are expressions, events, actions, and models. A type signature is associated with each of these categories indicating the type of element denoted by a notation in this category. Type checking regulates the combinations of notations. The categories of notations are discussed in Section 4.2.

Section 4.3 discusses *notational styles*, which are representations of notations in higher-order logic. These representations capture the structure of the original notation using keywords. The keywords take arguments of particular types that can be matched to the type signatures of the categories of notations. These matches, which we call *join points*, indicate how notations can be linked together to create a specification. Part of a specification written in one notational style can be viewed as a "lego-like" building block, which fits together with other parts of the specifications based on the interfaces provided by the join points. These categories provide a framework to try new combinations of notations.

The categories of notations are *lifted* in that they map configurations to meanings. Configuration lifting is used to express dynamic behaviour. Lifting frees the user from having to indicate explicitly whether a reference to a name is the value of the name in the previous configuration or in the next configuration. Lifting is discussed in Section 4.4.

A brief introduction to the heating system example is found in Section 4.5.

Next we present examples of notational styles of specifications currently defined in the framework. These notational styles are

- S+: higher-order logic which serves as both the base formalism for the framework and a notation that can be used for specification (Section 4.6)
- Lifted S+: a subset of S+ for expressions (Section 4.7)
- TableExpr: a tabular style of specifying expressions (Section 4.8)
- CoreEvent: a core set of event keywords (Section 4.9)
- CoreAction: a core set of action keywords (Section 4.10)
- CoreSc: states and transitions as found in statecharts (Section 4.11)

The statecharts notation has some associated notations that can only be used if a statecharts model is used. These are

- ScExpr: expressions particular to statecharts (Section 4.12)
- ScEvent: events particular to statecharts (Section 4.13)
- CommAction: actions that extend the communication primitives of statecharts (Section 4.14)
- CommEvent: events that extend the communication primitives of statecharts (Section 4.15)

The CoreAction and CoreEvent notations are strongly based on actions and events found in the STATEMATE tool [HL$^+$90]. The reader is assumed to have some familiarity with many of these notations.

Keywords and their type signatures indicating how notational styles fit together are presented. An informal understanding of the meaning of the keywords is provided. The next chapter will formally present their semantics.

All notations are subject to the well-formedness constraints of type checking. The type signatures are stated in S+. Use of the S+ notation is indicated by `verbatim` font. Use of the textual representations of notations required no extensions to the parser and type checker for S+.

Section 4.16 briefly discusses how to specify assumptions about the environment.

4.1 Model-oriented notations

Model-oriented specifications denote a next configuration relation, i.e., they constrain the relationship between a configuration and its immediate successor(s). A configuration maps a set of named locations to values at a time[1]. It can be thought of as a "snapshot" of the dynamic behaviour of the system [i-L91]. Model-oriented notations can form a part of a model-oriented specification. The operational semantics for a model notation result in a relation between two configurations. If the relation is a function, then the specification is deterministic.

The framework declares `config` as an uninterpreted type representing configurations in higher-order logic. The sequence of two configurations is called a `step` and formalised by a pair of configurations. The first element of the pair is called the *previous* configuration of a step and the second element is the *next* configuration.

Model-oriented notations often provide special structures such as "states". The meaning of the notation defines the changes in these structures. In contrast, the specification must explicitly describe changes to the declared parts of the system. We call these user-declared parts *assignable names* to distinguish them from other names of the specification. Others have called them controlled quantities [PM95].

[1]For our purposes, it is convenient to interpret a configuration as a mapping from time to names to values. There is nothing inherent in this work that restricts a configuration to be a mapping from time. The study of modal logics examines the concept of interpretations of names in different worlds [Che80].

4.2 Semantic categories

Different notations are better at describing different parts of the specification. Component notations of many model-oriented notations can be categorised using the following:

an expression: a value in a configuration.

an event: an instantaneous occurrence, such as a change in the value of a Boolean expression between two configurations.

a model: a relation between two configurations that indicates moving from the first configuration to the second is a legal step in the system. It includes constraints on the notation-specific structures of the system, such as the states, as well as the assignable names.

an action: a constraint on the changes in the values of assignable names. An action describes not only the changes, but also what assignable names are being modified and what happens if the action is not taken. As an element of a model, this information allows the semantics to resolve race conditions where multiple actions affect the same name. It also allows the semantics to constrain assignable names to keep the value they had in the previous configuration if they are not explicitly changed by an action.

We refer to these categories as "semantic" categories drawing from terminology used for denotational semantics.

Requirements notations, such as SCR and RSML, provide notations to describe elements in these categories. For example, mode transition tables in SCR are models. Notational styles in the same category, such as mode transition tables and statecharts, both in the model category, can be thought of as notations describing the same kind of behaviour.

Table 4.1: Type signatures for semantic entities

Category	Type Signature
expression	(ty)`exp`
event	(ty)`event`
model	`step -> BOOL`
action	`action`

A notation that belongs to one category can rely on an element in another category to produce a specification. For example, the transitions of a statechart are labelled with events and actions.

Having identified this categorisation, the framework provides a flexible way to integrate elements of these different categories. Each category has a type signature associated with it as found in Table 4.1. A step is a pair of configurations and has the type `config # config`. A model is a function that takes an argument of type `step` and returns a truth value indicating if the argument is a legal step for the system. The types `exp`, `event` and `action` will be defined in Section 5.3. The category of expressions can be subcategorised by the type of the expression, such as Boolean or numeric. Events can also be specialised to a model and therefore are parameterised by a type.

In this dissertation, we mainly concentrate on composition of notations for a specification that uses one notational style in the semantic category of models. Specifications stated in multiple notational styles found in the model category can be combined using conjunction as in Zave and Jackson [ZJ93]. Abadi and Lamport [AL93] also advocate that composition of parallel components should be defined by logical conjunction for the sake of proving properties. Other work on combining different model-oriented notations, such as that of Pezzè and Young [PY97], concentrates on combining models.

4.3 Notational styles

Elements of the semantic categories are *notational styles* in the framework. A notational style is a packaged embedding of a notation. It includes one or more keywords. A *keyword* denotes a function that takes arguments of the type of one or more of the categories. It returns an element of the type of the category of the notational style it belongs to, or it returns an element that is used in building up a specification in the notational style. The types of the arguments to keywords indicate *join points* to notational styles found in other categories or the same category. In essence, the type signatures provide interface descriptions. By picking particular instantiations of the interfaces, a specification in a collection of notational styles is created. We restrict ourselves to specifications of finite structure, e.g., a fixed number of rows in a table.

A notational style in higher-order logic may not exactly match the syntax of the original notation (except in the case of higher-order logic as a notation itself). However, the style captures the structure of the original notation. To integrate notations, a common representation must be found. We have chosen text to represent the notations. An interface to a parser or graphical editor can be provided to output these textual representations. In previous work, we created an interface to the STATEMATE tool. In our current textual representation for statecharts, we have chosen to leave out information such as graphical position because this information is not necessary for determining the meaning of a statechart specification. A textual representation that includes this information could have been used to ensure the reversibility of the representation mapping.

Figure 4.1 describes the notational styles presented in this chapter. The names of notational styles label the boxes. Within the box, examples of keywords with the join points are illustrated. For example, the `EvCond` keyword of the CoreEvent notational style has join points to an event notation (ev) and an expression notation (ex). Triangular links indicate join points that are restricted to another particular notational style rather than a semantic category. For example, the `En` keyword of the ScEvent notational style for entering a state requires a statechart structure (sc), which is created in the CoreSc notational style.

These styles were chosen based on elements of existing requirements specification notations. Parnas [Par93a] has advocated predicate logic as a good notation for specifications. The use of the notations SCR, RSML, and Parnas tables on large size industrial projects has demonstrated that tables and state transition notations are intuitive for specifiers. Our experience in writing and working with these notational styles on the two major examples presented in Chapter 8 corroborates the experience of others indicating the usefulness of these notations.

4.4 Lifted notational styles

Many model-oriented notations hide the details of the multiple configurations from the user. The position of the reference to a name within the use of a keyword of the notation implicitly indicates whether it is a reference to the previous value of the name or the next value. For example, in $x := x + 1$, the x on the left-hand side of the assignment operator (:=) refers to the value of x in the next configuration and the reference to x on the right-hand side refers to its value in the previous configuration.

In some model-oriented notations, syntactical conventions are used to describe the values of names in different configurations. For example, in Z [Spi88] specifications the value of the name in the next configuration is the primed version of its name in the previous configuration. In Z, $x' = x + 1$ means the value of x in the next configuration equals the value of x in the previous configuration plus one. Schema operators produce a model from a set of Z schemas.

Our framework concentrates on notations (actions, events and expressions) that are lifted. A lifted notation depends on the concept of a configuration that maps names to

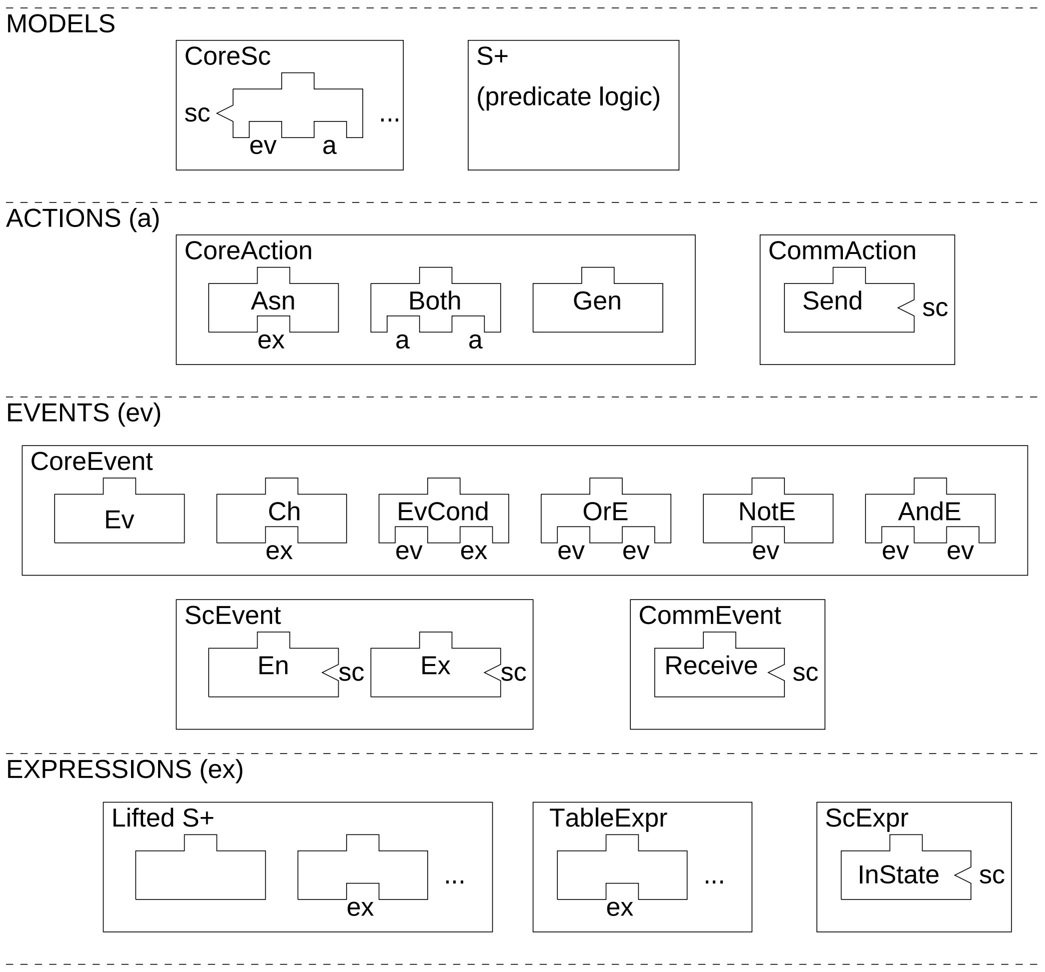

Figure 4.1: Example notations

values. Unlike Z, the same name is used multiple places in a specification but can refer to its value in different configurations depending on its position. For example, in the action `Asn x (x+1)`, the first instance of `x` refers to its value in the next configuration. The second instance of `x` refers to its value in the previous configuration.

Predicate logic does not have any built-in notions of changes over time. The concept of configurations must be formalised to define the semantics for the changes in the values of names. There are multiple ways that a configuration can be represented in an embedding of notations in logic. These ways are

- The names are functions of time [Gor85, PM95].
- The names are fields of a record. Accessing the field of different records constitutes accessing the value of the name at different times [Raj95].
- The configuration takes a name and returns a value for the name in that configuration as in denotational semantics where the configuration represents the "memory" [Gor79].

We chose a fourth approach[2] of making each name a mapping from a configuration to a value, which is conveniently done in higher-order logic where functions can have arguments that are functions. The user is freed from having to group explicitly all the names of the specification. In our approach, configurations are not necessarily associated with particular times, rather a specification denotes a relationship between previous and next configurations that may occur at any two sequential instants of time. In our approach to operational semantics, it is only necessary to refer to the previous value and the next value of a name.

We introduce the type `config`, and consistently use `cf` and `cf'` as the names of the parameters referring to the previous configuration and the next configuration in the

[2]While we do not claim this approach is novel, we are unaware of other work using exactly the same approach. A related approach is found in Staunstrup [Sta94].

definition of predicates constraining the relation between configurations. These identifiers are used in Chapter 7 to determine automatically references to previous and next values for analysis.

Notational styles in the model category may be lifted or not. S+ by itself is not a lifted notation and is included in the model category. Existing non-lifted notations, such as Z [Spi88], often do not consist of multiple notations, so consequently no categories like non-lifted events have been created. However, it would be possible to extend the framework with new categories should this situation arise. Models stated in lifted and non-lifted categories can be used together in one specification.

4.5 Heating system

A heating system loosely based on an example found in Booch [Boo91] is used to illustrate specification and analysis in the framework. This section provides a brief informal explanation of this system.

The heating system consists of a controller, a furnace and multiple rooms, which operate concurrently. We specify a system that has three rooms. Each room has a temperature gauge which determines the actual temperature, a sensor indicating if the room is occupied or not and a valve. The valve can be in one of three positions: closed, half open, and open. Another input is the set temperature for the room. The desired temperature is a function of the set temperature, the sensor indicating if the room is occupied or not and the living pattern. The living pattern indicates for any time whether a room is expected to be occupied, expected to be occupied within thirty minutes, or not expected to be occupied either now or in the next thirty minutes. Based on the desired temperature the room controls its valve position, waits to see if that has the desired effect, and if not, communicates a request for heat to the controller.

The controller ensures the furnace is on if any rooms are requesting heat and off if no room needs heat. The controller turns the furnace on and off. It also responds to the main heat switch turning the heating system on and off.

The furnace turns on and off based on instructions from the controller. It has a certain power-up time. Faults can occur in the furnace, which are communicated to the controller. After a fault the furnace cannot be restarted until a reset is received by the controller.

The specification of the heating system is used throughout this chapter for illustration. The complete specification can be found in Appendix A. It is written using the notational styles: CoreSc, TableExpr, CoreAction, CoreEvent, ScEvent, and higher-order logic (S+).

4.6 S+

The S+ notation is used both as a predicate logic notation in the model category and as the base formalism for embedding notational styles of specification. A subset of S+ is used as a notation for expressions. S+ is also used to declare the types and names of specifications.

S+ is a slight variant of the S notation [JDD94]. The motivation for developing S was the need for an "industrial strength", machine-readable formal notation that did not rely on special symbols (and therefore formatting tools for presentation). S+ uses common English words, such as "FORALL", as keywords. S+ is based on higher-order logic and has much in common with the "object language" of the HOL theorem proving system. S+ has the same semantics as the HOL object language so only its syntax will be presented in this dissertation. Like Z, and unlike the HOL object language, S+ includes constructs for the declaration and definition of types and constants. These constructs do not require extensions to the semantics of higher-order logic.

This section presents an overview of the S+ syntax. S+ is case-sensitive and the identifier spaces for types and constants are separate.

4.6.1 Types

This section describes type expressions, type declarations, type definitions, type abbreviations, and the built-in types of S+.

Type expressions

Type expressions are used in the definition of types and in the declaration and definition of constants. There are three kinds of type expressions. A reference to a type identifier is a type expression. A type expression of the form `ty1 -> ty2` describes expressions that are functions from elements of type `ty1` to elements of type `ty2`. The Cartesian product of types is stated using the syntax `ty1 # ty2`. These type operators are right associative.

Expressions in the S+ notation can be polymorphic, meaning they are "indifferent to the types of their arguments" [Han87]. A type checker infers the type of an expression and ensures that it is legal. For a polymorphic function, some elements of the type are unconstrained. The usual practise in tools supporting polymorphic notations is to accept expressions with some elements of the type unconstrained and record the expression's type using a form such as ":* -> **", to indicate that the domain and range of the function are unconstrained.

In S+, specifiers must use type parameters to declare explicitly parts of a type expression that are polymorphic[3]. Type parameters precede a declared or defined type or constant. For example the following constant definition states that the `identity` function takes one polymorphic argument:

```
(:ty)
identity (x:ty) := x;
```

[3]Type parameters are not an original idea in their distinctness from inferring the polymorphism. Gries and Gehani describe early work on explicit type parameters in programming languages [GG77].

The type parameter representing this polymorphism is `ty`. Type inference proceeds with the use of these type parameters as well-defined types within the context of this definition. Any unconstrained types remaining after type checking render the S+ definition invalid. The type of a constant can only contain declared and defined types and explicitly declared type parameters.

Our choice to force the user to declare explicitly the polymorphic elements of a definition may be seen as cumbersome. However, this choice is motivated by our belief that explicit recognition of polymorphism can uncover some errors in a specification through type checking.

Type declarations

A type declaration introduces one or more new uninterpreted types. An example of a type declaration is:

```
: Books, Names;
```

Nothing is said about the elements that are members of these types except that some elements exist (i.e., the type is not empty). A type declaration may be used in an S+ specification the same way that the declaration of one or more "basic types" may be used in a Z specification to achieve a desired level of abstraction [Spi88].

Type definitions

A type definition introduces a new type and defines its elements using constructors. A constructor is a function that maps zero or more objects to a unique object of the new type. For example, the possible valve positions in the heating system are specified using the type definition:

```
: Valve_Pos := OPEN | HALF | CLOSED ;
```

The type name is `ValvePos` and its constructors are `OPEN`, `HALF` and `CLOSED`. The constructors must be unique. The other type definitions found in the heating system specification are:

```
: Room := KITCHEN | LIVING_ROOM | BEDROOM;
: Behaviour := NOT_OCCUPIED | EXPECT_SOON | EXPECT_NOW ;
```

A type definition may be recursive – that is, the new type may appear on the right-hand side of the type definition provided that at least one of the branches does not contain a reference to the new type.

Type definitions can have type parameters. For example, the definition for the type `list` is:

```
: (ty)list := CONS :ty :(ty)list | NIL ;
```

Type abbreviations

Type abbreviations are short forms or aliases for type expressions. In the framework, they are commonly used to create lifted types for names in a specification. For example, the name `valve_Pos` is used as an abbreviation for the type expression that lifts the `Valve_Pos` type:

```
: valve_Pos == config -> Valve_Pos;
```

The other example of a type abbreviation found in the heating system is:

```
: behaviour == config -> Behaviour;
```

Built-in types

S+ includes three built-in type declarations, two type definitions, and three built-in type abbreviations. The built-in types are:

```
:config;

:BOOL := T | F;

:bool == config -> BOOL;

:NUM;

:num == config -> NUM;

:STRING;

:string == config -> STRING;

:(ty)list := CONS :ty :(ty)list | NIL;
```

The `NUM` type represents the set of rational numbers[4]. The type `BOOL` is the set of truth values. The types `num` and `bool` are abbreviations for the lifted Boolean and numeric types.

4.6.2 Constants

Constants are identifiers with types and possibly definitions. The type of a constant can be polymorphic but the constant cannot be overloaded (i.e., have more than one definition for the same identifier depending on the type of its arguments).

The name of a constant can be a string of characters without blanks, or any string of characters including blanks enclosed in double quotes. Quoted strings can be used to include characters that have special significance for naming conventions, making it possible to maintain a close correspondence between references to a constant in informal documentation and its formalisation.

[4]As S+ is just a notation without logical reasoning support, “num” can be chosen to be any kind of numbers. The parser accepts decimal numbers. Our current implementation of symbolic functional evaluation truncates numbers to treat them as C integers. Errors such as dividing by zero would cause arithmetic exceptions in our current implementation.

This section describes constant declarations, constant definitions and the built-in constants of S+.

Constant declarations

Constant declarations introduce new constants of particular types. Since no meaning is defined for the constant it is called uninterpreted.

For example each room has a valve position and an actual temperature:

```
valvePos : Room -> valve_Pos;
actualTemp : Room -> num;
```

Because a type abbreviation is used to create the lifted type `valve_Pos` as shown on page 75, an expression such as `valvePos LIVING_ROOM` is a lifted expression.

Constant declarations are used to introduce the names of a specification. Other names in the heating system are:

```
requestHeat : Room -> bool;
livingPattern : Room -> behaviour;
occupied : Room -> bool;
setTemp : Room -> num;
warmUpTime, coolDownTime : num;
furnaceStartupTime: num;
```

The names `warmUpTime` and `coolDownTime` represent how long a room waits before checking to see if a change in the valve position has the desired effect on the temperature of the room.

A name that is an array can be specified using a function from a type for the indices to a type for the values of the array.

A current deficiency in the S+ notation is the lack of an explicit record type construct. However, a useful abstraction for records is provided by using an uninterpreted type for the record together with a special syntax for function application called the "dot notation", which is similar to accessing fields in a record (Section 4.6.3). This approach has the benefit of being only a partial specification of the fields of a record and a more abstract representation of a record.

Constant declarations can also be used to introduce operations in a specification without providing any details about the behaviour of that operation. For example, `+` (addition), is a built-in declared constant. For the types of automated analysis currently implemented in the framework `+` is considered an uninterpreted constant.

Constant definitions

A constant definition is equivalent to the declaration of a new constant and the introduction of a definitional axiom for the new constant. The right-hand side of the definition must be an expression (Section 4.6.3). Constants can be recursively defined. Examples of constant definitions in the heating system are:

```
sT := setTemp;

aT := actualTemp;

dT := desiredTemp;

tooCold (i:Room) := ((dT i) - aT i) > C 2;

tooHot (i:Room) := ((aT i) - dT i) > C 2;

vOpen (i:Room) := valvePos i = C OPEN;

vClosed (i:Room) := valvePos i = C CLOSED;

roomNeedsHeat := exists (i:Room). requestHeat i;

noRoomsNeedHeat := NOT(roomNeedsHeat);
```

The definitions of `sT` and `aT` provide short-hands for the names `setTemp` and `actualTemp`. The definition of `dT` provides a short-hand for the function `desiredTemp`, which is defined using a decision table in Section 4.8. The definitions, `tooCold`, `tooHot`, `vOpen`, and `vClosed`, specify conditions for each room. For example, the condition `tooCold` is true when the difference between the desired temperature (`dT`) and the actual temperature (`aT`) is greater than 2. `C` is a built-in constant that acts as an "lift" operator. The expression `C 2` has the value `2` in all configurations. The definitions `roomNeedsHeat` and `noRoomsNeedHeat` describe conditions that depend on the status of all rooms in the heating system.

Constants can be infix by using the syntax "`(_`" and "`_)`" on either side of the constant when it is declared or defined. A function can be curried or uncurried in the way it takes its arguments. Curried functions take their arguments one at a time, as in `f a b`. Uncurried take their arguments in tuples as in, `f (a,b)`.

Constants can be defined using pattern matching definitions. For example, the following defines the length operation on lists:

```
(:ty)
length (NIL) := 0 |
length (CONS (a:ty) b) := 1 PLUS (length b);
```

The first argument must be an element of a type defined by a type definition. Constructors of this type split the definition into different cases. The operator `PLUS` is a non-lifted built-in constant.

Pattern-matching definitions do not have to include cases for all constructors of the type. If some constructors are missing, the definition partially specifies a function. For analysis, the unspecified value is treated similarly to an uninterpreted constant (Section 6.4.2).

Built-in constants

The built-in constants of S+ are found in Appendix B. Some of the commonly used ones are presented in Figure 4.2.

```
(:ty) LET (x:ty) := x;

(:ty) FORALL : (ty -> BOOL) -> BOOL;
(:ty) EXISTS : (ty -> BOOL) -> BOOL;

(_ /\ _) : BOOL -> BOOL -> BOOL;
(_ \/ _) : BOOL -> BOOL -> BOOL;
~ : BOOL -> BOOL;

a (_ AND _) b (cf:config) := (a cf) /\ (b cf);
a (_ OR _) b (cf:config) := (a cf) \/ (b cf);
NOT a (cf:config) := ~(a cf);

(:ty)
COND (T) (a:ty) b := a |
COND (F) a b := b;

a (_ ==> _) b := (NOT a) OR b;

(:ty) C (x:ty) (cf:config) := x;

(:ty1,:ty2) FST : (ty1#ty2)->ty1;
(:ty1,:ty2) SND : (ty1#ty2)->ty2;

(_ PLUS  _) : NUM->NUM->NUM;
a (_ + _) b (cf:config) := (a cf) PLUS (b cf);

(:ty) (_ EQ _) : ty->ty->BOOL;
(:ty) (a:config->ty) (_ = _) b cf := (a cf) EQ (b cf);

(_ GREATER_THAN _) : NUM -> NUM -> BOOL;
a (_ > _) b (cf:config) := (a cf) GREATER_THAN (b cf);
```

Figure 4.2: Commonly used built-in constants of S+

By convention, built-in constants are identifiers consisting of all capital letters or symbols. There are lifted and unlifted versions of most of the operators, with the common name, such as `+`, used for the lifted operator.

Join points exist between operators such as `+` and other notations. These lifted arithmetic operators can take expressions of the appropriate type in any notational style of expression.

4.6.3 Expressions

Expressions appear on the right-hand side of a constant definition in a specification. There are four types of S+ expressions: constants, variables, applications, and lambda abstractions. The set of constants can be further refined to differentiate between constructors, defined constants, and uninterpreted constants.

Any constants used in an expression must have been previously declared or be the constant being defined (in recursive definitions). All variables must be bound as parameters in a definition or as the variable of a lambda abstraction. Variables cannot have the same name and a different type within the same scope.

Function application is stated as `f a` for the function `f` applied to the argument `a`. Lambda abstractions are stated as `\x.E` (where `E` is an expression) or as `function x. E`. Examples of expressions used in the heating system specification are:

```
(\x. x = C EXPECT_NOW)
(\x. C -5 <= x AND x < C -2);
```

Expressions can also be written using post-fix function application, called the *dot notation*, such as:

```
KITCHEN.actualTemp
```

This expression is semantically equivalent to the prefix application of the function

Table 4.2: S+ syntactic sugar for expressions

Syntactic Sugar	Expression
`[` exp_1 `;` exp_2 `; ... ]`	`CONS` exp_1 `(CONS` exp_2 `... NIL)`
`let` par `:=` exp_1 `in` exp_2	`LET ((` \\par `.` exp_2`)` exp_1`)`
`forall` par_1 `...` par_n `.` exp	`\cf. FORALL (`\\$par_1$ `. ...` `(FORALL (`\\$par_n$ `.` exp `cf)) ... )`
`exists` par_1 `...` par_n `.` exp	`\cf. EXISTS (`\\$par_1$ `. ...` `(EXISTS(`\\$par_n$ `.` exp `cf)) ... )`
`if` exp_1 `then` exp_2 `else` exp_3	`\cf. COND (`exp_1 `cf) (`exp_2 `cf) (`exp_3 `cf)`
`if` exp_1 `then` exp_2	exp_1 `==>` exp_2

`actualTemp` to the expression `KITCHEN`, i.e., `actualTemp KITCHEN`. Post-fix function application is useful for specifiers accustomed to the idea of accessing a field of a record.

S+ includes syntactic sugar for particular expressions. Table 4.2 describes the meaning of this special syntax.

4.7 Lifted S+ expressions

Lifted S+ expressions are those that have type `config->ty` where `ty` is a type parameter. The expression category has the type signature `(ty)exp` which is an abbreviation for `config->ty`, i.e.,

```
: (ty)exp == config -> ty;
```

Thus the type `num` is `(NUM)exp` and `bool` is `(BOOL)exp`.

The presence of the type `(ty)exp` in type signatures for any keywords of a notational style indicates a join point to any notation producing lifted expressions of the correct type.

4.8 TableExpr style of expressions

The popularity of table-based notations, such as those of Parnas, RSML and SCR, indicates that the use of tables has many advantages as a specification technique. Their conciseness, modularity, and similarity to other notations used by engineers makes them a good choice as a notation for non-experts. Previous successful efforts of using tables provide a good precedent for the readability of a tabular style of specification. These efforts include the AND/OR tables of the TCAS II project [LHHR94] and the SCR notation used in the A-7 aircraft Operational Flight Program [Hen80]. Our own interaction with domain experts at Hughes International Airspace Management Systems independently confirms their readability [DJP97a].

Tables work well for specifying combinations of conditions that produce different outcomes. The notational style of tables used as an example in the framework is a slight variant of AND/OR tables.

An AND/OR table consists of a series of rows labelled by Boolean expressions (conditions). The columns to the right of the label contain "T" for true, "F" for false, or "." for "don't care". A row entry is meant to represent the case where the condition found in the label is true or false. A "don't care" value means that the entry could contain either true or false. An AND/OR table specifies a predicate that is true if the conjunction of the entries in any column results in true.

Our notational style of tables is an extension of AND/OR tables. TableExpr includes keywords for specifying functions and predicates. An example of a table in TableExpr is found in Table 4.3. AND/OR tables can only specify predicates. In our function tables, a final row is added that specifies a return value for the function if the set of conditions represented by the column is satisfied. A final column labelled the "Default" column can be specified. It contains only an entry in the final row representing the value of the function if none of the sets of conditions of the columns are satisfied. If no default column is specified and the disjunction of the column conditions is not a tautology, the table denotes a function where the value for combinations of conditions not covered in a column is unspecified. In the semantics, the unspecified value is the uninterpreted constant `UNKNOWN` (Section 5.5).

We also extend the possible entries in a row of a table to put related entries in one row. This is not possible in AND/OR tables. It is advantageous both for readability and for analysis. Section 7.3.1 describes how the related entries in a row provides useful information to analysis.

Row entries are predicates that take the label of the row as an argument and return true or false. Inspired by Parnas tables, which allow related conditions to be placed in the same row, we extend the possible row entries such that related predicates that apply to the same row label can be used in the same row. For example, the predicates in a row may partition the range of values for a number into separate cases. This mix of AND/OR tables and Parnas tables proved useful in the examples used to illustrate the dissertation. A complete introduction to our tabular notation can be found in Day, Joyce and Pelletier [DJP97b].

Tables 4.3 and 4.4 are graphical representations of the two function tables used in specifying the heating system. Table 4.3 is a function table resulting in an expression for the desired temperature in a room. There are three possible values for the desired temperature, as specified in the last row of the table, namely `sT i` (the set temperature of the room), `sT i - C 5` (five degrees below the set temperature), and `sT i - C 10` (10 degrees below the set temperature). The choice of one of these values depends on the value of the sensor indicating whether the room is occupied or not (`occupied i`, which has Boolean value) and the living pattern of the room. The living pattern indicates that if the room is not expected to be occupied, is expected to be occupied now, or is expected to be occupied soon (i.e., within thirty minutes). Columns specify combinations of values for the living pattern and occupied sensors, that result in certain values for the desired temperature.

Row entries are predicates on the value of the row label. These predicates have the type signature:

```
: (ty) rowentry == :(ty)exp->bool;
```

In a table, an underscore in a row entry indicates where to substitute the row label into the expression.

TableExpr includes three keywords for describing row entries. The keyword `True` means the value of the row label is true, and `False` means the value of the row label is false. The keyword `Dc` means "don't care", which is represented graphically by a ".". These keywords have the following type signatures:

```
True : bool -> bool;
False : bool -> bool;
(:ty)Dc : (ty)exp -> bool;
```

These keywords all match the type signature for row entries. `True` and `False` match the more specific `(BOOL)rowentry` type.

The expression of the row label can be any lifted expression:

```
: (ty) rowlabel == (ty)exp;
```

The type represented by the parameter `ty` must be the same in the row label and all row entries.

Our tabular style includes two keywords that are useful for specifying conditions over lists of expressions. These keywords can be used in row labels and are finite lifted versions of the quantifiers "forall" and "exists":

```
(:ty) AllOf : ((ty)exp)list -> ((ty)exp -> bool) -> bool;
(:ty) AtLeastOneOf : ((ty)exp)list -> ((ty)exp -> bool) -> bool;
```

Table 4.3: Desired temperature of a room

				Default
`occupied i`	`True`	`False`	`False`	
`livingPattern i`	`.`	`_ = C EXPECT_NOW`	`_ = C EXPECT_SOON`	
`desiredTemp i`	`sT i`	`sT i`	`sT i - C 5`	`sT i - C 10`

Table 4.4: Valve position

dT i - aT i	_ < C -5	C -5 <= _ AND _ < C -2	C -5 <= _ AND _ < C -2	C -2 <= _ AND _ <= C 2	C 2 < _ AND _ <= C 5	C 2 < _ AND _ <= C 5	C 5 < _
valvePos i	.	_ = C OPEN	_ = C HALF	.	_ = C CLOSED	_ = C HALF	.
nextVp i	C CLOSED	C HALF	C CLOSED	valvePos i	C HALF	C OPEN	C OPEN

The first argument to these row labels is a list of expressions. The second argument is a lifted predicate that takes items in the list and returns a Boolean value. The expression denoted by the row label is the conjunction or disjunction respectively of the predicate applied to the items in the list.

The type `row` is a list of expressions of type Boolean that can be combined with other rows to form conditions for the columns of the table:

```
: row == (bool)list;
```

The keyword `Row` describes the arrangement of an expression giving the row label followed by a list of row entries. It has the type:

```
(:ty) Row :(ty)rowlabel -> ((ty)rowentry)list -> row;
```

The keywords `Table` and `PredicateTable` capture the structure of the rows and columns of tables. Both kinds of tables contain a list of rows. In a function table (`Table`), this list is followed by a list of return values for the function. These keywords have the following types:

```
PredicateTable : (row)list -> bool;
(:ty) Table :(row)list -> ((ty)exp)list -> (ty)exp;
```

Function tables are expressions of type `(ty)exp` whereas predicate tables are expressions of type `bool`. A well-formed predicate table is one that has rows of equal length. A well-formed function table has rows of equal length and the list of return values must be equal to the length of each row, or greater than the length of each row by one (the default column).

Figures 4.3 and 4.4 present the textual representations of the tables specifying the desired temperature of a room and the next valve position. The placeholders represented graphically as "_" appear as lambda abstractions. We developed a simple tool

```
desiredTemp (i:Room) :=
Table
[ Row (occupied i)  [ True; False ; False ];
  Row (livingPattern i)
      [  Dc;  (\x.x = C EXPECT_NOW) ; (\x.x = C EXPECT_SOON)]
]
[ sT i; sT i; sT i - C 5; sT i - C 10 ];
```

Figure 4.3: Textual representation of desired temperature table

```
nextVp i :=
Table
[ Row (dT i - aT i)
      [ (\x.x < C -5);
        (\x. C -5 <= x AND x < C -2);
        (\x. C -5 <= x AND x < C -2);
        (\x. C -2 <= x AND x <= C 2);
        (\x. C 2 < x AND x <= C 5);
        (\x. C 2 < x AND x <= C 5) ;
        (\x. C 5 < x)];
 Row (valvePos i)
      [ Dc ;
        (\x. x = C OPEN) ;
        (\x. x = C HALF) ;
        Dc ;
        (\x. x = C CLOSED) ;
        (\x. x = C HALF) ;
        Dc] ]
[ C CLOSED; C HALF; C CLOSED; valvePos i; C HALF; C OPEN; C OPEN];
```

Figure 4.4: Textual representation of valve position table

```
if (True(occupied i)) then
     sT i
else if (False(occupied i) AND
            (livingPattern i = C EXPECT_NOW)) then
     stT i
else if (False(occupied i) AND
            (livingPattern i = C EXPECT_SOON)) then
     sT i - C 5
else
     sT i - C 10;
```

Figure 4.5: Meaning of table specifying desired temperature

to convert our textual representation into a tabular, cross-referenced representation in HTML [DJP97a].

A semantically equivalent representation in S+ of the function specified in Figure 4.3 is found in Figure 4.5. There is an order of precedence to the columns in a function table. A function can have only one return value for each set of arguments. In the cases where two columns both have the value true, this order of precedence determines the result of the function.

A predicate table is similar to an AND/OR table in that the return expression for every column is true and the default value is false. Therefore, the last row is not needed. The order of precedence of columns does not matter in a predicate table.

In summary, this tabular style includes keywords for creating function and predicate tables. It also includes a keyword for creating a row, two keywords for creating row labels, and three keywords for creating row entries. Row labels, row entries, and return values for a function table are join points to any notational style in the expression category.

4.9 CoreEvent style of events

An event is an instantaneous occurrence, such as a change in the value of a Boolean expression between two configurations. An example of an event is a button being pushed.

In this section, we present the keywords of the CoreEvent notation, which each return an element of the type `event`. The keywords are: `Ev` (primitive event), `Ch` (change in a condition), `EvCond` (event in combination with a condition), `Tm` (timeout event), `TmB` (finite delay timeout), `AndE`, `OrE`, `NotE`, and `NonEvent`. As shown below, these keywords are all parameterised by a type that becomes specific when the CoreEvent notation is used with a particular notational style of model. Some events take other events as arguments. Events that are arguments of another event are called *nested* events.

The CoreEvent notation will be used in combination with the CoreSc model notation in examples presented in Section 4.11. However, in our framework CoreEvent could also be used in new combinations of notations. For example, if the SCR language is added to the framework, CoreEvents could be used to specify events in a mode transition table.

A primitive event is a name that the specifier associates with an occurrence in the environment. The phenomenon being observed can be specified as a Boolean constant. The following type, specified by a type abbreviation, is used for these Boolean constants:

```
: simpleEvent == bool;
```

The primitive event associated with the constant occurs whenever the value of the name of the constant is true in the next configuration. The keyword `Ev` is used to specify primitive events:

```
(:ty) Ev :simpleEvent -> (ty)event;
```

A well-formedness constraint is that the first argument to this keyword evaluates to a constant. External primitive events that can occur in the heating system are:

```
heatSwitchOn, heatSwitchOff, userReset: simpleEvent;
```

The event of one of these primitives occurring is:

```
Ev heatSwitchOn
```

A change in a condition is an event. The value of the condition can change from true to false or false to true. The keyword Ch is used to specify the event of a change in a condition:

```
(:ty) Ch :bool -> (ty)event;
```

The argument to this keyword is a join point to any expression notation. The argument must be of Boolean type.

An event can also be the occurrence of an event when a particular condition is true. The condition is an expression of type bool, so the keyword EvCond has the type signature:

```
(:ty) EvCond : (ty)event -> bool -> (ty)event;
```

The first argument, which has the type event, is a join point to any notational style of event. The second argument, a condition, is a join point to any expression notation. For example, in the heating system, the event of a room being too cold is used. This event is specified as:

```
EvCond NonEvent (tooCold i)
```

The nested event NonEvent is described later in this section.

An event occurs in a particular step. It can be useful to have an event that occurs when a certain number of steps have passed since another event occurred. This event is called a timeout event and is specified using the keyword Tm which takes an event and an expression that returns a number as parameters:

```
(:ty) Tm : (ty)event -> num -> (ty)event;
```

The delay for a Tm event could be arbitrarily large since it is of type num. To prove particular properties of a system, we need to know that this delay is finite. The specification can include this detail explicitly by representing the value of the timer as a vector of Boolean values (i.e., a bit vector) using the keyword `TmB`. `TmB` is very similar to the `Tm` construct but its second argument (the timeout delay) is a bit vector:

```
(:ty) TmB : (ty)event -> (bool)list -> (ty)event;
```

A well-formedness constraint is that the bit vector is of finite length. This event has a more liberal meaning than the `Tm` event, i.e., it admits fewer behaviours to the specification. Users can selectively choose to provide a more detailed specification to particular timeout events as they attempt to prove different properties. This choice is a simple way of giving the user control over the size of the configuration space that needs to be explored in analysis.

An event can depend on the combination of other events using the keywords `AndE` and `OrE`:

```
(:ty) AndE : (ty)event -> (ty)event -> (ty)event;
(:ty) OrE : (ty)event -> (ty)event -> (ty)event;
```

At times it is useful to specify the event of another event not occurring. This event is specified by the keyword:

```
(:ty) NotE : (ty)event -> (ty)event;
```

It occurs in the steps that the event passed as the argument to the keyword does not occur.

Transitions that do not depend on any event must have some form of label. For these transitions, the keyword `NonEvent` is used:

```
(:ty) NonEvent: (ty)event;
```

This event always occurs in a step.

4.10 CoreAction style of actions

Action notations describe how the values of names in a specification change as a result of a step. Actions are intended to be components of a model and can be used with a variety of transition-based notations so it is worthwhile to consider action notations separately from model notations. CoreAction has the keywords `Asn`, `Both`, `Gen`, and `NoAction`.

The intent of an action is to modify the value of a name. The simplest form of action is the assignment statement. The type signature for this keyword is:

```
(:ty) Asn : (ty)exp -> (ty)exp -> action
```

The second argument can be any expression and is therefore a join point to other expression notations. In the heating system, there are the following uses of the assignment action:

```
Asn (requestHeat i) (C T)

Asn (requestHeat i) (C F)

Asn (valvePos i) (nextVp i)
```

The assignments `Asn (requestHeat i) (C T)` and `Asn (requestHeat i) (C F)` change the value of the request heat flag for a room. This flag is set by a room and read by the controller. The assignment `Asn (valvePos i) (nextVp i)` sets the valve position for a room to the value specified by the expression `nextVp i`, which is specified using a function table. The short-hand `adjValve i` is assigned to this action:

```
adjValve i := Asn (valvePos i) (nextVp i);
```

This action will be used as the action on a statechart transition for the heating system example demonstrating the ease with which a new combination of notations can be created in our framework.

Using predicate logic, we provide shortened aliases for these actions that can be used in the heating system specification, such as:

```
rH i := Asn (requestHeat i) (C T);
```

The first argument to Asn is the name to be assigned a value. Not all expressions are valid for this first argument. A well-formedness constraint restricts the first argument to expressions that evaluate to an uninterpreted constant applied to zero or more constructors. This constraint ensures the first argument to Asn refers to a distinct name. Uninterpreted functions describing fields in a record cannot appear as the first argument of an assignment statement. Record fields described by uninterpreted functions cannot be updated using the Asn action. An entire record could be updated by assigning a value to an uninterpreted constant. The new value would have to be represented by another uninterpreted contant because the values of an uninterpreted type are unknown. (This limitation provides motivation for adding a proper record construct to a future version of S+.)

The keyword Both is a list of actions. The choice of whether there is an order to the evaluation of these actions is determined by the notational style that uses the actions. Section 5.8.3 describes how statecharts handle an action containing a list.

```
Both: action -> action  -> action;
```

The two arguments to this keyword are join points to any notational style of action.

Besides representing observations of the environment, primitive events can be used to describe internal events. For example, if after some external event a sequence of operations is to be carried out, the next operation in the sequence could be triggered by an internal event once the previous one has been completed. CoreAction includes the keyword Gen, which assigns a name of Boolean type the value true. The type signature for Gen is:

```
Gen : simpleEvent -> action;
```

In the heating system there are four internal events that coordinate activity between the controller and the furnace:

```
furnaceRunning, furnaceReset, activate, deactivate: simpleEvent;
```

An example of an action that causes one of these events to occur is:

```
Gen activate
```

The keyword `NoAction` specifies no changes in the system and has the type:

```
NoAction : action;
```

CoreAction is a very simple action notation. For some specifications, it could be worthwhile to extend the framework to include action notations that have more of a programming language flavour, such as process specifications in structured analysis [DeM79].

4.11 CoreSc style of models

A statechart [HPSS87] is a hierarchical and concurrent state machine with transitions to move the system between states. This graphical notation is often used for specifying reactive systems. For simplicity we will continue to refer to our notational style based on this notation as statecharts, although it should be interpreted as a textual representation of a slight variant of the notation originally described by Harel. For more detailed presentations of statecharts and its variants, the reader is referred to works by Harel et al., Pnueli and Shalev, and von der Beeck. [HPSS87, Har87, Har88, PS91, vdB94].

Figure 4.6 is a statechart specification of the controller component of the heating system. There are three types of states in a statechart. An OR-state consists of substates. It is usually graphically depicted as a rounded box with boxes inside of it representing its substates. If the system is in an OR-state, then it is also in exactly one of the substates of an OR-state. In Figure 4.6, `CONTROLLER_ON` is an OR-state. The substate of an OR-state indicated using an arrow with no source state is called the default state. When a state is not decomposed into substates, it is called a basic state. `OFF` is a basic state in the controller statechart.

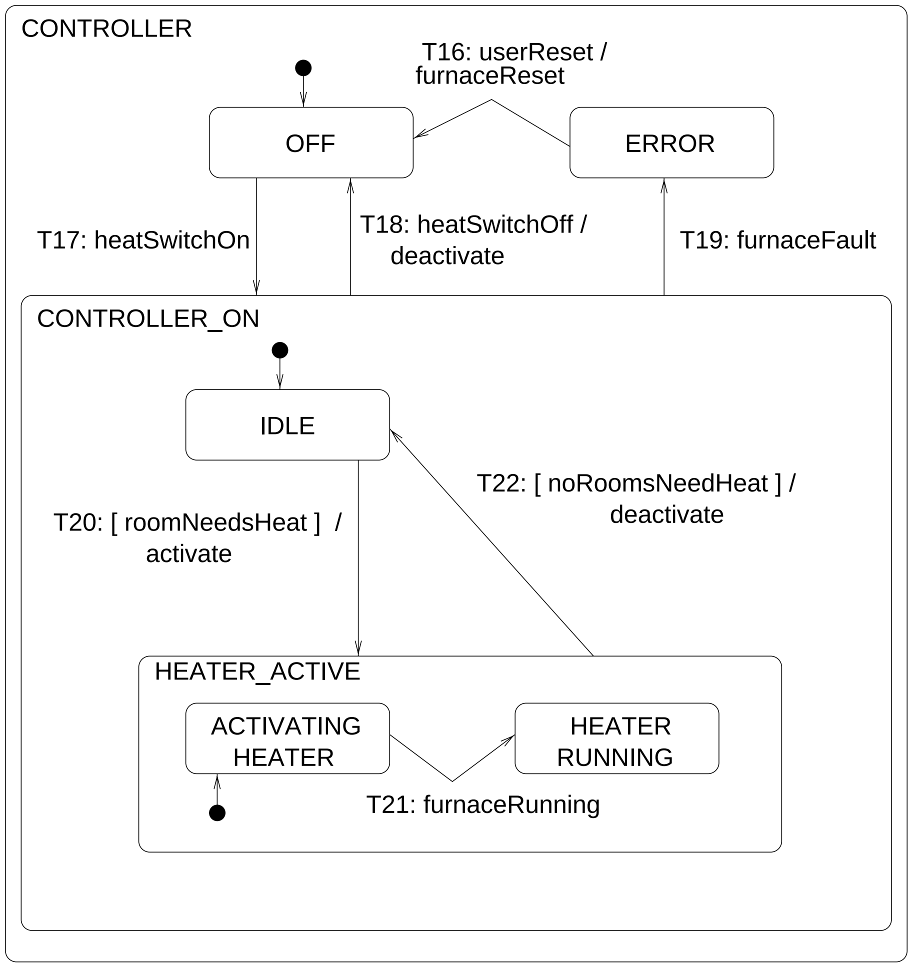

Figure 4.6: Controller statechart

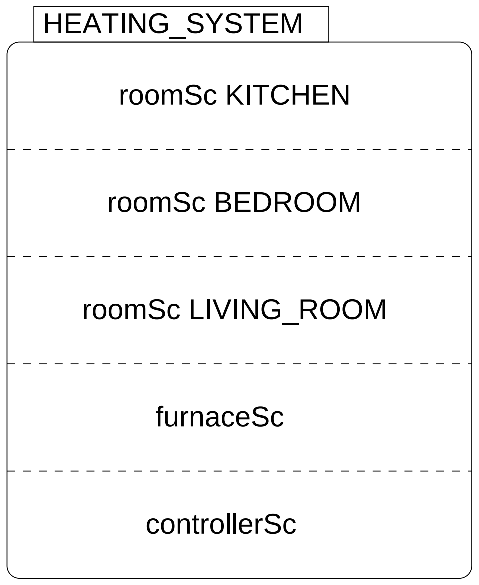

Figure 4.7: Heating system top-level specification

Figure 4.7 is the top-level specification of the heating system showing its five concurrent components (three rooms, the furnace, and the controller). The state `HEATING_SYSTEM` is an AND-state, which represents concurrent operation. It is represented as a rounded box divided by dashed lines. Each area of the AND-state is labelled with the name of its component statechart (therefore the controller component is labelled `controllerSc` rather than by its state name of `CONTROLLER`). If the system is in an AND-state, then it is also in every component state of the AND-state.

Every state must have a unique name. The state names for a statechart are defined by the specifier as type constructors. State names can be parameterised by other types. For example, the rooms of the heating system all have the same behaviour. The names of their states are parameterised by the name of the room, as in:

```
: stateName :=
  ...
  ROOM :Room |
  NO_HEAT_REQUESTED :Room |
  IDLE_NO_HEAT :Room |
  WAIT_FOR_HEAT :Room |
  HEAT_REQUESTED :Room |
  IDLE_HEATING :Room |
  WAIT_FOR_COOL :Room |
  ...
```

States are connected by transitions. A transition consists of a unique identifier, a source state, an event, an action, and a destination state. Just as with states, the unique identifiers labelling transitions are specified by type constructors that can be parameterised. The keywords for statecharts are parameterised by the types of state names and transition names.

Intuitively a step in a statechart is the choice of a set of non-conflicting transitions whose source states are part of the previous configuration and whose events have just occurred. Two transitions conflict if they both originate from any level of hierarchy within the same OR-state. The next configuration after the step consists of: 1) the destination states of the chosen transitions (if a transition's destination state is an OR-state, the destination state is the OR-state's default state); 2) the effects of the actions of those transitions; and 3) an updated version of the status of the events of concern to the system. The next chapter will describe in detail the choice of transitions to take and the meaning of actions in the presence of race conditions (multiple transitions modifying the same name). Broadcast communication is used in a statechart model where all parts of the system are aware of all events that occur and the value of all expressions.

Rather than describing a particular notation for the events and actions of a statechart, we instead consider these elements of a transition join points to other categories. Anything that has the return type `event` or `action` can be used where appropriate in a statechart. Therefore, this notational style is called "core statecharts". An example of a hardware description language that integrates essentially core statecharts with another notation is SpecCharts [NVG92], which augment statecharts with VHDL.

The type abbreviation `trans` describes the type of transitions:

```
: (stateName, transName) trans ==
  transName #
  stateName #
  (transName)event #
  action #
  stateName ;
```

The elements of the tuple are the unique transition identifier, the source state, the triggering event (customised for statecharts), the resulting action, and the destination state. The event and action elements are join points to event and action notational styles respectively. For example, the transition connecting states `IDLE` and `HEATER_ACTIVE` in Figure 4.6 is specified as:

```
(T20, IDLE, EvCond NonEvent roomNeedsHeat, Gen activate, HEATER_ACTIVE)
```

In the diagram, the customary notation of ev [cond] / act was used to describe events and actions labelling transitions.

The keywords `OrState`, `AndState`, and `BasicState` are used to describe the hierarchy of states. An element of type `(stateName,transName)sc` specifies the *statechart structure*. This type will be defined in the next chapter. The type signatures for the keyword `OrState` is:

```
(:stateName,:transName) OrState
  :stateName ->                           /* state name */
   stateName ->                           /* default state */
   ((stateName,transName)sc)list ->       /* substatecharts */
   ((stateName,transName)trans)list ->  /* transition list */
   (stateName,transName)sc;
```

The type signatures for the keyword `AndState` is:

```
(:stateName, :transName) AndState
  :stateName ->                           /* state name */
   ((stateName,transName)sc)list ->       /* substatecharts */
   (stateName,transName)sc;
```

The type signatures for the keyword `BasicState` is:

```
(:stateName, :transName) BasicState
  :stateName ->                           /* state name */
   (stateName,transName)sc;
```

The keyword `OrState` takes arguments that are the name of the state, the name of its default substate, a list of a statecharts describing its substates, and a list of transitions. The default state of an OR-state must be in the list of substates of the OR-state. The keyword `AndState` takes the name of the state and a list of statecharts describing its substates. An immediate substate of an AND-state cannot be a source or destination state for any transition. A basic state has only a state name. Figure 4.8 is the textual representation in our statecharts style of the controller specified in Figure 4.6. This example demonstrates how expressions in higher-order logic are used in events, which are used to trigger transitions of a statecharts. Thus, statecharts are used in combination with higher-order logic expressions that include a quantifier (e.g., `roomNeedsHeat` defined on page 78). The room

```
controllerSc :=
  OrState CONTROLLER OFF
  [BasicState OFF;
   BasicState ERROR;
   OrState CONTROLLER_ON IDLE
      [BasicState IDLE;
       OrState HEATER_ACTIVE ACTIVATING_HEATER
       [BasicState ACTIVATING_HEATER;
        BasicState HEATER_RUNNING]
       [(T21, ACTIVATING_HEATER, Ev furnaceRunning,
              NoAction, HEATER_RUNNING)] ]
      [(T20, IDLE, EvCond NonEvent (roomNeedsHeat),
              Gen activate, HEATER_ACTIVE);
       (T22, HEATER_ACTIVE, EvCond NonEvent (noRoomsNeedHeat),
              Gen deactivate, IDLE ) ] ]
   [(T16, ERROR, Ev userReset, Gen furnaceReset, OFF);
    (T17, OFF, Ev heatSwitchOn, NoAction, CONTROLLER_ON);
    (T18, CONTROLLER_ON, Ev heatSwitchOff, Gen deactivate, OFF);
    (T19, CONTROLLER_ON, Ev furnaceFault, NoAction, ERROR)];
```

Figure 4.8: Textual representation of the controller statechart

statechart is found in Figure 4.9. This statechart is parameterised by the room name. It uses the next valve position table to specify an action of a statechart transition. The short-hand `adjValve` was defined on page 94 in terms of the next valve position table.

The keyword `Sc` takes a statechart structure and returns a model:

```
(:stateName, :transName)
Sc :(stateName,transName)sc -> model;
```

When a specification is stated in the statechart notation, the configuration of the system includes the set of basic states that the system is currently in as notation-specific names. Figure 4.10 presents the top-level description of both the statechart structure and the heating system specification.

Our variant of statecharts does not cover all aspects of the original statechart notation. For example, we do not describe history states.

```
roomSc (i:Room) :=
let waitedForWarm :=
      \i.Tm (En (WAIT_FOR_HEAT i) (roomSc i)) warmUpTime in
let waitedForCool :=
      \i. Tm (En (WAIT_FOR_COOL i) (roomSc i)) coolDownTime in
   OrState (ROOM i) (NO_HEAT_REQUESTED i)
   [OrState (NO_HEAT_REQUESTED i) (IDLE_NO_HEAT i)
      [BasicState (IDLE_NO_HEAT i);
       BasicState (WAIT_FOR_HEAT i)]
      [(T8 i,IDLE_NO_HEAT i,EvCond NonEvent (tooCold i),
           adjValve i,WAIT_FOR_HEAT i);
       (T9 i,WAIT_FOR_HEAT i,EvCond NonEvent (NOT (tooCold i)),
           NoAction, IDLE_NO_HEAT i);
       (T10 i, WAIT_FOR_HEAT i, waitedForWarm i,
           adjValve i, WAIT_FOR_HEAT i)] ;
    OrState (HEAT_REQUESTED i) (IDLE_HEATING i)
      [BasicState (IDLE_HEATING i);
       BasicState (WAIT_FOR_COOL i)]
      [(T15 i,IDLE_HEATING i,EvCond NonEvent (tooHot i),
           adjValve i,WAIT_FOR_COOL i);
       (T14 i,WAIT_FOR_COOL i,EvCond NonEvent (NOT(tooHot i)),
           NoAction, IDLE_HEATING i);
       (T13 i, WAIT_FOR_COOL i, waitedForCool i,
           adjValve i, WAIT_FOR_COOL i)] ]
   [(T11 i, WAIT_FOR_COOL i, EvCond (waitedForCool i) (vClosed i),
            cancelrH i, NO_HEAT_REQUESTED i);
    (T12 i, WAIT_FOR_HEAT i, EvCond (waitedForWarm i) (vOpen i),
           rH i, HEAT_REQUESTED i)];
```

Figure 4.9: Textual representation of the room statechart

```
heatingSystemScStruct :=
  AndState HEATING_SYSTEM
  [ roomSc (KITCHEN);
    roomSc (BEDROOM);
    roomSc (LIVING_ROOM);
    furnaceSc;
    controllerSc];

HeatingSystem := Sc heatingSystemScStruct;
```

Figure 4.10: Textual representation of heating system top-level specification

4.12 ScExpr (statechart expressions)

For coordination between components of the system, it can be useful to know if the system is in a particular state of a statechart specification. The keyword for this expression is `InState` and it takes as arguments a state name and the state structure that the state is a part of:

```
(:stateName,:transName)
InState : stateName -> (stateName,transName)sc -> bool;
```

4.13 ScEvent (statechart events)

Events that are of particular interest for coordination among components of a statechart are entering and exiting states. The keywords `En` and `Ex` describe these events:

```
(:stateName, :transName)
En :stateName -> (stateName,transName)sc -> (transName)event
(:stateName, :transName)
Ex :stateName -> (stateName,transName)sc -> (transName)event
```

These keywords return events customised for statechart models.

In the heating system, the furnace takes a certain amount of time to start up (`furnaceStartupTime`). Once it has been activated, it waits this amount of time before indicating to the controller that the furnace is running. The end of this delay is signalled by the event:

```
Tm (En FURNACE_ACTIVATING furnaceSc) furnaceStartupTime
```

4.14 CommAction (communication actions for statecharts)

The statecharts notation uses broadcast communication. At times, a more directed form of communication can be useful between components of a specification stated in the statecharts notation as found in the TCAS II specification in RSML. We found the same to be true in writing a formal specification of the ATN system.

Directed communication is accomplished using the action `SendData`. This communication is still broadcast but contains "addresses" in the form of state names (the destination and source states of the message). `SendData` can be thought of as the previously described `Gen` action with addresses. It works together with the `ReceiveData` keyword of the CommEvent notation discussed in the next section.

The action `SendData` takes a statechart structure as its first argument and a state name as its second argument. Its third argument is a message, which is a user-defined tag labelling the kind of communication. The elements of the type `msg` are defined in a type definition by the specifier. Therefore, the type `msg` is a type parameter to the type signature of this keyword. The fourth argument is an expression specifying the data being sent. The type of the `SendData` primitive is:

```
(:stateName,:transName,:msg,:ty) SendData
     :((stateName,transName)sc ->
      stateName ->
      msg ->
      (ty)exp ->
      action;
```

The source of the message is the state name at the root of the statechart provided as its first argument. Its destination is the state name provided as its second argument.

At times, it only matters that the message is sent and no restrictions are made on the data. The primitive `Send` has this meaning. Its type signature is:

```
(:stateName,:transName,:msg) Send
     :((stateName,transName)sc ->
      stateName ->
      msg ->
      action;
```

Send can be used as a conservative abstraction of SendData to prove properties that do not rely on the data being correctly transferred. This abstraction was necessary in the ATN example because of the size of the configuration space (Section 8.2.2).

4.15 CommEvent (communication events for statecharts)

Receiving data of a directed communication is an event. The event ReceiveData takes the statechart structure it belongs to as its first argument, the state name of the sender of the message is its second argument, a message, and data. The receiver of the message is the state name of the root of its first argument. The type signature of ReceiveData is:

```
(:stateName,:transName,:msg,:ty) ReceiveData
     :((stateName,transName)sc ->
      stateName ->
      msg ->
      (ty)exp ->
      (transName)event;
```

The message must be a type constructor. The value of the data argument of this event only has a legitimate value at the time a message is received, i.e., the communication has no buffer. Thus, this primitive is usually used in conjunction with an assignment action when transferring data.

Different types of data may be communicated by means of a SendData/ReceiveData interaction. The type variable ty is used in the declarations of both SendData and

`ReceiveData` to specify that these constants are polymorphic in the type of data being sent or received. For meaningful communication, the instantiation of this type parameter at the receiving event should be identical to its instantiation at the sending action. This constraint cannot be directly enforced by means of type checking. For instance, the sending action may specify that a value of type `bool` is being sent, but the corresponding receiving event (i.e., the one receiving messages addressed to the pair of states and labelled by the same message) may specify that a `num` value is expected. If this kind of error occurs in a specification, the value of the received data is not limited to being equivalent to the sent data. Hence, the specification of the `SendData`/`ReceiveData` interaction will be more conservative than the specifier intended. A practical strategy to mitigate this kind of error in a specification is to define type-specific messages and versions of the communication keywords limited to particular types of messages and particular types of data for an application.

The keyword `Receive` is similar to `ReceiveData` except that only the message matters and no data is received. It has the type signature:

```
(:stateName,:transName,:msg)
ReceiveData
      :((stateName,transName)sc ->
       stateName ->
       msg ->
       (transName)event;
```

Because these events are particular for statecharts, they return events of type `(transName)event`.

For example, in the ATN, the `ACSE 1` component uses the `SendData` primitive to send the message `A_ASSOCIATE_cnf_pos` to the `CF 1` component as in:

```
SendData (ACSE 1) (CF 1) A_ASSOCIATE_cnf_pos (ACSEData 1)
```

The data associated with this message is the value of the name `ACSEData 1`. The `CF 1` component receives this message using a `ReceiveData` event as in:

```
ReceiveData (CF 1) (ACSE 1) A_ASSOCIATE_cnf_pos (dataACSE 1)
```

At the time this event triggers a transition, the name `dataACSE 1` contains the data that was in the `ACSEData 1` name in the previous configuration.

4.16 Environment

The correct functionality of a system may depend on the environment satisfying certain constraints. For instance, there could be physical constraints on the relationships between inputs, or restrictions on the values of related uninterpreted constants. These assumptions form part of the specification and are often needed to eliminate impossible scenarios in analysis output.

The assumptions about the environment can be specified in any notational style. In the heating system higher-order logic is used to state some simple assumptions. These assumptions document "common knowledge" about properties of numbers that our current implementations of automated analysis cannot infer. For example, the following relates the conditions `tooCold` and `tooHot` with the possible entries in the table specifying the next valve position (`nextVp`):

```
env :=
  (forall i.
    let delta := dT i - aT i in
    (tooCold i =
         ( ((C 2 < delta) AND (delta <= C 5)) OR
         (C 5 < delta))) AND
    (tooHot i =
         ( (delta < C -5) OR
         ((C -5 <= delta) AND (delta < C -2))))) cf;
```

The value of some inputs to the system may not change over time. These values can be declared as non-lifted constants in S+ and then used in the specification by preceding them with the `C` keyword, which returns the same value for its argument in any configuration.

4.17 Summary

This chapter has presented semantic categories, which systematise combinations of notations in writing a specification. Type checking regulates the combinations of notations. The notations are represented textually as notational styles in the common format of higher-order logic. The categories of events, actions, and expressions are lifted in that they have value relative to particular configurations. Lifting is used to express dynamic behaviour.

Examples of notational styles that belong to each category were presented. The keywords of a style have type signatures that indicate join points to notations in the same category or in other categories. Well-formedness constraints for these notations are all decidable.

This chapter has presented new combinations of notations, such as using an expression in higher-order logic to specify the trigger of a statechart transition.

The next chapter presents how a meaning is associated with notational styles to make it possible to carry out configuration space exploration analysis on specifications in the framework.

Chapter 5

Semantics

This chapter presents the semantic functions for notational styles. We describe our solutions to sub-problems five and six of Chapter 1, namely, determining the meaning of a notation (Section 1.2.5) and associating meaning with representation (Section 1.2.6).

Semantic functions (or just "semantics") associate a meaning with every well-formed sentence of a notation. The semantics are written in a form that can be used directly in the framework to bring together multiple notations for analysis. Once written, the semantics can be re-used for multiple kinds of analysis and need only be consulted by specifiers when there are disputes about the meaning of a notation.

A notational style is a packaged embedding in our variant of higher-order logic (S+). The keywords presented in the previous chapter are semantic functions that are defined here. Each notation is packaged with its semantics as a "lego-like" building block to associate meaning with representation. The technique of packaged embeddings makes our framework extensible.

Section 5.1 introduces formal operational semantics. Section 5.2 discusses the techniques of shallow and deep embeddings. Section 5.3 revisits the type signatures for the categories of notations providing more details on the regulation of combinations of notations (Section 1.2.3). This section presents some accessor functions for working with notations in different categories.

In our framework, the semantics for a model define a next configuration relation. These semantics are an operational semantics and consist of a set of constraints on the relation. A next configuration relation is used in configuration space exploration techniques.

Semantic descriptions for some notations only apply to deterministic specifications and rely on "conformance" analysis to ensure the semantics are applicable to a specification. This conformance analysis is often undecidable. Examples of semantics for deterministic specifications include Heitmeyer et al.'s description of SCR [HJL96] and Heimdahl's semantics for RSML [Hei94]. In these notations, the semantics assume that only one transition can be taken at any time so conformance analysis must be done prior to any other types of analysis.

In general, we take the view that sources of nondeterminism should be known in a specification but not necessarily disallowed. Nondeterminism may exist as a specification is being developed even if it is desirable to ensure nondeterminism is eliminated in the final version of a specification. Nondeterminism is also often necessary to state environmental constraints. Analysis should be applicable throughout the specification's development. Therefore, we define the semantics of models as relations rather than functions.

As mentioned in Chapter 4, the semantics are written in S+ as definitional extensions of the formalism. A few notation-specific uninterpreted functions are introduced to capture fundamental concepts of a notation, such as being in a state of a statechart. The semantic functions are "executable" in that, when evaluated for a particular specification, they result in an expression that includes only built-in constants, constants introduced by the user, and the notation-specific uninterpreted constants.

The semantics of S+ are the same as those of higher-order logic and therefore are not presented here. The remainder of this chapter mirrors the structure of the previous chapter in presenting the semantics for each notational style. We demonstrate how semantics can be defined to associate meaning with the notational styles. This chapter is detailed to provide guidance for experts who wish to extend the framework with new notations.

While the main benefit for our framework of an embedding of notations in higher-order logic is to bring multiple notations together for analysis, the semantics can also be used to prove properties of the semantics themselves. Section 5.13 presents a property about the compositionality of AND-states in a statechart that makes it possible to partition the specification to help in analysis.

The definitions of semantic functions presented in this chapter are used directly as input to the analysis procedures[1]. The full version of the semantics for all example notational styles can be found in Appendices C through K.

5.1 Formal operational semantics

Operational semantics define the meaning of a notation by describing its behaviour on an abstract machine [Win93]. Automata are often used to define the operational semantics of a notation [Gor79]. In our framework, the semantics of model-oriented notations are defined as next configuration relations in logic, i.e., the transition relation of an automaton where the possible states of the automaton are the reachable configurations of the system. These semantics are formal because they are written in logic. They provide an interpretation of the notation in a formal theory. The next configuration relation maps pairs of configurations to a Boolean value indicating if it is a legal step to go from the first configuration of the pair to the second. Common forms of automated analysis explore the configuration space using a next configuration relation. As an introduction to this topic for the reader, in this section we briefly introduce other types of semantics and indicate why they were not chosen for use in our framework.

Structured operational semantics developed by Plotkin [Plo81] are a type of oper-

[1]A minimal amount of type information necessary for these definitions to pass type checking has been omitted for presentation. This type information is found in the appendices.

ational semantics in which the behaviour of the abstract machine is described by rules based on the constructs of the notation. Derivations can use these rules to prove the equivalence of constructs. These rules are based on a decomposition of the structure of the constructs. Because compositionality in the meaning of some notations, such as statecharts, is not based on the structure of the specification, structured operational semantics alone are not suitable for our framework. Briefly, transitions that cross state boundaries in a statechart result in a compositionality based on scope of transitions rather than state hierarchy (Section 5.8).

Denotational semantics (invented by Stratchey and Scott) define the meaning of a notation as a series of functions from configurations to configurations. Each function describes the meaning of one construct in the notation. Using denotational semantics, it is possible to compare the meaning of constructs without including a configuration argument to the function. The ability to define the meaning of a notation compositionally based on its constructs is inherent to the denotational semantics approach. Denotational semantics are often used to reason about notations within theorem proving environments. Examples of denotational semantics for programming languages can be found in Gordon [Gor79] and Winskel [Win93].

Axiomatic semantics are used to reason about specific programs by rewriting the program into more basic statements in the same notation. Reasoning about the programs is usually accomplished by means of rewriting in a theorem prover.

Operational semantics are the most suitable choice for our purposes for three reasons. First, models denote automata. Second, the meaning of some specification notations cannot be defined compositionally. Third, we provide a common interface to analysis techniques by means of logic and therefore wish to define the meaning of the notation in logic (rather than in terms of itself). This work uses definitions in higher-order logic to define the formal operational semantics for model-oriented notations.

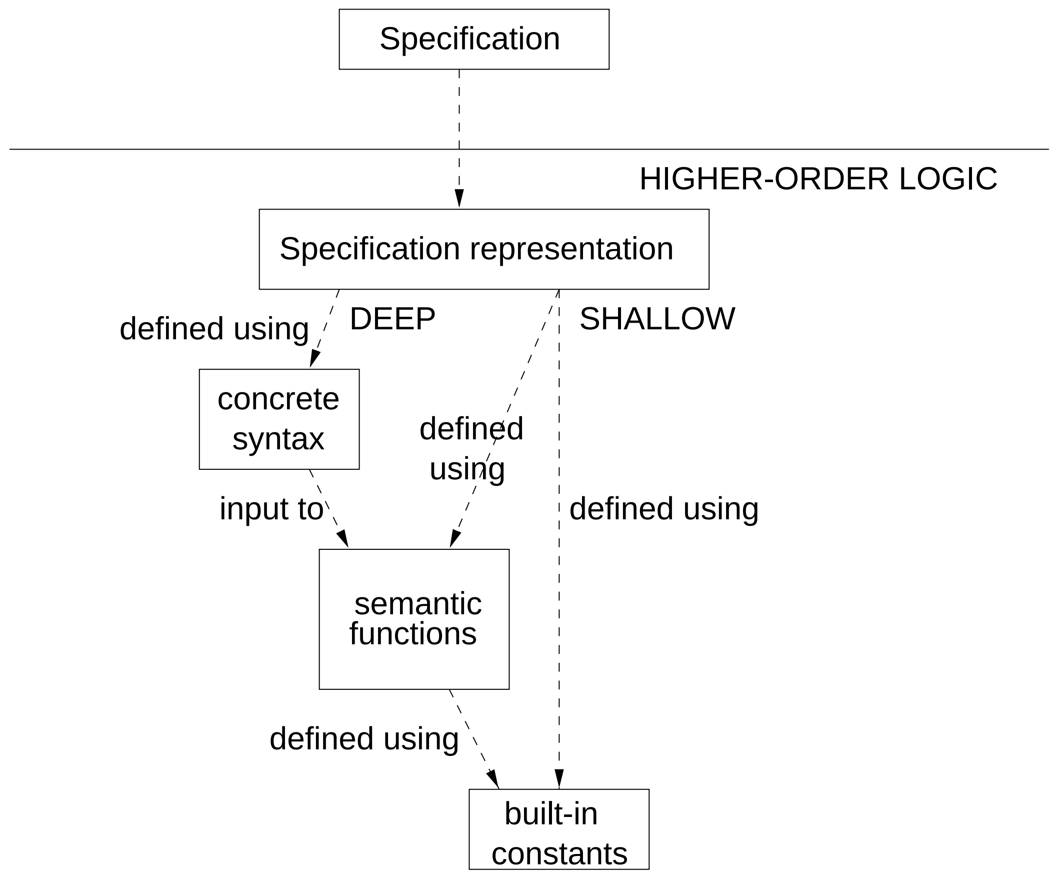

Figure 5.1: Deep and shallow embeddings

5.2 Embeddings

An embedding of a notation is a description of its semantics in logic. This technique is often used with theorem provers to study a notation. There are two common methods for writing embeddings: shallow and deep. We use the description of these terms found in Boulton et al. [BGG$^+$92]. An illustration of these two techniques in found in Figure 5.1. The framework uses a combination of these two techniques to achieve the packaging of notations with their meanings.

In a *deep* embedding, the concrete syntax is represented in the logic as a type. This representation is called an abstract syntax and mirrors the original notation in structure. The approach of deep embeddings is depicted in the path on the left-hand side of Figure 5.1

that goes through the concrete syntax. Functions are defined that map the abstract syntax to its meaning in the logic. For example, the following introduces an abstract syntax for an expression notation that consists of two operators `ADD` and `SUB`:

```
: expr := NUM :num
        | ADD :expr :expr
        | SUB :expr :expr ;
```

The above type definition introduces three type constructors: `NUM`, `ADD`, and `SUB`. The constructor `NUM` is necessary to turn a number into an element of the abstract syntax. A pattern-matching definition can be used to map elements of the type `expr` into their logical meanings:

```
SemExpr (NUM a) := a |
SemExpr (ADD a b) := SemAdd (SemExpr a) (SemExpr b) |
SemExpr (SUB a b) := SemSub (SemExpr a) (SemExpr b) ;
```

The functions `SemAdd` and `SemSub` are defined in the logic as the addition and subtraction functions:

```
SemAdd := + ;
SemSub := - ;
```

The meaning of the expression `ADD (SUB (NUM 5) (NUM 4)) (NUM 6)` is the result of applying the function `SemExpr` to this expression in the abstract syntax.

In a *shallow* embedding, there is no distinction in the logic between the syntax and the semantics. The syntax is represented as defined constants. The approach of shallow embeddings is depicted in the path on the right-hand side of Figure 5.1. The steps of representing an abstract syntax and then writing functions that map the abstract syntax into its logic meaning are not part of a shallow embedding. The meaning is captured directly in the syntax itself. The step of applying a function to return the meaning of a piece of syntax is unnecessary. Using the same example, a shallow embedding of the tiny expression notation is:

```
ADD a b := a + b;

SUB a b := a - b;
```

The extra construct `NUM` is not needed. The function `ADD` is equivalent to the function `SemAdd` in the deep embedding. The meaning of the expression `ADD (SUB 5 4) 6` is determined by evaluating this expression in the logic.

A shallow embedding has the desired property of extensibility for our framework. Without a shallow embedding, adding a new notation that can be used at the join points of existing notations would require additions to existing notation's abstract syntax and the functions mapping the abstract syntax into the logic. Another advantage of a shallow embedding is that the type checker of the logic can be used as the type checker for all notations. This approach would not work for notations that have different type checking rules than the logic.

A shallow embedding has two disadvantages. The first is that the distinction between the notation and the logic is lost, which means reasoning based on the form of the specification cannot be carried out. For example, Section 1.2.10 described how the arrangement of entries in a row may form a partition of a numeric value that can be used to create a more precise abstraction for analysis. Recognition of this structure and a definition of the abstraction cannot be defined in the same logic as the shallow embedding of the notation. For our framework, this disadvantage is not a serious concern. The analysis functions are implemented in C and can walk over the parse tree representation of the specification.

A second disadvantage to a shallow embedding is that in order for it to mirror the structure of the original specification the way a deep embedding does, the meaning of the notation must be compositional based on the structure of the notation. Statecharts do not have the property of compositionality in meaning. This disadvantage can cause shallow embeddings to be a largely expanded (and potentially unrecognisable) representation of the specification. This expansion would compromise the rigour that is achieved through direct use of the semantic functions.

This second disadvantage of shallow embeddings means they are not suitable for all notations to achieve the desired qualities of the framework. For notations such as statecharts, we use a deep embedding where the structure is represented using type constructors. To create the lego-like block needed for extensibility, we include the keyword `Sc` in the statecharts notational style. `Sc` takes a statechart structure created by type constructors as a parameter. `Sc` is only needed at the top of the statechart, i.e., every substatechart does not need this prefix. `Sc` packages the meaning of a statechart with the notation allowing our approach to be extensible. The join points to other notations are still regulated by type consistency.

In summary, we use both deep and shallow embeddings. We call our approach packaged embeddings. Shallow embeddings are used when the meaning of the notations is compositional. Deep embeddings are used when the notation is not compositional but we maintain the property of extensibility by prefixing the structure with its semantic function. In our experience this prefix has not proved cumbersome. Our approach is extensible to new notations. Another advantage of packaged embedding in higher-order logic is that parameterisation of specification parts does not require any extension to either the notation or its semantics.

5.3 Semantic categories

The previous chapter presented the categories of notations used in model-oriented specifications. There are expressions, events, actions and models. Expressions have the type signature:

```
: (ty) exp == config -> ty;
```

As explained earlier in Chapter 4, the types `bool` and `num` are equivalent to `(BOOL)exp`, and `(NUM)exp` respectively.

We use the following type abbreviation to describe a step:

```
:step == config # config;
```

Models have the type signature:

```
: model == step -> BOOL;
```

The first element of the pair of configurations constituting a step is called the *previous configuration*. The second element of the pair is called the *next configuration*. These elements are referred to using the functions:

```
Prev (x:step) := FST x;
Next (x:step) := SND x;
```

The type signatures for the event and action categories are type abbreviations and will be explained in Sections 5.3.1 and 5.3.2. The complete set of type signatures and associated accessor functions for these categories are found in Appendix C.

We use tuples in S+ as a form of data structure for capturing the multiple elements necessary to describe the meaning of the event and action categories. Functions are used to access elements of these data structures.

5.3.1 Events

Events are instantaneous occurrences, such as changes in conditions. The meaning of an event is an indication of whether the event occurred in the previous step. The assignment of meaning to events such as timeouts, also requires keeping track of the history of an event. Capturing the history of events when only two configurations (the previous and next) can be referenced may involve creating auxiliary names that have values in a configuration. The values of these auxiliary history names require updating in each step. Giving meaning to nested events requires knowing if other events are occurring in this step. Finally, an event may include some initialisation constraints, which, in turn, depend on whether other events are occurring at initialisation. Therefore, an event contains five components. These components are captured in a type abbreviation describing a five-tuple:

```
:(label)eventinfo ==
     bool #                              /* Occurred */
     (BOOL->(label)ext_step -> BOOL) # /* Update */
     ((label)ext_step -> BOOL) #       /* Occurs */
     (BOOL-> bool) #                   /* Init */
     bool);                            /* OccursAtInit */
```

The components of an event are accessed using the functions, `Occurred`, `Update`, `Occurs`, `Init`, and `OccursAtInit`.

The "occurred" field references only one configuration – the previous configuration of a step. It indicates if the event occurred in the previous step and therefore takes an argument that is a configuration. The "occurs" and "update" fields may depend on both the previous and next configurations of a step.

The "update" and "init" fields ensure that the history of the event is updated and initialised, respectively. The semantics for event notations introduce uninterpreted constants to capture this history. In a nested event, the history of an event at one level may be sufficient to deduce the history of events at lower levels. To reduce the number of constraints (i.e., the size of the configuration space), a Boolean flag is the first argument of these functions indicating if history constraints have already been set.

The "init" and "occurs at init" fields depend on only one configuration – the initial configuration.

Some events require more information than just the previous and next configurations in a step. This information can be customised for the model using the event and is captured in the type parameter `label`. For example, the events of entering or exiting a state of a statechart depend on the set of transitions followed in a step because looping transitions are possible where the meaning of the event cannot be determined from the status of the states in the previous and next configurations. The definition of `ext_step` (for "extra information in a step") and the associated functions for determining the previous and next configurations are:

```
:(label)ext_step == (config # config) # (label->BOOL);
(:label)ExtPrev (x:(label)ext_step) := FST (FST x);
(:label)ExtNext (x:(label)ext_step) := SND (FST x);
```

These functions are used in the semantic functions for particular event notations.

Keeping track of an event's history for timeouts requires a means of identifying which event the history is associated with. We chose a simple identifier scheme to match histories with events based on the nesting of events. Each top-level event must have a unique label associated with it. A top-level event is one that is used by a model rather than by another event. The label is often the transition name. To isolate a nested event within the top-level event, a path is constructed that follows the branching of the nested events as direct subevents (S), or left (L) or right (R) branches. Figure 5.2 presents an example of the assignment of identifiers to an event and its component events using the label "T2". Starting from the label of the event, the path part of the identifier is determined from the nesting of events. The initial path is an empty list.

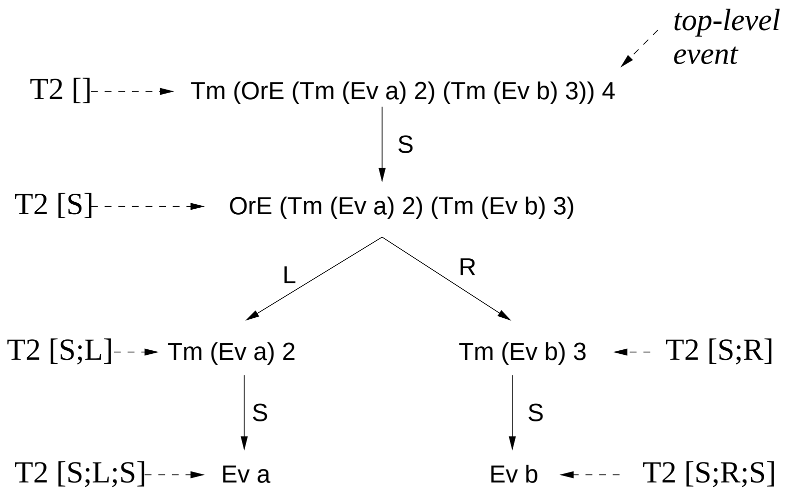

Figure 5.2: Example of event identifiers

An event is a function that takes a label and a path and returns the event information:

```
: eventLabelPath := S | L | R;

: path == (eventLabelPath)list;

: (label) event ==

     label -> path -> (label)eventinfo;
```

Within the semantic definitions for events, the functions `Sub`, `Left`, and `Right` are used to construct the path for nested events. To give the meaning of some events, it is not necessary to use the event's identifier.

5.3.2 Actions

An action is the specification of zero or more modifications to the values associated with particular assignable names from one configuration to the next. If there are multiple actions in a specification, there is the potential that multiple actions will constrain the value of the same name in the next configuration creating a race condition. The semantics of the notational style of model determine how race conditions are handled. An action returns the meaning of the modifications in a step. For the model to maintain the values of assignable names that do not change, it needs to know the assignable names affected and a relation over a step that ensures that the value of the name does not change. Thus a modification has three parts:

- an identifier for the name being modified
- the change it effects (a model)
- a condition ensuring the name does not change (a model)

Names are uninterpreted constants. Comparing two names to ensure that their identifiers are different is a meta-level operation. S+ includes a built-in uninterpreted function called `NAME`, which takes as an argument an assignable name and returns a string. Informally, we interpret the return value of `NAME` to be an identifier that can be used to distinguish names from each other. Section 6.9.3 shows how implicit assumptions about the equality and inequality of the return values of this uninterpreted function applied to names are used in evaluation.

Since an action can contain multiple modifications, its type signature is a list of modifications. A modification is a three-tuple capturing the necessary information for describing an action:

```
: mod == STRING # model # model;
: action == (mod)list;
```

The order of the elements in the list is not significant with respect to its semantics.

5.4 Common functions

Standard list manipulation functions, such as `hd` (head), `tl` (tail), `map`, and `append` are used in the definitions of the semantic functions. The functions `every` and `any` take a predicate and produce the conjunction or disjunction, respectively, of the predicate applied to every element of the list. The functions `Every` and `Any` take a lifted predicate and carry out the same operation resulting in a lifted expression. The polymorphic constant `UNKNOWN` is used to represent an unknown value if the function is partially specified. The declarations and definitions of these constants can be found in Appendix D.

5.5 TableExpr

The tabular style of expression specifies functions. We begin by describing the meaning of a row. A row in a table is a list of predicates to be applied to the row label. The keyword `Row` has the type signature:

```
(:ty) Row :(ty)rowlabel -> ((ty)rowentry)list -> row;
```

The type parameter `ty` is the type of the value of the row label in a configuration.

A meaning is associated with the keyword `Row` by defining it as a semantic function. The meaning of a row is a list of conditions produced by applying the predicate in each row entry to the row label. `Row` is defined as:

```
(:ty)
Row (rl:(ty)rowlabel) (res:((ty)rowentry)list) :=
   RowAux res rl;
```

The auxiliary function `RowAux` recursively applies the row label to each row entry:

```
(:ty)
RowAux NIL (rl:(ty)rowlabel) := NIL |
RowAux (CONS (re:(ty)rowentry) res) rl :=
    CONS (re rl) (RowAux res rl);
```

The next step in giving meaning to a tabular specification is to combine the predicates in the rows to produce an expression that represents a column. The `Columns` function produces a list that consists of the expression formed by conjoining the elements at the same position in each row:

```
Columns (rs:(row)list) :=
    /* last column is a list of empty lists */
    COND ((hd rs) EQ NIL) NIL
    (CONS (Every (hd) rs) (Columns (map rs tl)));
```

A predicate table, specified using the keyword `PredicateTable`, is a function that returns true if any of its columns are true. It has the type signature:

```
PredicateTable :(row)list -> bool;
```

It is defined as:

```
PredicateTable rowMatrix :=
    Any (\x.x) (Columns rowMatrix);
```

The order of the columns does not matter in a predicate table.

In a function table, the columns have precedence determined by their left to right order. When the condition represented by the column is satisfied, the value at the bottom of the column is returned. The keyword `Table` has the type signature:

```
(:ty)
Table :(row)list -> ((ty)exp)list -> (ty)exp;
```

The meaning of `Table` is defined by nested conditional expressions. It is defined using the auxiliary function `TableAux`, as in:

```
(:ty)
Table rowMatrix resultRow :=
     TableAux (Columns rowMatrix) resultRow;
```

The auxiliary function `TableAux` walks over the columns and matches them to values in the result row. When the list of columns has been exhausted, either the default value (the last value in the list of return values), or the value `UNKNOWN` for a partially specified function is returned:

```
(:ty)
TableAux (NIL) (resultRow:(ty)list) :=
    COND (resultRow EQ NIL)
    UNKNOWN
    (hd resultRow) |
TableAux (CONS col cols) resultRow:=
    if col
    then (hd resultRow)
    else TableAux cols (tl resultRow);
```

By definition a function table cannot be inconsistent since there is a priority order that the columns are matched in determining the value of the function. However, as will be seen in Chapter 7, consistency checking can be carried out to ensure that the order of the columns makes no difference to the meaning of the table.

Next, we can define the keywords of the style that are possible row entries in a table:

```
(:ty)Dc := \(x:(ty)exp). C T;
True :=  \x. x = C T;
False := \x. x = C F;
```

The keywords used in row labels, `AllOf` and `AtLeastOneOf` have the following definitions:

```
(:ty) AllOf (rl:((ty)exp)list) (p:(ty)exp->bool) := Every p rl;
(:ty) AtLeastOneOf (rl:((ty)exp)list) (p:(ty)exp->bool) := Any p rl;
```

These semantic functions result in the specification of the desired temperature of the heating system presented in Figure 4.3 having the meaning found in Figure 4.5 on page 90.

5.6 CoreEvent

This section presents the meanings of the keywords `Ev`, `Ch`, `EvCond` and `Tm` of the CoreEvent notational style of events. The complete set of semantic definitions for CoreEvent can be found in Appendix F.

An event specified using the `Ev` keyword is a primitive event depending only on its argument, which is an uninterpreted Boolean constant. If its argument is true in the next configuration then the event is occurring in this step:

```
(:label)
EvOccurs (ev:simpleEvent) (step:(label)ext_step) :=
   ev (ExtNext step);
```

The event occurred in the previous step if its argument is true in the previous configuration (`cf`):

```
EvOccurred (ev:simpleEvent) (cf:config) := ev cf;
```

A primitive event occurs at initialisation if its argument is true at initialisation:

```
EvOccursAtInit (ev:simpleEvent) (cf:config) := ev cf;
```

Using these elements, the meaning of the `Ev` keyword is defined as a function that maps a label and a path into a five-tuple of the type `eventinfo`:

```
(:label)
Ev (ev:simpleEvent) (lab:label) (p:path) :=
    (EvOccurred ev,
     \flag. \step. T,      /* no history to update */
     EvOccurs ev,
     \flag. C T,           /* no initialisation of history needed */
     EvOccursAtInit ev);
```

Each line in the right-hand side of this definition is one element of the event's information.

The `Ch` event is similar to a primitive event except that it notes the change in a condition rather than simply that the condition is true. To record the occurrence of a change, we introduce a history function as an uninterpreted constant. This history function, called `Changed`, is declared as:

```
(:label) Changed : label -> path -> bool;
```

This function returns true if the event associated with the label and path identifier occurred in the previous step. Therefore, it can be used directly as the "occurred" component of an event:

```
(:label)
ChOccurred (lab:label) (p:path) (cf:config) :=
    Changed lab p cf;
```

The `Ch` event occurs in a step if the value of the condition is not the same in the previous and next configurations:

```
(:label)
ChOccurs (cond:bool) (step:(label)ext_step) :=
    ~(cond (ExtPrev step) EQ cond (ExtNext step));
```

The update component of the event ensures that the value of the history function in the next configuration captures the result of the `ChOccurs` component:

```
(:label)
ChUpdate (cond:bool) (lab:label) (p:path)
  (flag:BOOL) (step:(label)ext_step) :=
    (~flag) \/
    (Changed lab p (ExtNext step) EQ ChOccurs cond step);
```

The constraint on the `Changed` function is only necessary if it has not previously been captured (i.e., `flag` is true). For example, if this event is the component of a timeout event, a later section shows that the timeout event captures sufficient history and that this extra constraint is not needed[2]. These elements can be grouped together to give a definition for the `Ch` event:

[2]The meaning would not change if the constraint is included.

```
(:label)
Ch (cond:bool) (lab:label) (p:path) :=
   (ChOccurred lab p,
    ChUpdate cond lab p,
    (:label)ChOccurs cond,
    \(flag:BOOL).\(cf:config). Changed lab p cf EQ F,
    \(cf:config). F);
```

A `Ch` event does not occur at initialisation (the fifth component) and the history function has the value false for the event at initialisation (the fourth component).

Other event notations include more specialised events for changes in conditions, such as the condition becoming true. Discussions of the semantics of these notations can be found in Heitmeyer et al. [HJL96] and Atlee and Buckley [AB96].

The meaning of the `EvCond` keyword relies on the meaning of its two arguments, which are an event and a condition (Boolean expression). An `EvCond` event occurred in the previous step if its component event occurred and its condition is true in the previous configuration:

```
(:label)
EvCondOccurred (ev:(label)event) (b:bool) (lab:label)
        (p:path) (cf:config) :=
   Occurred ev lab (Sub p) F cf /\ b cf;
```

An `EvCond` event occurs if its component event is occurring in this step and its condition is true in the next configuration:

```
(:label)
EvCondOccurs (ev:(label)event) (b:bool) (lab:label) (p:path)
          (step:(label)ext_step) :=
   EventOccurs ev lab (Sub p) step /\ b (ExtNext step);
```

The event occurs at initialisation if its component event occurs at initialisation and its component condition is true:

```
(:label)
EvCondOccursAtInit (ev:(label)event) (b:bool) (lab:label) (p:path) :=
    EventOccursAtInit ev lab (Sub p) AND b;
```

The `EvCond` keyword is defined as:

```
(:label)
EvCond (ev:(label)event) (b:bool) (lab:label) (p:path) :=
    (EvCondOccurred ev b lab p,
     Update ev lab (Sub p),
     EvCondOccurs ev b lab p,
     Init ev lab (Sub p),
     EvCondOccursAtInit ev b lab p);
```

Updating this event involves updating any history that must be kept for its component event (second element of the tuple). Initialising this event involves initialising its component event. Initialisation of the condition is done by the user.

The meaning of a `Tm` (timeout) event depends on the history of occurrences of its component event. To capture this history, the uninterpreted constant `TimeEventLastOccurred` is introduced with the following constant declaration:

```
(:label) TimeEventLastOccurred : label -> path -> num;
```

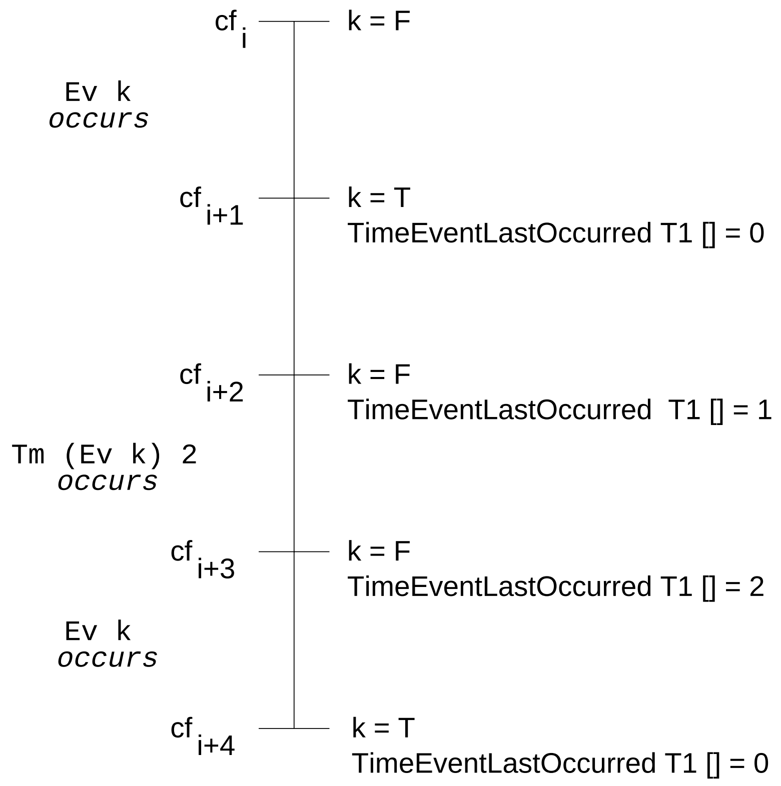
cf i
k = F
Ev k occurs
cf i+1
k = T
TimeEventLastOccurred T1 [] = 0
cf i+2
k = F
TimeEventLastOccurred T1 [] = 1
Tm (Ev k) 2 occurs
cf i+3
k = F
TimeEventLastOccurred T1 [] = 2
Ev k occurs
cf i+4
k = T
TimeEventLastOccurred T1 [] = 0

Figure 5.3: Behaviour of Tm (Ev k) 2

This function returns the number of steps that have passed since the event determined by the label and path identifier occurred. Maintaining the status of this uninterpreted constant is only necessary for timeout events because the meaning of other events can be derived from their component events. Figure 5.3 shows the behaviour of the event `Tm (Ev k) 2`, with label `T1` through a series of configurations.

A timeout event occurred in the previous step if the time since the event last occurred, as recorded by the function `TimeEventLastOccurred` applied to the timeout identifier, equals the delay of the timeout:

```
(:label)
TmOccurred (n:num) (lab:label) (p:path) (cf:config) :=
    (TimeEventLastOccurred lab p = n) cf ;
```

The delay of the timeout is always evaluated in the previous configuration. The event component of a timeout is not needed in this definition.

Similarly, a timeout event occurs in this step if the value of the history function in the next configuration is the same as the value of the delay in the next configuration:

```
(:label)
TmOccurs (n:num) (lab:label) (p:path) (step:(label)ext_step) :=
   (TimeEventLastOccurred lab p = n) (ExtNext step);
```

The update component of a timeout event constrains the possible value of the function `TimeEventLastOccurred` for this event in the next configuration. Its next value is either its previous value incremented by one, if the component event does not occur in this step, or the value zero, if the component event does occur. The history function `TimeEventLastOccurred` records only the most recent occurrence of the event. The update component is[3]:

```
(:label)
TmUpdate (ev:(label)event) (lab:label) (p:path) (flag:BOOL)
    (step:(label)ext_step) :=
  (COND (EventOccurs ev lab (Sub p) step)
    ((TimeEventLastOccurred lab p = (C 0)) (ExtNext step))
    (TimeEventLastOccurred lab p (ExtNext step) EQ
         ((TimeEventLastOccurred lab p + (C 1)) (ExtPrev step))))
    /\
    Update ev lab (Sub p) F step;
```

The component event of the timeout may also require updating constraints as found in the last line of the above definition where `Update` is used. The argument `F` (false) to `Update` indicates that this history has been captured for the component event.

[3]In the appendix, this function is not defined using the `COND` operator but rather the expanded expression of this operator. This expansion avoids the need for rewriting in evaluation as described in the next chapter.

A timeout can only occur at initialisation if its delay is zero (making it equivalent to the component event by itself) and its component event occurs at initialisation:

```
(:label)
TmOccursAtInit (ev:(label)event) (n:num)
         (lab:label) (p:path) (cf:config) :=
    (n cf EQ 0) /\ EventOccursAtInit ev lab (Sub p) cf;
```

The initial value of the `TimeEventLastOccurred` function is constrained at initialisation to be zero if the component event is occurring at initialisation. Otherwise its value is unconstrained. Initialisation of the nested event is also required. The initialisation component of a timeout event is defined as:

```
(:label)
TmInit (ev:(label)event) (n:num) (lab:label) (p:path)
  (flag:BOOL) (cf:config) :=
    (TmOccursAtInit ev n lab (Sub p) cf EQ
           ((TimeEventLastOccurred lab p = (C 0)) cf ))
    /\ Init ev lab (Sub p) F cf;
```

The complete meaning of a timeout event is:

```
(:label)
Tm (ev:event) (n:num) (lab:label) (p:path) :=
    (TmOccurred n lab p,
     TmUpdate ev lab p,
     TmOccurs n lab p,
     TmInit ev n lab p,
     TmOccursAtInit ev n lab p);
```

The same event used in multiple places in a system may result in duplicate constraints. These constraints will never be in conflict. The same timeout used multiple places will have different identifiers and therefore maintain the value of the function `TimeEventLastOccurred` for different identifiers even though those values will always be equivalent.

5.7 CoreAction

The `Asn` action is an assignment of a value to a name in the next configuration. Its meaning is defined as a three-tuple:

```
Asn (v:(ty)exp) (exp:(ty)exp)
    := [( NAME v,
          \step. v (Next step) EQ exp (Prev step),
          \step. v (Next step) EQ v (Prev step),
        )];
```

Each element of this tuple is one of the fields of an action. These fields are: an identifier for the assignable name being changed, the meaning of the change, and the constraint that the name does not change in value. The second argument to the assignment statement is evaluated in the previous configuration. For example, the action in the heating system, `Asn (requestHeat KITCHEN) (C T)` has the meaning:

```
[(NAME (requestHeat KITCHEN),
  \step. (requestHeat KITCHEN) (Next step) EQ (C T) (Prev step),
  \step. (requestHeat KITCHEN) (Next step) EQ
             (requestHeat KITCHEN) (Prev step))]
```

The `Both` action concatenates the lists of modifications from each of its component actions:

```
Both (a1:action) (a2:action) := append a1 a2;
```

The action `Gen` assigns the value true to its argument in the next step. If this action is not taken then the argument has the value false (i.e., the internal event did not occur). It is defined as:

```
Gen (ev:simpleEvent)
   := [( NAME ev,
         \step. ev (Next step) EQ T,
         \step. ev (Next step) EQ F)];
```

The `No_action` keyword returns an empty list of modifications:

```
NoAction := NIL;
```

5.8 CoreSc

The semantics for statecharts describe the behaviour of the transitions and states of the specification. The enabling of individual transitions is determined by the current set of states and by the status of events on their triggers in the previous configuration. In turn, the enabling of individual transitions determines possible combinations of transitions that can be taken together in a step. Multiple possible combinations may exist because a statechart can be nondeterministic. Transitions that are taken result in actions that constrain the step.

The semantics of the statecharts style cannot be defined compositionally over the state hierarchy because transitions can cross state boundaries and have priority based on their scope. The semantics could be considered compositional over the scope of transitions, but this approach does not deal appropriately with race conditions if multiple transitions modify the same variable. Therefore, we use an alternate approach than structured operational semantics to give the semantics of statecharts.

The semantics for our statecharts variant are decomposed into three parts. The first part, called the transition state condition (`TransStateCond`), deals with the fundamental elements of the notation: transitions and states. It constrains the set of transitions that can be taken in a step and the result of taking those transitions on the set of states. The second part of the statechart semantics is the name condition (`NameCond`), which describes the changes in values of assignable names based on the set of transitions taken. This part of the semantics interfaces with an action notation. The third part is the event condition (`EventCond`), which constrains the history of events to be maintained as defined by the event notation. Both the transition state condition and the event condition interface with an event notation. The meaning of a statechart, defined in terms of these parts, is:

```
(:stateName,:transName)
ScAux (s:(stateName,transName)sc_struct)
      (trs:((stateName,transName)trans)list)
      (step:(transName)ext_step) :=
    TransStateCond s trs step /\
    NameCond s trs step /\
    EventCond s trs step ;


(:stateName,:transName)
Sc (s:(stateName,transName)sc_struct) (stp:step) :=
    let trs := transInState s in
    existsn (length trs)
    (\flags.
    let ext_step := (stp,match (map trs transLabel) flags) in
    ScAux s trs ext_step;
```

For a statechart `s` and pair of configurations `cf` and `cf'` in the parameter `stp`, `Sc` evaluates to true if and only if `cf'` is a possible successor of the current configuration `cf`.

For coordination between the transition state condition, name condition and event condition, each transition is associated with a Boolean variable that indicates whether or not the transition is taken in this step. These Boolean variables are created and existentially quantified using the function `existsn`. Within the semantic definitions for statecharts, a step (a pair of configurations) is extended to include not only the configurations but also a third element that matches transition labels with an ordered list of these Boolean variables. This extension captures information used in multiple places in the semantics. It is captured in the step argument because this information may be needed by the events of transitions. This step, with extra information, is an instantiation of the type definition for `ext_step` presented in Section 5.3.1 where label is a transition name, as in:

```
:ext_step == (config # config) # (transName->BOOL);
```

Therefore, `ExtPrev` and `ExtNext` will be used to access the two configurations of the step. The function `TransTaken` takes two arguments: a step with this extra information, and a transition name. It returns the Boolean variable associated with that transition name. This function has the definition:

```
(:stateName,:transName)
TransTaken (tr:(stateName,transName)trans)
  (step:(transName)ext_step) :=
    (SND step) (transLabel tr);
```

The label of the transition is returned by the accessor function `transLabel`. The next section describes accessor functions for the statechart structure, such as `transInState`. These functions provide useful abstractions in presenting the semantics and isolate the semantics from the form of the textual representation chosen. Subsequent sections define the semantic functions `TransStateCond`, `NameCond`, and `EventCond`. The definitions of the accessor functions, as well as the complete semantic functions, are found in Appendix H.

One optimisation for evaluation appears in the appendix but not in this description. This optimisation is described in the next chapter in Section 6.6.

All the semantic functions for statecharts are parameterised by the type of state names and transition names. These types are defined in the specification.

5.8.1 Accessor functions for the statechart structure

A preliminary step in the definition of the semantics is the presentation of accessor functions for the statechart structure that allow us to filter out syntax concerns and concentrate on semantic issues. The type definition for the statechart structure is:

```
: (stateName,transName) sc_struct :=
    OR_STATE
       :stateName                                 /* state name */
       :stateName                                 /* default state */
       :((stateName,transName)sc_struct)list    /* substatecharts */
       :((stateName,transName)trans)list        /* transition list */
    | AND_STATE
       :stateName                                 /* state name */
       :((stateName,transName)sc_struct)list    /* substatecharts */
    | BASIC_STATE
       :stateName                                 /* state name */
```

The keywords `OrState`, `AndState`, and `BasicState` are defined to return the appropriate element of this type:

```
(:stateName,:transName)OrState := (:stateName,:transName)OR_STATE;
(:stateName,:transName)AndState := (:stateName,:transName)AND_STATE;
(:stateName,:transName)BasicState := (:stateName,:transName)BASIC_STATE;
```

These keywords are synonyms for the type constructors. In the naming convention we adopted, type constructors are written in upper case letters and keywords have the first letter upper case and the remainder of the identifier lower case.

The following accessor functions to this structure are used in our presentation:

- `stateName` s: returns the name of the root state in the statechart s.
- `state` s stn: returns the substatechart that has the statename stn at its root in the statechart s.
- `isBasicState` s: returns true if the root state in the statechart s is a basic state.
- `isAndState` s: returns true if the root state in the statechart s is an AND-state. If both of the functions `isBasicState` and `isAndState` return false for a statechart, then it must be an OR-state.
- `stateSubstates` s: returns a list of the substatecharts of the statechart s.
- `basicStatesEntered` s: returns a list of the names of the basic states entered when the statechart s is entered. This calculation follows the default arrows.
- `basicStatesExited` s: returns a list of the names of all the basic states within the statechart s.
- `transScope` tr s: returns the *scope* of the transition tr, which is the state name of the least common ancestor OR-state of the source and destination of the transition tr in statechart s. For example, the scope of transition `T22` in Figure 4.6 on page 96 is `CONTROLLER_ON`.
- `transInState` s: returns a list of the transitions within the statechart s.
- `transSrc` tr: returns the name of the source state of the transition tr.
- `transDest` tr: returns the name of the destination state of the transition tr.

- `transEvent` *tr*: returns the event of the transition *tr*.
- `transLabel` *tr*: returns the name of the transition *tr*.

The name of a state is distinguished from the statechart structure whose root state has that name.

5.8.2 Transition state condition

A step in a statechart specification involves following a (possibly empty) set of transitions to move from one set of states to another set of states. The fundamental elements of a statechart are its states and transitions. It is only necessary to know the set of basic states that the system is currently in because the status of all other states in the hierarchy can be determined from the basic states. Our definition of the core semantics for statecharts relies on one uninterpreted function that describes the status of the basic states in a configuration:

```
InBasicState : stateName -> bool;
```

The core semantic functions constrain the values of this function in the previous and next configurations for all the basic states of the statechart. They also constrain the set of Boolean flags representing which transitions are taken. This section presents the definition of the predicate `TransStateCond` which gives these constraints.

Only transitions that are enabled can be taken in a step. A transition is enabled if the system is currently in its source state and the event triggering the transitions has just occurred.

A system is in a state at any level in the hierarchy if it is in any of the basic states that are descendants of this state within the hierarchy. This constraint is[4]:

[4] The `COND` operator is used here because the condition can be determined to be true or false at the time of evaluating the function for a particular statechart structure and therefore the expression can be reduced without rewriting.

```
(:stateName,:transName)
inAnyBasicState cf (s:(stateName,transName)sc_struct) :=
    COND (isBasicState s) (InBasicState (stateName s) cf)
      (COND (isAndState s)
        (every (inAnyBasicState cf) (stateSubstates s))
        (any (inAnyBasicState cf) (stateSubstates s)));
(:stateName,:transName)
Instate stName (s:(stateName,transName)sc_struct) (cf:config) :=
    inAnyBasicState cf (state s stName) ;
```

The first predicate applies to a statechart structure. The second predicate has a state name as an argument. The statechart associated with this state name is determined by means of the accessor function `state`.

The predicate that determines whether or not a transition is enabled depends on the value of the `Instate` predicate applied to its source state and the `EventOccurred` predicate applied to its event:

```
(:stateName,:transName)
Enabled (s:(stateName,transName)sc_struct)
  (tr:(stateName,transName)trans) :=
    Instate (transSrc tr) s AND
    EventOccurred (transEvent tr) (transLabel tr) [];
```

The function `EventOccurred` is defined as the accessor function for the "occurred" component of the event applied to the value true indicating the event is a top-level event. Accessor functions for event components (in this case the "occurred" component) require an identifier. The transition name is used as the first part of this identifier in this case. Since this event is a top-level event, the second part of the identifier is an empty list.

Events are evaluated relative to the previous configuration[5].

The choice of transitions is prioritised by the scope of the transitions. An enabled transition that has a higher scope in the state hierarchy than another enabled transition will be taken and the one at the lower level will not be taken. The core semantic functions for statecharts can therefore be recursively defined relative to the hierarchy of the statechart and the scopes of the transitions:

```
(:stateName,:transName)
TransStateCond (s:(stateName,transName)sc_struct) trs step :=
    TransStateCondAux s (map trs (\t.(transScope t s,t))) step;
```

The second argument to `TransStateCondAux` is a list of pairs that match the scope of a transition (a state name) with its description.

The definition of `TransStateCondAux` will be presented by examining each type of state (AND, OR, basic) that could be at the root of the statechart structure found in its first argument. These parts are not presented as definitions because S+ does not allow mutually recursive definitions; for the moment, we present fragments of S+. The complete definition is found in Figure 5.4.

The effect of a transition on the basic states is to exit the source state of the transition and enter its destination state. The statechart hierarchy constrains the possible set of states that the system can be in for a configuration. The system can be in only one substate of an OR-state. If the system is in an AND-state, it must be in every substate of the AND-state. The definition of `TransStateCondAux` captures these constraints.

OR-states

If the root state is an OR-state then the transitions with this scope must be considered. Their effects cannot be confined to any smaller part of the statechart than this level. An

[5]Unlike the presentations of the semantics of statecharts presented in Harel et al. [HPSS87] and Pnueli and Shalev [PS91], we do not introduce the notion of microsteps.

auxiliary function, called `thisScope`, is used to partition the set of transitions into those with scope at this level and those with a scope at a lower level in the hierarchy. For the transitions with scope at this level, there are two possible scenarios. One is that there is an enabled transition from this set that is taken in this step. The second is that there are no enabled transitions with this scope in the hierarchy. These two scenarios are formalised as:

```
(let (trsThisLevel, rest) := thisScope trs  in
   (oneEnabledTransIsTaken s trsThisLevel step /\
      noTransAreTaken rest step)
   \/
   (noEnabledTrans s trsThisLevel step /\
     (every
      (\sub.TransCondAux sub (scopeWithin rest (stateName sub)) step)
      (stateSubstates s)))));
```

The first scenario is described by the predicate `oneEnabledTransIsTaken`. If this scenario holds then no transitions with lower scope in the hierarchy can be taken. The predicate `noTransAreTaken` contrains all the Boolean variables associated with transitions lower in the hierarchy to be false.

In the second scenario, there are no enabled transitions with this scope. Then the transition condition must hold in each of the substatecharts of this state for the transitions that fall within the scopes of the respective substatecharts. The function `scopeWithin` determines the transitions that fall within each of the relevent substatecharts. Every transition falls within the scope of one substatechart. Only one substatechart will contain any transitions that can be taken because the system can only be in one substate of an OR-state.

The predicate `oneEnabledTransIsTaken` takes as arguments the statechart structure at this level in the hierarchy and the transitions with this scope. It has the definition:

```
(:stateName,:transName)
oneEnabledTransIsTaken s (trlist:(trans)list) step :=
    oneTransTaken trlist step  /\
    every
    (\tr. ~(TransTakenInt tr step)  \/
           (Enabled s tr (ExtPrev step) /\
           (stateChange s tr step)))
    trlist;
```

This predicate first ensures that only one transition of the set is taken since only one transition can be taken within an OR-state using a function `oneTransTaken`, which places no priority on which transition (with this scope) is chosen if multiple ones are enabled. Second, `oneEnabledtransIsTaken` ensures that if a transition is taken, it must be enabled and it constrains the set of states in the next configurations using the predicate `stateChange`.

The effect of taking a transition on the status of the states is to exit its source state and enter its destination state. However, we must also ensure that the status of any other states within the scope of this transition remain the same. The status of all states is determined by the basic states. The predicate `stateChange` has the following definition:

```
(:stateName,:transName)
stateChange s (tr:(stateName,transName)trans) step :=
   let allBasicStatesInScope :=
        basicStatesExited s (stateName s) in
    let basicStatesEnt :=
        basicStatesEntered s (transDest tr) in
    every
       (\stn. COND (stn member basicStatesEnt)
                    (InBasicState stn (ExtNext step))
                    (~(InBasicState stn (ExtNext step))))
          allBasicStatesInScope;
```

A list of all the states within this scope is produced by applying the function `basicStatesExited` to the state name of the root of the statechart structure. If a basic state is not entered then the function `InBasicState` returns false for that state name in the next configuration.

Returning to the second scenario for an OR-state where there are no enabled transitions with this scope, the predicate `noEnabledTrans` is defined using `Enabled`. It is satisfied if there are no enabled transitions and no transitions are taken at this level.

```
(:stateName,:transName)
noEnabledTrans s (trlist:(trans)list) step :=
   (every (\tr. ~(Enabled s tr (ExtPrev step))) trlist) /\
   (every (\tr. ~(TransTaken tr step)) trlist);
```

AND-states

If the root state is an AND-state, there are no transitions with this scope because transitions cannot go between components of an AND-state. Transitions can be followed within each substate of an AND-state. Therefore, the set of transitions is partitioned by their scopes into the substates of the AND-state. `TransStateCondAux` must then hold for each of these substates:

```
(every
  (\sub. TransStateCondAux sub (scopeWithin trs (stateName sub)) step)
  (stateSubstates s))
```

Basic states

The final and simplest case is if the root state is a basic state. The transition set must be empty since the scope of a transition must always be an ancestor of its source and destination states and a basic state is not an ancestor of any state. In this case the status of the basic state does not change.

```
InBasicState (stateName s) (ExtNext step) EQ
        InBasicState (stateName s) (ExtPrev step)
```

Definition of TransStateCondAux

Putting together the three cases for the different types of states that can be at the root of a statechart results in the `TransStateCondAux` definition found in Figure 5.4. If no transitions are enabled in a step, then the function `TransTaken` will be false for all transitions and there will be no change in the status of the basic states. However, time is advanced and the status of events could change.

Many variants of statecharts have more complicated semantics with microsteps and history states. We favour the simplicity of these semantics for understanding. However, our semantics could be extended with the method of Anderson et al. [ABB$^+$96] where a fixed point of microsteps is reached before considering new inputs, to create other statechart variants for use in the framework. The meaning of history states could be captured through auxiliary history names as is done for events.

```
TransStateCondAux s (trs:(trans # scope)list) step :=
   COND (IsBasicState s)
     /* Basic State */
   (inBasicState (stateName s) (Next step) EQ
       InBasicState (stateName s) (Prev step))

   (COND (isAndState s)
     /* AND-State */
     (every
       (\sub. TransStateCondAux
                 sub (scopeWithin trs (stateName sub)) step)
       (stateSubstates s))

     /* OR-State */
     (let (trsThisLevel,rest) := thisScope trs  in
     (oneEnabledTransIsTaken s trsThisLevel step /\
      (every (\tr.~(TransTaken tr (Prev step))) rest))
      \/
     (noEnabledTrans s trsThisLevel (Prev step) /\
     (every
         (\sub.TransStateCondAux
                 sub (scopeWithin rest (stateName sub)) step)
         (stateSubstates s)))));
```

Figure 5.4: Definition of TransStateCondAux

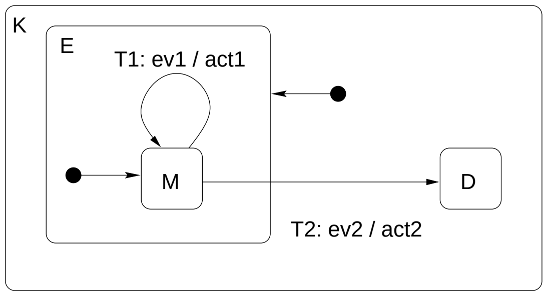

Figure 5.5: Example statechart

Example of evaluating the transition state condition

The simple example statechart of Figure 5.5 is used to illustrate the meaning of the transition state condition. The result of evaluating the predicate `TransStateCond` for this statechart is presented in Figure 5.6 for the step `(cf, cf')`. This tiny statechart has two transitions with the same source state (`M`) but different scopes. Transition `T2` has priority over `T1`. There are three cases that could occur. In the first case, transition `T2` is taken. Lines 2 through 5 of Figure 5.6 say that in this case transition `T2` must be enabled and the appropriate state change must occur. Line 3 is the result of the function `Enabled` and line 5 is the result of `stateChange`. Line 6 says that `T1` is not taken in this case.

The second case, where `T1` is taken, begins at line 7. Lines 7 and 8 say that `T2` is not enabled and is not taken. Lines 9 through 12 check whether `T1` can be taken and constrain what happens if it is taken. Because `T1` is within the scope of state `E`, line 12 only has to constrain the value of `InBasicState M` for the next configuration. Line 16 ensures the status of state `D` does not change.

In the third case, neither transition is enabled and therefore no transitions are taken. This case is a subcase of the previous one where `T2` was not taken. Lines 13 and 14 say that `T1` is not enabled and therefore not taken. Line 15 states that the status of state `M` does not change.

```
( TransTaken T2 cf /\
  (~ (TransTaken T2 cf) \/
       (InBasicState M cf /\ EventOccurred (ev2 T2 []) cf))
  /\
  ~ InBasicState M cf' /\ InBasicState D cf'
  /\ ~ TransTaken T1 cf) \/
 ( (~ (InBasicState M cf /\ EventOccurred (ev2 T2 []) cf))
    /\ ~ (TransTaken T2 cf) /\
    TransTaken T1 cf /\
    (~ (TransTaken T1 cf) \/
       ( InBasicState M cf /\ EventOccurred (ev1 T1 []) cf))
    /\ InBasicState M cf')    \/
( (~ (InBasicState M cf /\ EventOccurred (ev1 T1 []) cf))
   /\ ~ (TransTaken T1 cf) /\
   (InBasicState M cf' EQ InBasicState M cf) /\
   (InBasicState D cf' EQ InBasicState D cf))
```

Figure 5.6: TransStateCond evaluated for statechart in Figure 5.5

5.8.3 Name condition

An action is a list of three-tuples that each consist of an identifier for a name to be changed, the constraint that the value of the name changes as specified by the action, and a constraint that the value of the name does not change for when the transition is not taken.

The possibility of race conditions, where multiple transitions modify the value of the same name, means that the actions of transitions cannot be considered independently. The constraints on the results of actions are

- if a transition is taken, then the next configuration will includes the results of the transition's actions except where conflicts occur among the actions of the set of transitions taken
- if more than one modification is made to the same name (i.e., a conflict occurs) then exactly one of these modifications will be true in the next configuration
- if the value of a name is not modified by any transition chosen in a step, then the

name retains its previous value, provided that there is a transition that can modify its value (i.e., it is a controlled quantity [PM95])

These constraints are defined by the "name" condition because they concern the names of the system.

Only names under the system's control must keep their previous value if they are not modified (assignable names). Constants representing external names and events might not retain their values between steps.

The first step in defining the name condition is to partition modifications from all the transitions by the name that they modify. The function `GroupModifications` carries out this task for a transition list. For each name, it returns a tuple structure that consists of a predicate that states that the name does not change, and a list of transition and modification pairs, where all of the modifications apply to the same name. This list describes possible race conditions. The following accessor functions operate on this record structure:

- `nameNoChange` : returns the predicate stating that the name does not change when applied to a step
- `nameModList` : returns the list of transition and modification pairs
- `nameTransList` : returns the list of transitions that effect the name

Using this structure, the name condition has two cases. The first case is that no transitions that affect the name are taken and therefore its value does not change. The second case is that there is some transition that is taken and its action affects the value of the name. The definition of the name condition is:

```
(:stateName,:transName)
NameCond s (trs:((stateName,transName)trans)list) step :=
  every (\namerec.
   /* Case 1 */
   (every
      (\tr. ~(TransTaken tr step)) (nameTransList namerec)
    /\
    nameNoChange namerec (ExtPrev step, ExtNext step))
   \/
   /* Case 2 */
   (any
      (\(tr,md) . TransTaken tr step /\
                  md (ExtPrev step, ExtNext step))
      (nameModList namerec)))
  (groupedModifications trs);
```

No sequence is assumed among multiple actions on a transition so conflicts could occur within the actions of one transition as well as between multiple transitions.

This section has presented how the CoreSc notation resolves race conditions. Other notations might have other schemes for how race conditions are resolved. If the sets of names modified by the parts of the specification written in different notations are disjoint, multiple race condition resolution schemes can be used and the meaning of the specification is still well-defined. If the sets of names overlap, then an inconsistent specification (i.e., a specification that is not satisfiable) may result.

5.8.4 Event condition

The event notation used for a statechart specification may have its own constraints on the behaviour of the system. These constraints are defined in the "update" field of the events. The event condition presented here ensures that these constraints are maintained for every transition in the statechart. The function `EventUpdate` is defined as the event accessor function `Update` with the true flag to indicate the event is a top-level event. The event conditions has the following definition:

```
(:stateName,:transName)
EventCond s (trs:((stateName,transName)trans)list) step :=
     every
       (\tr.EventUpdate (transEvent tr) (transLabel tr) [] step)
       trs;
```

5.8.5 Initial condition

In the initial configuration of a system specified using a statechart, the set of basic states that the system is currently is the default basic states. The following predicate gives this constraint:

```
(:stateName,:transName)
InitStates :(stateName,transName)sc_struct -> bool;
```

The events of the transitions of the statechart may also require initialisation.

5.9 ScExpr

The keyword `InState` is an expression that can be used in a specification in CoreSc. Its meaning is defined in terms of definitions presented earlier for the transition state condition:

```
(:stateName,:transName)
InState stName (s:(stateName,transName)sc_struct) := Instate stName s ;
```

It has the type `bool`.

5.10 ScEvent

The events `En` and `Ex` for entering and exiting states are events that can be used in a specification in CoreSc. Two uninterpreted functions, `EnJustOccurred` and `EvJustOccurred`, are used to capture the history of these events. These functions have the following type signatures:

```
(:stateName) EnJustOccurred :stateName -> bool;
(:stateName) ExJustOccurred :stateName -> bool;
```

The components of the meaning of these events are defined using these functions. An `En` event occurred in the previous step if the function `EnJustOccurred` applied to the state name returns the value true:

```
(:stateName)
EnOccurred (stn:stateName) cf := EnJustOccurred stn cf;
```

An `En` event occurs in a step if a transition is taken that enters the state that is its argument:

```
(:stateName,:transName)
EnOccurs stn (s:(stateName,transName)sc_struct)
  (step:(transName)ext_step) :=
    let alltrans := transInState s in
    any
    (\t. TransTaken t step /\ (stn member (statesEntered s t)))
    alltrans ;
```

Because this function depends on both the previous and next configurations of a step, the uninterpreted constant `EnJustOccurred` is necessary to record this history in the next configuration. The set of transitions that are taken must be used to determine the status of this event rather than just the status of the states because of looping transitions

To update the status of the `En` event, the value of the `EnJustOccurred` function in the next configuration captures the value of `EnOccurs` for this step:

```
(:stateName,:transName)
EnUpdate stn (s:(stateName,transName)sc_struct) (flag:BOOL)
             (step:(transName)ext_step) :=
    ~flag \/
    (EnJustOccurred stn (ExtNext step) EQ EnOccurs stn s step);
```

The constraint on the `EnJustOccurred` constant need only be enforced if it has not previously been captured (i.e., `flag` is true). Otherwise, its history will be captured elsewhere.

An `En` event occurs at initialisation if the system starts in the state in the initial configuration.

All of the components of the `En` event are captured in the five-tuple in the following definition:

```
(:stateName,:transName)
En stn (s:(stateName,transName)sc_struct) (lab:transName) (p:path) :=
    (EnOccurred stn,
     EnUpdate stn s,
     EnOccurs stn s,
     EnInit stn s,
     EnOccursAtInit stn s);
```

The definitions for `EnInit` and `EnOccursAtInit` along with the semantic definitions for the event `Ex` can be found in Appendix J.

5.11 CommAction

Section 4.14 introduced the keywords `SendData` and `Send`, which allow a state to send directed communication to another state.

This action notation require the introduction of two uninterpreted functions to provide names to pass information between a sender and a receiver. The uninterpreted function `Msg` has the type signature:

```
(:stateName,:msg) Msg : stateName -> stateName -> msg -> bool;
```

It has the value true when the statechart of the first state name is sending a particular message to the statechart of the second state name. The data of the transaction is captured in the value of the function `Data`:

```
(:stateName,:msg,:ty)
Data: stateName -> stateName -> msg ->(ty)exp;
```

As an action, `SendData` modifies two names: the `Msg` function for the appropriate argument and the `Data` function for the same argument. Therefore, it returns a list of two modifications each consisting of an identifier for the name (in this case the name is the application of a function to arguments), the condition that states what happens if the value of the name changes and the condition describing the scenario where it does not change.

A `SendData` action sets the value of the `Msg` function for appropriate arguments to true in the next configuration. If it does not occur (and no other transitions modify the value), the value is set to false. It also sets the value of `Data` to the value of the data argument. If the action does not occur, there are no restrictions on the value of `Data` for these arguments. The complete definition of `SendData` is:

```
(:stateName,:transName,:msg,:ty)
SendData (s:(stateName,transName)sc_struct)
     (dest:stateName) (ms:msg) (data:(ty)exp) :=
  [(NAME (Msg (stateName s) dest ms),
    \step.(Msg (stateName s) dest ms (Next step)) EQ T,
    \step.(Msg (StateName s) dest ms (Next step)) EQ F);
   (NAME (Data (stateName s) dest ms),
    \step.
      (Data (StateName s) dest ms (Next step)) EQ (data (Prev step)),
    \step.T)];
```

The `Send` keyword is defined as only the first of the two modifications above. The complete set of semantic definitions for CommAction can be found in Appendix K.

5.12 CommEvent

Section 4.15 introduced the keywords `Receive` and `ReceiveData` as events for simple directed communication between states in a statechart specification. This section gives their meaning in terms of the five components of an event.

The `ReceiveData` event occurred in the previous step if the value of the `Msg` function is true for the appropriate addresses (state names) and messages:

```
(:stateName,:transName,:msg)
ReceiveDataOccurred
        (src:stateName) (dest:stateName) (ms:msg) (cf:config) :=
   Msg src dest ms cf;
```

Updating the status of the `Receive` event involves setting the value of the data argument to equal the value of the `Data` function. This update is done whether the event occurs or not. Therefore, the value of the data is only valid in the configuration following the step where the event occurs.

```
(:stateName,:transName,:msg,:ty)
ReceiveDataUpdate (src:stateName) (dest:stateName) (ms:msg)
    (data:(ty)exp) (flag:BOOL) (step:(transName)ext_step) :=
  data (ExtNext step) EQ Data src dest ms (ExtNext step);
```

The `ReceiveData` event occurs in a step if the value of the `Msg` function is true in the next configuration. At initialisation, no messages have been sent.

The `Receive` keyword has the same meaning as the `ReceiveData` keyword except there are no constraints in updating the status of the event since `Receive` does not include a data value. Appendix K contains the complete definitions.

5.13 Reasoning about the semantics

The semantic definitions presented in this chapter are packaged embeddings of the notations. The main benefit of this approach for the framework is the ease with which multiple notations can be used in one specification. The semantic definitions can be thought of as a translator to predicate logic, written in a functional programming language.

Another advantage of writing the meaning of a notation in higher-order logic is that it is possible to reason about the semantics themselves. Reasoning is often done using a deep embedding, which makes it possible to prove properties that hold over all uses of a notation. With a shallow embedding, proofs of a property of the semantics can only be phrased over all possible notations for a join point. For example, a property proven about statecharts must be over all possible event notations that have currently been described and could be added in the future because notations are combined based on type consistency. Thus goals must be phrased over all possible elements of the appropriate type.

For example, we hypothesise the following property of statecharts: if the root state is an AND-state and each component state of the state has the properties

- no transition within a component is triggered in whole or in part by the event of entering or exiting a state that is not within the component
- the sets of names modified by actions for each component are disjoint

then:

`Sc (AndState` $stname$ `[`st_1`;`st_2`;` $\ldots st_n$`])` $step \equiv$

  `Sc` st_1 $step$ `/\` `Sc` st_2 $step$ `/\` $\ldots$ `/\` `Sc` st_n $step$

Italic font is used to represent variables in the above expression. This property should be true for all possible values of the variables. In a statechart, no transitions can have scope outside the root state. Also, a well-formedness constraint ensures that transitions do not cross AND-state boundaries so the sets of transitions for each component are disjoint. We sketch a proof of this property in two parts.

The first part shows that:

`ScAux (AndState` $stname$ `[`st_1 `;`st_2`;` $\ldots st_n$`])` $transList$ $extStep \equiv$

  `ScAux` st_1 $transList_1$ $extStep_1$ `/\`

  `ScAux` st_2 $transList_2$ $extStep_2$ `/\`

  $\ldots$ `ScAux` st_n $transList_n$ $extStep_n$

The set of transitions within st_x is $transList_x$, and $extStep_x$ is the step that includes the mapping from transition labels within the component x to Boolean variables.

The second part of the proof shows that the existential quantification of the Boolean flags associated with the transitions can be moved in to apply just to the component containing that transition. The following logical rule is applicable:

$$x \notin FreeVarsB \Longrightarrow \exists x.A \wedge B \equiv (\exists x.A) \wedge B$$

The first part of the proof has to examine each part of the definition of `ScAux`. The transition state condition is decomposed into a conjunction because the root state is an

AND-state and the transitions are allocated to the substates of the AND-state based on their scope. Therefore, the property is valid for the transition state condition.

The name condition resolves race conditions. Thus the property is dependent on no race conditions existing between components. The name condition can then be partitioned based on the names. If the modified names of each component are disjoint, then the property is valid for the name condition.

The event condition is partitioned based on the transitions. If multiple transitions (even in different components) are triggered in whole or part by the same event, the same update of the event (i.e., incrementing or reseting of the counter) will be required as long as the update is based on information that can be seen by all components. Because entering and exiting events are dependent on the status of transitions, when the state of interest to an `En` or `Ev` event is outside the AND-state component the property will not hold. The part of the proof for the event condition would proceed by structural induction on the event of a transition.

The heating system satisfies the assumptions of this property of the CoreSc semantics, therefore it can be broken apart into its component statecharts. Breaking apart a statechart helps to deal with a large configuration space (because existential quantification is carried out earlier) and will be used in Chapter 7.

5.14 Summary

This chapter has presented our approach to determining the meaning of a notation and associating meaning with representation. The meaning of a notation is provided by operational semantics. The meaning is associated with the notational style using packaged embeddings. The semantics for the example notational styles of the previous chapter were defined.

The type signatures for the four categories of notations were presented. A model results in a next configuration relation, which is used in configuration space exploration analysis.

By using a packaged embedding of the notational styles, our approach is extensible. As described in the previous chapter, type checking regulates the possible combinations. Specifications can use higher-order logic as a notation, which means that uninterpreted constants are allowed and parameterisation of components in particular notations can be achieved without any extensions to the notation or its semantics.

Evaluation of the semantic definitions results in the meaning of the specification in "raw higher-order logic". The next chapter describes a technique, called symbolic functional evaluation, that can be used to evaluate the semantics. Direct use of the semantics ensures that all forms of analysis understand the same meaning for the specification.

The semantics are defined operationally for the sake of the desired type of analysis. Another potential use for them is to compare them to other versions of the semantics that are useful for other types of analysis. The semantics can also be used to establish properties of particular notations that can help in analysis. We demonstrated a derivation of a property of statecharts that will be used for analysis of the ATN example.

Once the semantics for a particular notational style have been written (likely by a formal methods expert), specifiers need not examine them except in cases where they question their intuitive understanding of the meaning of the notation. While initial investment is required in their preparation, the semantics are repeatedly used in analysis of multiple specifications.

Chapter 6

Symbolic Functional Evaluation (SFE)

The categories of notations systematise the way that notational styles can fit together in a specification (Chapter 4). The packaging of semantic definitions with the keywords of notational styles makes the framework extensible (Chapter 5). In this chapter we describe our solution to the seventh sub-problem introduced in Chapter 1, namely determining the meaning of a specification (Section 1.2.7). Our solution to this problem is the key to achieving rigour for our framework outside of a theorem proving environment.

Rigour is achieved by the direct use of the semantic definitions to determine the meaning of the specification. Expanding the definitions of the specification, the definitions of the keywords of the notational style, and the internal semantic definitions, results in an expression in "raw higher-order logic" (RHOL), i.e., an expression in terms of only the built-in constants and any uninterpreted constants that have been introduced. The expression in RHOL is semantically *equivalent* to the original specification. This chapter presents our technique for carrying out definitional expansion and normal order reduction called *symbolic functional evaluation* (SFE).

The key contributions found in this chapter are

- adaptation of algorithms from functional programming language theory to carry out definitional expansion and beta-reduction in higher-order logic
- identification of distinct levels of evaluation
- a method that preserves the "unevaluated" version of expressions
- integration of a selection of inference rules for higher-order logic that can be applied automatically

Section 6.1 describes how symbolic functional evaluation fits within the context of the framework and contrasts it to translation approaches. Section 6.2 motivates making a distinction between uninterpreted constants and other lambda calculus variables, and having *evaluation levels* for expressions in higher-order logic. Because the evaluation process need not always go as far as producing RHOL to be of use in automated techniques, modes of SFE produce expressions at different levels of evaluation. Section 6.3 describes the evaluation levels. Section 6.4 presents our algorithm for symbolic functional evaluation. Section 6.5 describes the data structures used to represent expressions in this algorithm and how the "unevaluated" form of the expression can be remembered for analysis output. Section 6.6 describes the small optimisation made in the semantics of CoreSc for evaluation. Section 6.7 discusses evaluation of the built-in constants. Finally, in Section 6.9, we consider how selected inference rules can be applied during symbolic functional evaluation to expose more information about the specification to analysis techniques.

Symbolic functional evaluation is applicable to any expression in higher-order logic. Therefore, it could have uses outside of the framework presented in this dissertation. For example, definitional expansion and beta-reduction are commonly used in theorem proving. SFE can be used to support the symbolic simulation style of proof that has been used in theorem proving approaches to the formal verification of digital circuits.

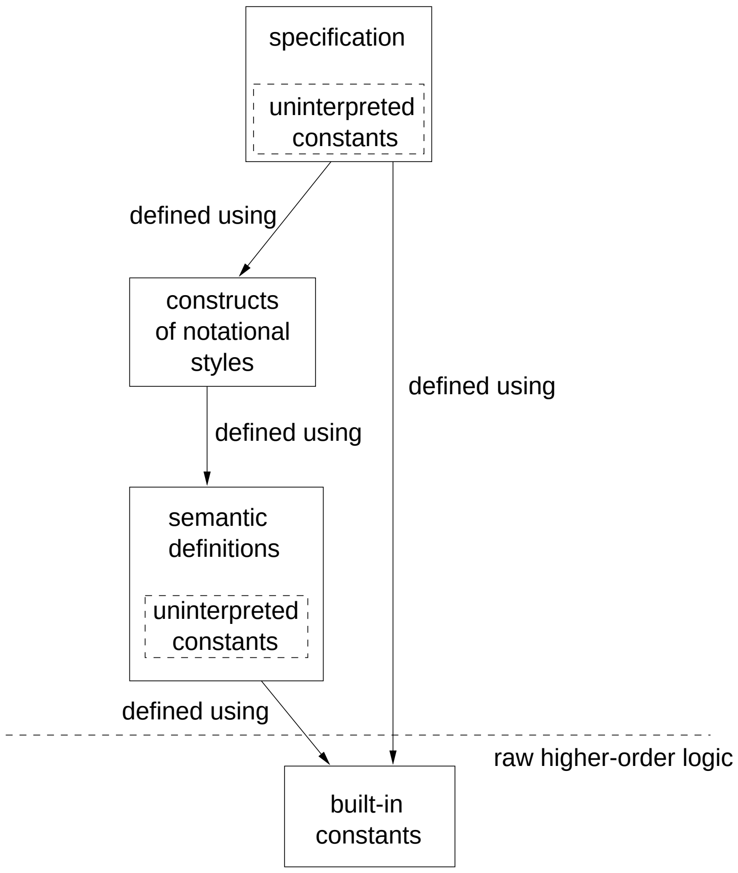

Figure 6.1: Specification within the framework

6.1 Purpose within the framework

Previous approaches to linking automated analysis to requirements specification notations have often involved translation from the original notation to the input notation of the analysis tool. The code of the translator can be considered an operational semantics. However, unless the original specification uses only elements of finite types, and does not use uninterpreted constants, this translation process often includes abstraction to produce a form of the specification in the input notation of the analysis tool. To the extent that the translator departs from the semantics of the notation, translation can introduce inconsistencies between the original specification and the output of the translator.

Figure 6.1 describes the relationship between the specification and the various elements of the framework. Our approach is to make direct use of definitions in logic to carry out the "translation" step. This approach can be thought of as writing a functional program to carry out the translation. The output of completely evaluating the specification is raw higher-order logic in that it includes only the built-in uninterpreted constants and any introduced uninterpreted constants. We differ from previous approaches in two important ways. First, the result of the evaluation is semantically equivalent to the original specification. No information is lost in this process. Second, the definitions are written in logic and therefore serve as a formal description of the semantics of the notational style. The semantic functions are a specification for a translator to RHOL. SFE evaluates these definitions. Evaluating the semantic definitions in logic ensures that the "translator" exactly matches its specification. They are the same. Rather than writing code for a tool that meets the specification of the translator, we just use the specification directly.

Definitional expansion is traditionally accomplished in theorem provers using rewriting techniques. These techniques use theorems of equality to replace an instance of an expression with its equal. General rewriting involves several steps that are unnecessary for definition expansion. First, theorems of equality must be proven truths in the logic. Second, and most important, general rewriting includes a step called unification, which searches for matches between expressions and theorems of equality. The use of a defined constant is easily matched to its definition by its name and does not require general unification. Third, rewriting often provides and requires more precise control of the specific definitions that are expanded. Symbolic functional evaluation requires none of these steps and therefore can avoid the complexities associated with more general rewriting. However SFE is not as general as rewriting.

6.2 Uninterpreted constants

Functional programming language theory provides algorithms and data structures to evaluate efficiently functional programs. Functional programs are essentially the lambda calculus without free variables. Uninterpreted constants used in both the specification and in the semantic definitions (such as `InBasicState`) do not have definitions. These constants are free variables in the lambda calculus. We extend Peyton Jones' algorithm found on Figure 3.2 on page 59 to include a case for variables. If the tip of the expression is a variable, it is recombined with its arguments as a function application. Substitution must also check for name capture. These extensions by themselves make no distinction between uninterpreted constants and bound variables. Functional programming languages already distinguish constructors from other variables. From this point on, we will use only the terminology for expressions from higher-order logic, which distinguishes constants and variables, both called variables in the lambda calculus. In this terminology, an expression is a constant, variable, abstraction, or application. Constants include uninterpreted constants and constructors. Quantification is the application of the higher-order functions FORALL or EXISTS to a lambda abstraction.

When evaluating expressions that include uninterpreted constants, there are two special considerations.

First, many automated analysis techniques work on finite specifications. Therefore, a finite abstraction of a specification in the framework may have to be created for analysis. One simple method for creating an abstraction of the specification is to consider any fragment of an expression without a logical connector at its tip as one Boolean variable. This method abstracts away any details about the specification within these fragments. Consequently when using this approach, it is sufficient to reach the weak head normal form of expressions that do not have logical connectors at their tips. We can tell when an expression will never evaluate to an expression with a logical connector at its tip when evaluation encounters an uninterpreted constant at the tip of an application. At this point, further evaluation exposes details about the specification that will be lost in the abstraction for analysis. Therefore, we do not always need to reduce all redexes and there is value in stopping early for efficiency.

The second special consideration is that unlike variables bound within some scope (and therefore free within another scope), uninterpreted constants will never change in the expression. The most time consuming part of the evaluation algorithm is the substitution of arguments for parameters. A subexpression that has uninterpreted constants but no variables will never change in a substitution, so the algorithm can save the effort of walking over this subexpression in substitution.

The first point motivates the desire to have modes of SFE whose goals are to evaluate an expression to the point where it is known to be in certain forms similar to weak head normal form and normal form (Section 3.2), and another level in between these two forms. The second point results in an optimisation in the substitution algorithm.

6.3 Levels of evaluation

Section 3.2 on page 52 presented definitions of normal form and weak head normal form of an expression. These forms are not exclusive categories, i.e., an expression in normal form is also in weak head normal form.

SFE is an evaluation process for substituting definition bodies for defined constants and reducing the redexes in an expression in higher-order logic. As described in the previous section, it is not always necessary to reduce all the redexes in an expression. Therefore, we want to have early stopping points in the process for efficiency. Two useful early stopping points are reaching the point at which two expressions can be compared (called the point of distinction), and reaching the point of being able to apply certain rewrite rules (discussed in Section 6.9).

To define precisely these stopping points, a tag is associated with each expression indicating how much evaluation has been carried out on that expression. This section defines the possible levels of evaluation for expressions. They are, in order: NOT_EVAL (not evaluated), PD_EVAL (evaluated to the point of distinction), RW_EVAL (evaluated for rewrite simplification), SYM_EVAL (evaluated with symbols), and FULL_EVAL (fully evaluated). At any time, each expression has exactly one tag. Initially every expression has the tag NOT_EVAL meaning it has not yet had any evaluation carried out on it. Evaluation of an expression can have two effects:

- It can produce a new equivalent expression with a different tag to indicate the new expression is evaluated to some extent.
- It can change the tag of the expression indicating it has been "examined" and can be categorised as some level of evaluation without changing the expression.

If a new expression is produced it is inserted in place of the old expression. Therefore, when we define levels of evaluation using the phrase "an argument in a function application has been evaluated to a certain level of evaluation", it should be understood to describe either of the two effects of evaluating an expression.

Expressions in higher-order logic are: 1) applications, 2) abstractions, 3) variables, and 4) constants. We subdivide the category of constants into: 4a) uninterpreted constants, 4b) defined constants, 4c) constructors, and 4d) built-in constants.

The levels of evaluation are distinguished mainly with respect to the extent to which the arguments of applications of uninterpreted constants, variables and constructors are evaluated. Defined constants and abstractions in redexes are eliminated in evaluation. The levels apply to expressions that do not have built-in logical constants, such as `AND`, at their tip, which will be discussed subsequently.

We define five levels of evaluation (and tags) in order. A BNF-like notation is used to define the set of tagged expressions that belong in each level. This BNF does not describe a notation for parsing. Rather, it is used as a compact way to describe the results of evaluation. A BNF is used because the tag (or level) of an expression is determined by the tags of its subexpressions. The terminals of this BNF are not described. They are the constants and variables of a particular specification. Section 6.4 shows how turning the BNF "upside-down" (i.e., starting from an expression rather than starting from the components of an expression) results in an algorithm that evaluates an expression to the level of a desired tag.

As will be seen in the definitions of expressions with the tags SYM_EVAL and FULL_EVAL, uninterpreted constants, and variables by themselves have the tag SYM_EVAL, and constructors have the tag FULL_EVAL once examined. Because these expressions have particular significance at the tip of a function application, the BNF uses their names as non-terminals.

The name of the tag is used as a non-terminal in the BNF to represent expressions that have that tag. Juxtaposition represents applications. The phrase "\ variable . *exp*" represents an abstraction. Items in brackets separated by "|" mean that one of these items must be chosen. A "*" after an item means zero or more of these items. A "+" after an item means one of more of these items. A series of BNF non-terminals called "level$x$up" is used to describe all the levels including level x and above.

6.3.1 Not evaluated

An expression with the tag NOT_EVAL has not had any evaluation carried out on it. Uninterpreted constants, variables, and constructors that have not been examined have the tag NOT_EVAL. The application of an expression with the tag NOT_EVAL to any other expression is considered NOT_EVAL.

```
NOT_EVAL  ::=  NOT_EVAL level1up+
            |  \ NOT_EVAL . level1up
level1up  ::=  ( NOT_EVAL | level2up )
```

6.3.2 Evaluated to the point of distinction

Expressions at the second level of evaluation have the tag PD_EVAL. By definition, expressions at this level are either applications or abstractions. The point of distinction means the expression can be compared (for example, using equality) to other expressions. If it is an application with a constructor at its tip, it can be determined to be not equal to another expression with a different constructor (of the same type) at its tip. If the application has an uninterpreted constant or variable at its tip, then it can never be distinguished from another expression[1]. If the expression is an abstraction, the abstraction can never be reduced.

For a function application to have the tag PD_EVAL, its tip must be an uninterpreted constant, a constructor or a variable, and at least one of its arguments has not had any evaluation carried out on it. If the expression is an abstraction, it cannot be reduced further and has the tag PD_EVAL if its body is not evaluated as much as possible (i.e., its body has a tag other than SYM_EVAL or FULL_EVAL).

```
      tip   ::=  (uninterpreted_constant | constructor | variable)
  PD_EVAL   :=   tip level1up* NOT_EVAL level1up*
             |   \ NOT_EVAL . ( PD_EVAL | RW_EVAL | NOT_EVAL)
  level2up  ::=  ( PD_EVAL | level3up )
```

This level is roughly equivalent to weak head normal form except that irreducible applications of built-in constants such as `AND` are not stopping points when found at the tip of the application.

For example, the expression `(f1 ((\x.x) 1)) AND (f2 k)` is at the level of PD_EVAL if `f1` and `f2` have been determined to be uninterpreted constants and at least one of `((\x.x) 1)` and `k` is NOT_EVAL.

[1]An uninterpreted constant is a distinct but unknown value, therefore it can never be determined to be not equal to another value using evaluation.

6.3.3 Evaluated for rewrite simplification

To apply the rewrite rules described in Section 6.9, the expression must be evaluated to the point of being able to determine their applicability. We have found determining this applicability requires carrying out the evaluation one step further than the PD_EVAL level in that all arguments to the function must be evaluated at least to the point of distinction. Expressions with the tag RW_EVAL include function applications in the range between PD_EVAL and SYM_EVAL. Therefore, at least one argument does not have the tag SYM_EVAL or FULL_EVAL, and no arguments have the tag NOT_EVAL.

```
RW_EVAL  ::=  tip level2up* (PD_EVAL | RW_EVAL) level2up*
level3up ::=  ( RW_EVAL | level4up )
```

6.3.4 Symbolically evaluated

An expression with the tag SYM_EVAL is in normal form where no redexes remain, and the expression contains uninterpreted constants or variables.

```
SYM_EVAL ::=  (uninterpreted_constant | variable) level4up*
           |  constructor level4up* SYM_EVAL level4up*
           |  \ variable . ( SYM_EVAL | FULL_EVAL)
level4up ::=  (SYM_EVAL | FULL_EVAL)
```

All expressions with terminating reduction sequences (i.e., ones that have a normal form) that are defined in terms of uninterpreted constants can reach this level of evaluation. An expression at this level cannot be evaluated any further.

6.3.5 Fully evaluated

In a functional program, all expression with a normal form can be reduced to the point of containing only constructors. Expressions in higher-order logic that contain only constructors have the tag FULL_EVAL assigned. Intuitively, an expression that can be reduced to this level is fully executable. All expressions with a terminating normal order reduction sequence will reach one of the levels SYM_EVAL or FULL_EVAL.

$$\mathsf{FULL\_EVAL} \quad ::= \quad \text{constructor } \mathsf{FULL\_EVAL}^*$$

For example, the expression `NIL`, once it has been examined to be a constructor is considered fully evaluated. The expression `CONS 1 NIL` is also considered to be fully evaluated once all of the subexpressions `CONS`, `1`, and `NIL` have been examined to be constructors.

6.3.6 Special cases

There are three special cases to consider in symbolic functional evaluation. Constants defined by non-pattern matching definitions can only appear at the tip of applications (or subexpressions) that have not been evaluated, or within arguments to expressions with the tags PD_EVAL or RW_EVAL. A constant defined by a pattern matching definition is a special case. An expression that is the application of a constant defined by a pattern matching definition must always have the first argument evaluated to the point of distinction to determine if the argument matches a case of the definition. If the argument does match a case, evaluation proceeds by substituting the arguments into the body of the pattern's definition as with simpler definitions. If a match cannot be found or if the argument does not have a constructor at the tip, then evaluation proceeds as if the tip is an uninterpreted constant.

Some built-in constants have special significance. Non-lifted conjunction, disjunction and negation are never stopping points for evaluation when they are at the tip of an expression. The level of evaluation of such an expression is defined as the lowest level of its arguments if they cannot be fully evaluated. The same is true for quantifiers over Boolean variables. Over non-Boolean variables, quantifiers are considered as uninterpreted constants.

The constants for the non-lifted arithmetic operators must have at least one of their arguments evaluated to the point of distinction to determine if the expression can be reduced. If reduction is not possible or there are not enough arguments for the operator, they are treated as uninterpreted constants.

6.4 Evaluation algorithm

The goal of symbolic functional evaluaton is to evaluate expressions in higher-order logic to the point where the expression falls into a particular level of evaluation. The user chooses the mode for SFE usually based on the minimum mode that is suitable for the type of analysis to be carried out.

For an input expression that falls into any level, there are three modes for SFE:

evaluate: produce an expression that is either fully evaluated or symbolically evaluated.

evaluate for rewrite simplification: produce an expression that is either fully evaluated, symbolically evaluated, or evaluated for rewrite simplification.

evaluate to the point of distinction: produce an expression that is in any level except not evaluated.

The above classification applies only to terminating evaluation sequences. No special provisions have been provided to check for non-termination.

We now present an algorithm that implements symbolic functional evaluation. It carries out lazy evaluation which means arguments to functions are not evaluated until they are used, and evaluation is carried out in place. This normal order reduction algorithm extends the spine unwinding algorithm presented in Section 3.2 on page 59 to deal with uninterpreted constants and variables.

Figure 6.2 gives the top-level algorithm. It is called initially with an expression, an empty argument list, and the desired level of evaluation of the expression. This algorithm

```
expression EvalExpression(expression *exp, expressionlist arglist, flag mode)

/* stopping point ? */
if (arglist==NULL) and (EvalLevel(exp) >= mode) then
    return exp
else if (mode == PD_EVAL) AND (EvalLevel(exp) >= mode) then
    Recombine(exp,arglist,NOT_EVAL)
endif

switch (formof(exp))

case VARIABLE (v) :
    SetEvalLevel(exp,SYM_EVAL)
    if (arglist) then
        return Recombine(exp, arglist, mode)
    else
        return exp
    endif

case APPLICATION (f a) :
    newarglist = add a to beginning of arglist
    newexp = EvalExpression(f, newarglist,mode)
    if (arglist==NULL) then
        ReplaceExpr(exp, newexp)
    endif
    return newexp

case ABSTRACTION (parem exp):
    (leftover_args,newexp) = Substitute(exp, parem, arglist)
    return EvalExpression(newexp,leftover_args,mode)

case CONSTANT(c) :
    return EvalConstant(exp, arglist, mode)
```

Figure 6.2: Top-level algorithm for symbolic functional evaluation

implements spine unwinding where the arguments to an application are placed on an expression list until the tip of the application is reached.

The possible values for the “mode” parameter are the three tags representing the minimum desired level of the output expression: SYM_EVAL, RW_EVAL, and PD_EVAL. The possible tags are ordered with the highest level being FULL_EVAL. The two functions `EvalLevel` and `SetEvalLevel` respectively access and set the tag giving the evaluation level in the data structure representing an expression.

The first part of the algorithm determines if the top-level expression (i.e., `args` is `NULL`) has been sufficiently evaluated already, in which case no processing needs to be carried out. If the expression has already been partially unwound (i.e., `args` is not NULL), but the left branch of the application has already been evaluated to PD_EVAL and this tag is the mode of SFE, then evaluation can stop and the left branch can be recombined with its arguments.

Figure 6.3 gives the algorithm for `Recombine`. If the desired level of evaluation is RW_EVAL then each argument is evaluated to the point of distinction. If the mode is PD_EVAL, then the arguments to the function do not need to be evaluated. If the mode is SYM_EVAL then the arguments also need to be evaluated to the level SYM_EVAL. The function `Combine` evaluates the arguments to the desired level and re-creates the application. Because evaluation is carried out in place, the arguments may already have been evaluated to some level. The evaluation level of the result is determined according to the rules found in Table 6.1 which are derived from the BNF for the evaluation levels. The first row for the case where the tip of an application has the tag NOT_EVAL should never be encountered.

Turning again to Figure 6.2, if the expression does not have the desired tag or higher, the algorithm proceeds based on the form of the expression. If the expression is a variable, its evaluation level is set to SYM_EVAL. If there are arguments to the expression, an application is recreated by combining it with its arguments.

```
expression Recombine (expression exp, arglist args, flag mode)

if (mode==RW_EVAL) then
    return Combine(exp,args,PD_EVAL)
else if (mode==PD_EVAL) then
    return Combine(exp,args,NOT_EVAL)
else
    return Combine(exp,args,mode)
endif
```

Figure 6.3: Rebuilding the application of an undefined function

Table 6.1: Combining evaluation levels

Right Branch / Left Branch	NOT_EVAL	PD_EVAL	RW_EVAL	SYM_EVAL	FULL_EVAL
NOT_EVAL	ERR	ERR	ERR	ERR	ERR
PD_EVAL	PD_EVAL	PD_EVAL	PD_EVAL	PD_EVAL	PD_EVAL
RW_EVAL	PD_EVAL	RW_EVAL	RW_EVAL	RW_EVAL	RW_EVAL
SYM_EVAL	PD_EVAL	RW_EVAL	RW_EVAL	SYM_EVAL	SYM_EVAL
FULL_EVAL	PD_EVAL	RW_EVAL	RW_EVAL	SYM_EVAL	FULL_EVAL

If the expression is an abstraction, the arguments are substituted for the parameters in the body of the lambda abstraction and the resulting expression is evaluated. Substitution is discussed in the next section (Section 6.4.1).

If the expression is an application, spine unwinding is carried out. After evaluation, the original expression is replaced with the evaluated expression, using the function `ReplaceExpr`, to accomplish evaluation in place to maximise the re-use of results. A pointer to the expression is used.

The case for expressions that are constants will be presented separately in Section 6.4.2.

6.4.1 Substitution

This section describes the `Substitute` function used in the algorithm of Figure 6.2. Beta-reduction relies on substitution to replace the parameters of an abstraction or definition with arguments. Multiple substitutions are made within one substitution operation by using a local context. A context is a matching of parameters with arguments. Once each parameter and its argument have been placed in the context, substitution by means of the algorithm of Figure 6.4 can proceed (`SubstituteAux`). Afterwards, if there are too few arguments for parameters (i.e., the number of arguments unwound in an application is less than the number of parameters), the expression is returned as an abstraction over the extra parameters. If there are more arguments than parameters, the leftover arguments are returned to the calling function. Leftover arguments often occur in evaluating function applications such as `FST` with an argument that is a pair of higher-order functions. The result of the substitution is later evaluated as an application with these arguments.

The `SubstituteAux` algorithm recursively walks over the expression making the appropriate substitutions. By adding a flag to every expression to indicate if it has any variables, we can optimise this process. Parameters are variables when they are used in an expression. If no variables are present in a subexpression, there will not be any

```
expression SubstituteAux(expression exp)

if (not (containsVar(exp))) then
   return exp
endif

switch (formOf(exp))

case VAR (v) :
   if (v is in a local context with a substitution r) then
       return r
   else
       return exp
   endif

case APPL (f a):
   return NewAppl(Substitute(f),Substitute(a)))

case ABS (parem exp):
   /*
   assumes a list of free vars in all arguments to be substituted has
   already been calculated in freevars
   */
   if (parem in freevars)
      newparem = a variable name that
                   is not in freevars and is not free in exp
      match parem with newparem in substitution list
      exp = NewAbs(newparem,Substitute(exp))
      remove parem from substitution list
      return exp
   else
      return NewAbs(parem,Substitute(exp))
   endif
```

Figure 6.4: Part of the substitution algorithm

substitutions to carry out and the algorithm does not need to walk over that subexpression. This optimisation arises because uninterpreted constants are not considered variables (as they would normally be in the lambda calculus).

If a variable is encountered, it checks for a substitution within the context. If it encounters an expression that is an abstraction, it must check for name capture. Our algorithm for dealing with name capture is based on the rules for substitution presented in Table 3.1 on page 52. The variables contained in all of the arguments being substituted are calculated the first time an abstraction is encountered in the expression. They are stored in the global variable `freevars`. If the parameter of the abstraction expression is also in the list of free variables, then the name of the parameter must be changed to avoid name capture of the free variables in the substituted arguments. A new name is chosen that is in neither `freevars`, nor the remaining list of parameters, and is not a free variable in the body of the abstraction. The matching between the old name and the new name is inserted into a local context just as a normal substitution.

6.4.2 Evaluation of expressions with constants at the tip

The evaluation of constant expressions is decomposed into cases depending on the category of the constant. Figure 6.5 presents an algorithm for this process. If the constant is a constructor, its evaluation level is set to FULL_EVAL and it is recombined with its arguments which are evaluated to the desired level of evaluation. If the constant is a built-in function, the particular algorithm for the built-in constant is executed. If the expression is an uninterpreted constant, its evaluation level is set to SYM_EVAL and it is recombined with its arguments. If the expression is a constant defined by a non-pattern matching definition, it is treated as an abstraction.

The most complicated case involves a constant defined by a pattern matching with at least one argument. Its first argument must be evaluated to the point of distinction and then compared with the constructors determining the possible branches of the definition.

If a match is found, the arguments to the constructor expression as well as the remaining arguments to the expression are substituted into the body of the appropriate branch of the definition. The expression resulting from this substitution is then evaluated. A match may not be found because the constant was partially specified (i.e., it was not defined for all constructors), or because its first argument has an uninterpreted constant at its tip. If a match is not found, the expression is recombined with its arguments as if it is an uninterpreted constant.

6.5 Data structures

The data structure representing an expression is a directed acyclic binary graph where, as much as possible, common subexpressions are the same graph. As an expression is first constructed, it is possible to ensure that there is no duplication of subexpressions through a canonical node representation, which is a particularly attractive feature since it makes checking for syntactic equality of expressions a constant time operation.

To carry out evaluation in place, placeholder nodes must be inserted in the graph. The expression that was evaluated becomes a placeholder pointing to the evaluated expression and the structure loses the property of being canonical. Therefore, equality checking cannot rely on this property; however, by beginning with an expression in canonical form, equality of pointers will reduce some of the time needed to check for equality of expressions.

The replace operation (`ReplaceExpr` used in Figure 6.2) has to ensure no loops are created in the data structure from evaluation of non-well-founded recursion. This process can be optimised by determining whether a definition is recursive on its input. If it is not recursive, then it can never create a loop.

As expressions are evaluated and no longer used, and new expressions are created, garbage collection becomes an issue. In general, with uninterpreted constants, evaluated expressions are often larger than their unevaluated forms. We chose to use reference

```
expression EvalConstant(expression exp, expressionlist arglist, flag mode)

if (isConstructor(c)) then
   SetEvalLevel(exp,FULL_EVAL)
   if (args) then
      exp = Recombine(exp,arglist,mode)
   endif
   return exp
else if (isBuiltInFcn(c)) then
   return appropriate_built-in_fcn(arglist, mode)
else if (isUninterpreted(c)) then
   SetEvalLevel(exp,SYM_EVAL)
   if (arglist) then
      return Recombine(exp,arglist,mode)
   else
      return exp
   endif
else if (isPatternMatchingDefinition(c)) then
   if (arglist==NULL) then
      SetEvalLevel(exp,SYM_EVAL)
      return exp
   else
      arg1 = EvalExpression(first arg on arglist,NULL,PD_EVAL)
      if (there is a constructor at the tip of arg1) then
          t = constructor at the tip of arg1
          argparts = arguments to t in arg1
          if (there is a branch of Defn(c) that matches t) then
              body = branch of Defn(c) that matches t
              constrpars = parameters to t in branch definition
              (leftovers_args,newexp) = Substitute(body,
                                     constrpars and rest of parameters to defn,
                                     argparts and rest of arglist)
              return EvalExpression(newexp,leftover_args,mode)
          else
              return Recombine(exp,arglist,mode)
          endif
      else
          return Recombine(exp,arglist,mode)
      endif
   endif
else /* non-pattern matching constant definition */
   (pars, body) = Defn(c)
   (leftover_args,newexp) = Substitute(body, pars, arglist)
   return EvalExpression(newexp,leftover_args,mode)
endif
```

Figure 6.5: Evaluating a constant expression

counting, where the number of expressions pointing to a subexpression are tallied, and garbage collection is carried out on the fly.

Experiments with the analysis techniques that will be described in the next two chapters pointed out the value of being able to present the user with the unevaluated form of expressions, which makes it easier to interpret the results of the analysis. This technique is part of our solution to the problem of reporting results to the user introduced in Section 1.2.9. For example, it is used in the output of completeness analysis of tables. Cases missing from a table are presented in terms of the row labels found in the specification, rather than the evaluated (and potentially expanded) row labels.

Evaluation in place, i.e., replacing the original expression with its evaluated expression, implies the original expression is no longer available. However since a placeholder is used to point to the new expression, by not discarding the subexpressions of the old expression, the less evaluated version of the expression is present. An option of SFE keeps the unevaluated versions of expressions. If the old expression is not kept, then our implementation of some of the automated techniques *collapses* the expression back to canonical form to save space and improve performance of equality tests. Statistics for examples presented in Chapter 8 will be described in terms of the collapsed size of an evaluated expression.

Keeping the unevaluated version of the expressions can increase the evaluation time because substitutions made in the evaluated expression must also be made in the original expression. Consequently keeping the unevaluated expressions is an optional feature for evaluation. We can also customise which function applications have their unevaluated version kept. For example, the unevaluated versions of the semantic definitions are of little use in the output. The user is only interested in the more abstract versions of the elements of their specification. These optimisations save time and space.

Even though evaluation is carried out in place, expressions are created and destroyed during evaluation. Consequently a cache is used based on pointer equality of expressions only, which significantly improves performance.

6.6 Optimisation of the CoreSc semantics

For efficient evaluation, one optimisation is made in the CoreSc semantic definitions found in Appendix H. Rather than having the `TransTaken` function use a list to match the transition labels to their flags, the flags are paired with the transition descriptions passed as arguments to the various functions. It does not change the meaning of the CoreSc notation.

As the transition list becomes long, this optimisation results in significant performance improvement. Before this optimisation, every evaluation of `TransTaken` had to search the list of transition labels to determine the flag for the transition. Using our optimisation, the transition descriptions are passed around to where they are relevant. By including the flag as an element of the transition description, it is only necessary to access the field of a tuple to determine the flag. This optimisation eliminates the search time making a considerable difference when the list of transition labels is long.

6.7 Built-in constants

Although it is possible to define the Boolean and arithmetic functions and constants as recursive lambda expressions such that the functions have the expected behaviour [Gor88b], evaluation of built-in functions in this manner may be regarded as more a matter of theoretical interest than a practical implementation strategy. Instead we use arithmetic and Boolean functions in the C programming language in our implementation.

Built-in functions, such as arithmetic operators, cannot be evaluated in a lazy order. Their arguments must be evaluated to the point of distinction before the evaluation of the function is carried out.

Evaluation of most of the built-in constants in S+ proceeds by first determining if enough arguments are present to carry out the operation. If not, then the arguments are evaluated to the level required and an expression giving the application of the built-in function to the arguments is returned. The level of evaluation of the application depends on the level of evaluation of the arguments.

If enough arguments are present, they are evaluated to the level required, which is at least the point of distinction. If the evaluated arguments have the tag FULL_EVAL then the function can be carried out and a fully evaluated expression is returned. Otherwise the return expression is the application of the built-in constant to its arguments and its level of evaluation depends on the level of evaluation of the arguments.

Some simplifications, such as the evaluation of `F /\ a` into `F` (false), are carried out for the Boolean operations even if only one of the arguments can be fully evaluated.

Addition and multiplication are both associative. This property can be used to simplify expressions of the form `(a+4)+3` to `a+7`. To make it easy to carry out this kind of simplification, if one argument to the operator is fully evaluated, the convention of always returning this argument on the left branch of the application is adopted.

To check the equality of two expressions, they must each first be evaluated at least to the point of distinction. If both the expressions have constructors at their tips, these constructors can be compared. If they are not equal, then false is returned. If they are equal, then evaluation proceeds through the arguments to the constructors to see if they are equal.

If one or both expressions do not have constructors at their tip then they can only be compared to see if they are syntactially equivalent. If so, then true can be returned.

6.8 Quantification over finite types

If the variable of quantification is of Boolean type, evaluation of the inner expression proceeds to the level required.

A simple enumerated type is one where the constructors do not take any arguments. If the variable of quantification is of a simple enumerated type, then the quantifier is eliminated by applying the inner expression to all possible values of the type. For example, using the definitions

```
: chocolate := Cadburys | Hersheys | Rogers ;
tastesGood : chocolate -> BOOL;
```

the expression

```
FORALL (\(x:chocolate). tastesGood (x))
```

is evaluated to:

```
tastesGood (Cadburys) /\ tastesGood (Hersheys) /\ tastesGood (Rogers)
```

6.9 Beyond evaluation

Symbolic functional evaluation expands the use of defined constants and carries out beta-reduction. In the examples illustrating the dissertation, a few additional inference rules, optionally applied as SFE proceeds, aid in the analysis techniques. These inference rules are all applied automatically.

6.9.1 Rewriting: if-lifting

An application of the built-in function `COND` is an if-then-else expression. Many analysis techniques work with expressions only in propositional logic so it can be useful to turn an if-then-else expression into one that uses only the logical connectives of conjunction, disjunction, and negation.

The condition of the if-then-else expression must have Boolean type. As recognised in Seger [SJ92], if the value returned by the expression is Boolean, then the following equality holds and can be used as a rewrite rule to eliminate the `COND` function. We call the following Rule #1.

```
COND a b c ≡ (a /\ b) \/ (~a /\ c)
```

A method for lifting the `COND` operator outside of equality operations is described in the work on integrating BDD-based simplification into PVS [Raj95]. Jones et al. [JDB95] also describe "if-lifting" of expressions as a heuristic for their validity checking algorithm. They present two rewrite rules[2]:

```
((COND a b c) EQ COND a d e) ≡ COND a (b EQ d) (c EQ e)

((COND a b c) EQ d) ≡ COND a (b EQ d) (c EQ d)
```

This procress is called "if-lifting" because the function `EQ` is moved in and the conditional operator is moved to the outside.

We generalise slightly on the approach of Jones et al. by considering any uninterpreted constant applied to an argument with `COND` at the tip of the argument. For example, if `g` is an uninterpreted function, then:

```
g (COND a b c) d ≡ COND a (g b d) (g c d)
```

If there are multiple arguments that have `COND` at their tip, there are more possible combinations, such as:

```
g (COND a b c) (COND d e f) ≡
    COND (a /\ b) (g b e)
     (COND (~a /\ b) (g c e)
  (COND (a /\ ~b) (g b f)  (g c f)))
```

[2]We use "COND" rather than "ite", and "EQ" rather than "=" and the English alphabet for variables rather than Greek letters.

The semantics for a model produces a next configuration relation which has Boolean type. Therefore, it must always be possible to remove completely the `COND` function by carrying out a sequence of these rewrite rules that eventually results in an expression that returns a Boolean value, which can be rewritten using Rule #1.

We have implemented an option for SFE in which it carries out rewriting for expressions using `COND` based on these rules. As in general rewriting, this option involves pattern matching and checking of types of parts of the expression. It also causes multiple evaluation iterations of an expression as a `COND` operator makes its way out of an expression. This option is used when specifications involve tables whose semantics are defined using the `COND` operator, and information within the table is required to prove a particular property of the sytem using automated analysis.

6.9.2 Rewriting: equality of constructor expressions

Rajan [Raj95] breaks apart expressions involving equalities between lists into equalities between elements of the list. More generally, this operation can be applied to expressions with constructors at their tips. If the constructors are the same, this expression can be reduced to equalities between the corresponding arguments to the constructor because constructors return distinct values of a type. For example, if "`,`" is a constructor for the tuple type, then the following is true:

$$\texttt{((a,b) EQ (c,d))} \equiv \texttt{((a EQ c) /\textbackslash\ (b EQ d))}$$

This rewrite rule can be used to replace an expression of the form of the left-hand side with the one on the right. For the examples carried out to illustrate the framework, this type of rewriting was not needed. However, it could be easily implemented in SFE.

6.9.3 Implicit assumptions: NAME

Section 5.3.2 discusses how the built-in operator `NAME` can be used as an identifier for the assignable name of an action. Model semantics use this component of the meaning of an action to make assignable names retain their values in a step and to resolve race conditions. The advantage of this approach is that a packaged embedding can be used where the names used in the specification are just constants, rather than strings in a concrete syntax.

In evaluating the semantics for a specification written in the statechart style, uses of the `NAME` constant appear in expressions such as:

```
NAME (requestHeat KITCHEN) EQ NAME (valvePos LIVING_ROOM)
```

Well-formedness constraints limit `requestHeat KITCHEN` and `valvePos LIVING_ROOM` to be lifted constants, or constants applied to constructors resulting in a lifted expression. These constraints ensure an action modifies a unique assignable name. To the user, it is quite obvious that the name `(requestHeat KITCHEN)` is distinct from the name `valvePos LIVING_ROOM`. Within the logic this distinction cannot be made because an expression such as

```
(requestHeat KITCHEN) EQ (requestHeat LIVING_ROOM)
```

refers to the equality of their values not their names.

As a convenience, to provide this meta-level of reasoning that the specifier could deduce, we implicitly assume that the evaluation of a specification, where the function `NAME` is encountered, is predicated on assumptions for all relevant comparisons using `NAME`, such as:

```
~(NAME (requestHeat KITCHEN) EQ NAME (valvePos LIVING_ROOM))
```

Likewise the following is an assumption:

```
NAME (requestHeat KITCHEN) EQ NAME (requestHeat KITCHEN)
```

Using these implicit assumptions, expressions involving `NAME` in evaluation can be reduced to false or true.

To implement this reasoning, SFE associates the concatenation of the identifiers of the argument with the application of `NAME` to an argument. For example, `NAME x` would return `'x'`. As constructors for the type `STRING`, these values can be compared to each other and distinguished.

This method works well for carrying out automated analysis. It provides flexibility in specification and frees the user from having to generate tedious assumptions. To reason about a specification using actions (possibly in conjunction with statecharts) within a theorem prover, these assumptions would need to be explicitly added (and could be automatically generated by SFE) to produce the same behaviour as is found in the evaluation that we have implemented.

6.9.4 Specialisation (universal instantiation)

Specialisation (or universal instantiation) is a derived inference rule in higher-order logic [GM93]. Given a term t' and a theorem $forall x.t$, it can be inferred that $t[t'/x]$, where t' replaces free occurrences of x in t. This inference rule can be used to make information found in an environmental constraint that involves universal quantification accessible to automated analysis techniques. For example, an environmental constraint found in the separation minima is:

```
forall (A:flight) . NOT (IsLevel A AND InCruiseClimb A)
```

This constraint limits combinations of values for the two uninterpreted constants `IsLevel` and `InCruiseClimb`. The type `flight` is an uninterpreted type. In analysis of the separation minima, there are only two flights that are relevant. Specialisation derives that for constants `f1` and `f2` representing flights:

```
NOT (IsLevel f1 AND InCruiseClimb f1) AND
        NOT (IsLevel f2 AND InCruiseClimb f2)
```

SFE has a "specialisation" option that carries out universal instantiation for any constants within the current context of the correct type when universal quantification (over non-Boolean and non-simple enumerated types) is encountered in evaluation.

6.10 Summary

This chapter has described how the meaning of a specification is determined in a rigorous manner in our framework. Symbolic functional evaluation expands the semantic functions and specification definitions. It also reduces redexes to expose more details of the specification for analysis. SFE is less general than rewriting but avoids the problem of general unification. It accomplishes the same task as rewriting and removes the need for theorem proving infrastructure. SFE uses lazy evaluation (arguments are not evaluated until they are used) and evaluation in place.

SFE plays a key role in linking notational styles and automated analysis techniques without the infrastructure of a theorem prover. Symbolic functional evaluation of a specification is usually the first step in performing any type of automated analysis. Because multiple forms of analysis rely on this same first step, we can ensure that all forms of analysis have the same meaning of the specification. Evaluation in place also means that within one run, the SFE step does not have to be repeated for multiple queries.

We have defined three modes of SFE based on the desired level of evaluation of an expression. The user can choose the mode based on the type of analysis. Usually the mode that reaches the minimum level of evaluation needed for the type of analysis is chosen for efficiency.

Other options for SFE include saving the unexpanded version of the expressions, rewriting of expressions involving the `COND` operator, and specialisation of expressions with universal quantification.

Evaluation of an expression can be a useful means of debugging even if no subsequent analysis is carried out. For example, evaluating the semantic functions applied to particular specifications was a great help in examining their correctness. Evaluation is a form of symbolic simulation.

Symbolic functional evaluation is applicable to any expression in higher-order logic. It can have uses outside the framework. For example, in a theorem prover such as HOL, it can be considered a "super-duper tactic" to expand quickly all definitions and reduce all redexes in a proof goal.

Chapter 7

Architecture and Link to Automated Analysis Procedures

This chapter describes our solution to the last three sub-problems introduced in Chapter 1, namely, abstraction and automated analysis procedures (Section 1.2.8), reporting analysis results (Section 1.2.9), and exploiting structure (Section 1.2.10). We describe the architecture of an implementation of our framework. We demonstrate how the framework links easily with some well-known configuration space automated analysis techniques and illustrate the use of these techniques on the heating system example.

We believe our approach constitutes a contribution to the second generation of formal methods-based analysis. A key characteristic of the second generation is the decoupling of specification notation from analysis technique. Symbolic functional evaluation allows analysis to be parameterised by the semantic functions. In this chapter we consider how automated analysis techniques can be linked into our framework. We demonstrate how model checking, simulation, completeness and consistency checking are all carried out on the heating system specification in our framework.

This chapter demonstrates the extensibility of the framework with new analysis techniques. The techniques of symmetry checking and BDD-based simulation of specifications with uninterpreted constants are introduced (Sections 7.5.3 and 7.8 respectively).

We demonstrate our claim that information contained in the structure of the specification can be used to supplement BDD-based analysis approaches by producing a better abstraction of the specification (Section 7.3.1).

Our architecture is novel in that it uses only a lightweight parse tree interface to allow multiple automated analysis procedures to be applied to the specification within the same tool. This approach contrasts with theorem provers. Theorem provers include decision procedures as automated techniques for returning "yes or no" answers; they do not usually provide facilities for returning other results. Non-specialist users are often unable to take advantage of the powerful deductive mechanisms found in a theorem prover. Automated analysis techniques include more than just decision procedures. In particular they return results that help indicate the source of a problem in a specification. For example, completeness checking of a table can determine not only if the table is complete but also the cases that are not covered if the table is incomplete. We found it is sufficient to provide a common parse tree interface to implement a variety of automated analysis procedures.

To bridge the gap between higher-order logic and automated techniques, a toolkit of re-usable elements is provided. Often analysis procedures can only be applied to a finite specification. We have grouped commonly used techniques in analysis procedures in a toolkit. Creating this toolkit allows the same functionality to be re-used rather than re-implemented and makes it easier to implement new automated analysis techniques. This toolkit includes symbolic functional evaluation (Chapter 6) and Boolean abstraction to create a finite abstraction of a specification.

Section 7.1 describes the architecture of our implementation. Most of the remainder of the chapter describes how existing analysis techniques fit into our framework. Section 7.2 describes the general method for implementing analysis procedures. Section 7.3 discusses Boolean abstraction and binary decision diagrams, which are elements of the toolkit. This section covers the problem of reporting analysis results and exploiting structure. Completeness, consistency and symmetry checking of tables is described in Section 7.5. Section 7.6 discusses another element of the toolkit: the automatic separation of the constraints in the next configuration relation into those on previous values of names and those on next values of names. This element is used in symbolic CTL model checking and simulation, which are described in Sections 7.7 and 7.8. In each of the example analysis techniques, we consider how environmental constraints can increase the accuracy of the results.

The new contributions found in this chapter are

- an architecture where multiple automated analysis procedures can be applied to the same specification through a lightweight parse tree interface
- the use of a finite partitioning of numeric ranges suggested by the structure of a row in a table
- symmetry checking of the tabular style using BDDs
- automatic separation of constraints in the next configuration relation into those on previous values of names and those on next values of names
- BDD-based simulation of specifications with uninterpreted constants.

The framework and the decision procedures described in this chapter have been implemented in a tool called Fusion[1]. In many cases, the actual commands entered by a user to invoke the analysis procedures are included to provide intuition for how the framework is used in practise.

[1]There is no relationship between our implementation and any commercial tools named "fusion".

7.1 Architecture

The first tool support for the S language was a parser and type checker called Fuss, written in C, created by the author, Jeff Joyce and Michael Donat. Fusion uses a slightly modified version of this parser and type checker. An abstract data type interface is provided to the node structure representing the parse tree[2], and to the data type containing the declared and defined types and the declared and defined constants and constructors. The parse tree is produced by the parser and type checker.

Often tools supporting formal notations, such as theorem provers, provide a "term language", which is the notation and a "meta-language", which is used to manipulate the term language. For example, the user interface to the HOL theorem prover is the programming language ML [Pau91]. Expressions in ML are used to pass definitions to the core of HOL. Having to understand and use two languages to accomplish the specification task may create a steep learning curve for prospective users of HOL.

Alternatively, the Voss Verification System [Seg93] has a general-purpose functional programming language as its interface, which is used both to write the specification and to program automated analysis procedures. This approach requires the specification language to include programming facilities.

Our choice was to make the S+ notation the input language to Fusion and implement analysis techniques in C. To provide a user interface to the analysis techniques, we needed a way for the user to instruct Fusion to carry out operations using various parts of the specification. `%` commands were created to carry out this task. A `%` command is a meta-level command that causes the tool to look up the C procedure to be performed in a registry and then carry out that procedure. All arguments to `%` commands must be constants. For example, a later section will describe the `%simulate` command, which takes a constant giving a next configuration relation and a constant defined by a list of

[2]Future implementations may hide the node structure and provide an abstract data type interface to the parse tree only.

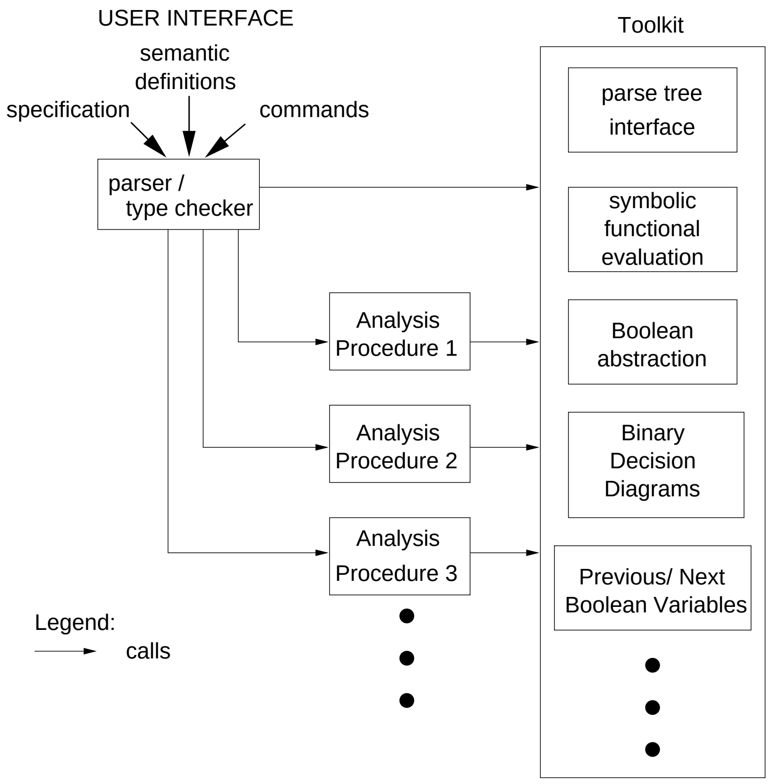

Figure 7.1: Architecture of Fusion

conditions. This method makes Fusion easily extensible and allows a user to write specifications without any knowledge of the potential analysis procedures (except type checking) until they are needed.

Fusion's architecture is illustrated in Figure 7.1. It takes as input a specification, the definitions of the semantic functions described in Chapter 5, and commands. Analysis procedures are called when the user enters a command. Results of the analysis procedures are produced to standard output.

Many analysis techniques use common procedures that can be grouped (and therefore re-used) in a toolkit. The first element in our toolkit is the interface to the parse tree.

The second element is symbolic functional evaluation as described in Chapter 6. The first step in all the automated analysis procedures described in this chapter is to use symbolic functional evaluation to expose the details of the specification to the desired evaluation level.

The third element is Boolean abstraction, which is a technique for creating a specification with a finite configuration space in Boolean logic from one with a possibly infinite state space. The efficiency of automated analysis procedures is often highly dependent on a way of manipulating Boolean expressions. Coudert et al. [CBM89] showed the value of representing Boolean expressions symbolically using reduced ordered binary decision diagrams (BDDs) [Bry86]. The fourth element of the toolkit is a BDD package. Boolean abstraction and binary decision diagrams are described in Section 7.3.

The fifth element is a technique for automatically recognising constraints on previous and next values of names of the specification (Section 7.6), which is used in conjunction with the Boolean abstraction technique to explore the configuration space of a model.

Some automated analysis procedures are just wrappers around elements of the toolkit. For example, Boolean simplification is carried out using Boolean abstraction and BDDs.

7.2 Implementing an analysis procedure

Using a parse tree representation of the specification, many different analysis procedures can be implemented that return information about the specification. The toolkit provides operations that are used by multiple procedures.

An analysis procedure is implemented as a function in the C programming language and is associated with a particular command in Fusion in the registry. Facilities are provided for passing arguments to the procedure. In a typical analysis procedure, the first step is to look up the definition of the S+ constants passed as arguments. The expression of the definition body is evaluated to the desired evaluation level using SFE to expose details about the specification. Rewriting and specialisation are applied during this process according to the settings of the options. Structures, such as that of the `Row` keyword, are also identified.

Analysis procedures manipulate the specification to return particular information to the user. This manipulation often takes the form of creating and evaluating a logical condition concerning the specification. If the condition is not true then information about the reason why it is not true is returned.

Conditions can be created in analysis procedures using a variety of methods. For example, a condition can be formed as an S+ expression or a series of S+ expressions derived from the specification that are later conjoined together.

At some point in the analysis procedures, conditions over non-finite specifications are usually abstracted. Boolean abstraction is a simple method of creating a finite abstraction of the specification that is used by the examples presented in this chapter. An abstraction created this way is often represented in a data structure called a binary decision diagram. Operations on BDDs are used to determine the truth or falsehood of the condition.

If the condition is false, expressions are isolated to indicate the source of the problem. These expressions need to be put back in terms of the specification. This process involves turning the expressions back into S+ and possibly outputting them in their structured form using the "unevaluated" forms of S+ expressions.

Alternatively, analysis procedures, such as simulation, produce results showing how a series of conditions are satisfied. The conditions in this case describe configurations as the specification moves from step to step.

In presenting the elements of the toolkit not previously described and the examples of automated analysis currently implemented in Fusion, `verbatim` font is used to represent S+ fragments of the specification and semantics, *math mode* is used to describe logical operations, and SMALL CAPS font is used for C functions.

7.3 Boolean abstraction and BDDs

Specifications within the framework can involve elements of unknown or arbitrarily large types (numbers). Many analysis procedures can only be applied to finite specifications. A finite abstraction of the specification must be created to apply these analysis procedures.

Abstractions can be created by hand or by automatic means. The choice of abstraction greatly affects the properties that can be demonstrated for the system. An abstraction can be proven to maintain certain properties of the system. A simple automatic technique for creating a finite conservative abstraction is Boolean abstraction. It produces a conservative abstraction, which has more possible behaviours than the original specification. If a universal property, such as the absence of unsafe behaviour, is shown to be true of a conservative abstraction of the specification then it is also true of the original specification.

This section discusses two elements of the toolkit: Boolean abstraction (Section 7.3.1) and binary decision diagrams (Section 7.3.2). These elements are presented together because abstracted versions of specifications can be represented using BDDs. Section 7.3.3 describes an automated analysis technique called Boolean simplification, which is simply a wrapper around the abstraction process. Section 7.3.4 discusses variable ordering, which is a key component to the use of BDDs.

7.3.1 Boolean abstraction

Rajan presents an algorithm for creating a conservative abstraction in Boolean logic of an arbitrary expression in higher-order logic [Raj95]. The expression is decomposed based on the applications of logical connectors. In carrying out the decomposition, any subexpression that is not the application of a logical connector to arguments is not decomposed any further. In the abstraction, these subexpressions are considered as independent Boolean variables. The abstracted expression is the result of applying the same logical connectors used in the original expression to the Boolean variables. For example, the expression

```
InBasicState (IDLE_NO_HEAT KITCHEN) cf /\
      (valvePos KITCHEN cf) EQ CLOSED
```

can be represented in Boolean logic as

$$a \wedge b$$

using the substitutions:

a	for	`InBasicState (IDLE_NO_HEAT KITCHEN) cf`
b	for	`(valvePos KITCHEN) cf EQ CLOSED`

Rajan's algorithm creates binary encodings of values of finite types. These encodings are similar to those carried out automatically in Ever [HDDY93].

Fusion uses this algorithm to create a Boolean abstraction of an arbitrary S+ expression. We call the S+ expressions that are matched to Boolean variables *substitutions*. A list of the matches between the substitutions and variables is maintained in the abstraction process so the same subexpression (lexically) used multiple places in the expression is considered the same Boolean variable. The following two sections discuss the encoding of finite types and the use of the row structure captured by the `Row` keyword in the tabular style of expressions to encode partitions of numeric ranges.

If Boolean abstraction is applied in the analysis technique, then symbolic functional evaluation need only be carried out to the level of PD_EVAL. Further evaluation exposes details that are lost in the abstraction.

Encoding of finite types

The Boolean abstraction process is conservative in that related expressions may be treated independently. For example, in the specification

```
: Valve_Pos:= OPEN | HALF | CLOSED ;
: valve_pos == config -> Valve_Pos;
valvePos : Room -> valve_pos;
expr1 :=
   valvePos KITCHEN cf EQ OPEN /\
   valvePos KITCHEN cf EQ CLOSED;
```

a simple Boolean encoding of `expr1` would treat the two subexpressions

`valvePos KITCHEN cf EQ OPEN`, `valvePos KITCHEN cf EQ CLOSED`

as independent Boolean variables. A satisfying assignment of the encoding of `expr1` is a case where both of these subexpressions are true. However the type definition implicitly includes an axiom that the constructors `OPEN` and `CLOSED` are unique.

Following in the work carried out in both model checking research [HDDY93] and theorem-proving research [Raj95], a less conservative abstraction can be automatically created using an encoding of finite types. Currently, we restrict the application of this process to elements of finite types where the type constructors take zero parameters.

The substitution for an S+ expression can now involve multiple Boolean variables. In the abstraction process, if an expression like `valvePos KITCHEN cf EQ OPEN` is encountered, one of the operands of the application of `EQ` is checked to see if it is of finite type. If so, this expression is encoded as an equality between two vectors of Boolean values (one for each operand). If the operand is a constructor, it is assigned a vector of concrete Boolean values representing its position in the declaration of the type. The first type constructor listed in its definition has the bit vector representing the number zero and so on. If the number of Boolean variables needed to encode an element of the type is not equal to a power of two, then the last element of the type is encoded as all possible remaining encodings (as in Damon et al. [DJJ96]).

For example, the expression `valvePos KITCHEN cf EQ OPEN` would be represented as the Boolean expression $\neg x1 \wedge \neg x2$ if the constructor `OPEN` is represented as $[F; F]$ and the expression `valvePos KITCHEN cf` is represented as $[x1; x2]$. The matching of `valvePos KITCHEN cf` with the vector $[x1; x2]$ is recorded for future substitutions. The list of associated substitutions is also maintained. In this case, this list is:

`valvePos KITCHEN cf EQ OPEN`	$\neg x1 \wedge \neg x2$
`valvePos KITCHEN cf EQ HALF`	$x1 \wedge \neg x2$
`valvePos KITCHEN cf EQ CLOSED`	$(\neg x1 \wedge x2) \vee (x1 \wedge x2)$

Partitioning of numeric values

The `Row` keyword of the tabular style of expressions provides a structure for grouping related conditions. The specifier can use it to partition an expression with a numeric value into ranges of interest. The Boolean abstraction process uses a partition specified by the structure to create a less conservative abstraction for conditions on numeric values. This technique is used in addition to the encoding of finite types. The first row of the table specifying the valve adjustment of the heating system is found in Table 7.1. The entries of this row partition the possible numeric values of the difference between the actual and desired temperatures into five ranges of interest:

```
1  dT i - aT i < C -5
2  (C -5 <= dT i - aT i) AND (dT i - aT i < C -2)
3  (C -2 <= dT i - aT i) AND (dT i - aT i <= C 2)
4  (C 2 < dT i - aT i) AND (dT i - aT i <= C 5)
5  C 5 < dT i - aT i
```

Structure is notation-specific. Therefore, to exploit the structure we must extend Fusion with some knowledge about this particular structure. This structure is encountered in the evaluation step carried out by SFE. We extend SFE with a registry of keywords and

Table 7.1: Row 1 of table specifying valve adjustment

	1	2	3	4	5	6	7
dT i - aT i	_ < C -5	C -5 <= _ AND _ < C -2	C -5 <= _ AND _ < C -2	C -2 <= _ AND _ <= C 2	C 2 < _ AND _ <= C 5	C 2 < _ AND _ <= C 5	C 5 < _

procedures to be carried out when those keywords are encountered in evaluation. SFE checks this registry every time it evaluates an application. The `Row` keyword is associated with a simple "interval checking" algorithm that takes the list of expressions in a row and determines if they represent a non-overlapping partition. In our current implementation, interval checking works for S+ expressions that contain relational numeric operations and have a constructor on at least one side of the operator. Interval checking also returns any ranges not used in the row entries. This partition is used in the Boolean abstraction step of analysis to encode in Boolean values the related expressions that had previously been considered independent. Our registry mechanism makes it possible to extend easily SFE with other mechanisms that exploit structure for analysis.

This section has described our technique for using information contained in the structure of a specification to supplement BDD-based approaches to analysis by producing a more precise abstraction of the specification.

7.3.2 Reduced ordered binary decision diagrams

A reduced ordered binary decision diagram [Bry86] (BDD) is a data structure for representing expressions in Boolean logic. A BDD is a directed acyclic binary graph with nodes representing the Boolean variables of the expression. The variables are in the same order along all paths in the graph (although not all variables need to be present along a path). Each path leads to a leaf node of T (true) or F (false). The edges of the graph represent assignments of the values T or F to the variable of the node that is the source of the edge. The same subexpressions of an expression are shared in the tree. The size of the representation is highly dependent on the order chosen for the variables. Figure 7.2 is an example of a BDD for the expression $a \wedge b$ with variable ordering a then b.

This data structure is well suited for use in automated analysis procedures for a number of reasons. First, for a particular variable order, an expression is represented in canonical form, which makes comparison of expressions possible in constant time. Sec-

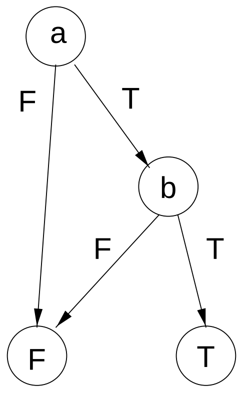

Figure 7.2: BDD for $a \wedge b$

ond, the sharing of subexpressions creates a compact representation. Third, a BDD can represent multiple logic functions in one directed acyclic graph that has a different root for each function. Fourth, BDDs can be efficiently manipulated in logical operations.

A BDD can be built to represent the Boolean abstraction of an S+ expression. The process of building the BDD occurs concurrently with creating the abstraction. The S+ operations of conjunction, disjunction, negation, and existential and universal quantification of Boolean values in an expression invoke the respective operations on the BDD data structures.

Our implementation makes use of the BDD and memory management packages developed by David Long at Carnegie Mellon University [Lon]. Our implementation can be easily changed to use a different BDD package.

7.3.3 Boolean simplification

Creating a Boolean abstraction of an S+ expression and then reversing the process, can be a useful method of simplifying expressions that include quantification over Boolean variables. The resulting expression is logically equivalent to the original. Fusion provides the command `%bddsimp <constant1> <constant2>` for this purpose. This command evaluates `constant1` to the desired level of evaluation, creates a BDD representation of the expression, and then creates an S+ expression from the BDD.

Boolean simplification was used in the heating system example to build the BDD for the next configuration relation. Section 5.13 showed that if the components of an AND-state are independent then the next configuration relation for the system can be partitioned into a conjunction of the next configuration relations for the components. This result allows the existential quantification of the transition flags to be pushed inwards. BDD simplification eliminates these quantified variables, making it possible to build the heating system's next configuration relation from the simplified version of the next configuration relation of each component, reducing the size of intermediate BDDs.

A BDD with an associated list of substitutions, created through Boolean abstraction, can be turned back into the S+ expression it represents by traversing the BDD from its root and finding the substitution matching the variable at the root node of the BDD. Two subexpressions are created recursively: one for the substitution having the value true, and the other for the substitution having the value false. Using this method the BDDs for the subexpressions do not need to be computed; they are simply the left and right branches of the BDD.

For encodings of finite types and partitions, all associated substitutions that involve the variable at the root of the BDD must be considered. The BDD being converted must be conjoined with each of the BDDs representing the related substitutions (unlike for a simple substitutions, which can just use the left and right branch of the BDD). For example, the result of applying Boolean simplification to the expression `j EQ k` where `j` and `k` are of type `Valve_pos`, is:

```
 (j EQ OPEN /\ k EQ OPEN) \/
 (j EQ HALF /\ k EQ HALF) \/
(j EQ CLOSED /\ k EQ CLOSED)
```

Both the abstraction and the reverse abstraction process benefit from the use of a cache in implementation.

This reverse abstraction technique together with SFE's ability to remember the unevaluated forms of expressions allows us to report analysis results to the specifier in terms of the original specification (Section 1.2.9).

7.3.4 Boolean variable order

For the most part, the automated procedures can be invoked by the user without being conscious of the Boolean abstraction process. However, the size of a BDD is highly dependent on the variable order chosen. In Fusion's implementation of Boolean abstraction, the Boolean variables representing the substitutions are created as new substitutions are encountered in building the BDD from the S+ expression. The order of the Boolean variables is the order of their creation. In dealing with large specifications, for example the ATN described in the next chapter, the size of the BDDs becomes a limiting factor.

To deal with this problem, we developed a way of supplying a variable ordering to Fusion for building BDDs. The variable ordering is a list of S+ expressions of type `order`. Because there are three types of substitutions (simple, finite type and partitions), `order` has three constructors, each of which contains the appropriate information for one type of substitution. Its definition is:

```
: order :=
    SE :BOOL :(NUM)list |
    FT :BOOL :(NUM)list :(BOOL)list |
    PT :(NUM)list :(BOOL)list;
```

Each type of substitution is accompanied by a list of numbers representing the position in the order of the Boolean variables used to represent the S+ expressions.

The first constructor is for a single Boolean S+ expression that is abstracted using one Boolean variable. For example, the following is a substitution for the heating system:

```
SE (requestHeat KITCHEN cf') [ 4 ]
```

The presence of this element in the S+ order indicates that the Boolean variable associated with this S+ expression is to be the fourth variable in the order.

The second type of order information represents an encoding of a finite type. For example, the following is for the encoding of the S+ expression `valvePos KITCHEN cf`, which is of type `Valve_Pos`:

```
FT (Element (valvePos KITCHEN cf)) [ 12 ; 8]
[   valvePos KITCHEN cf EQ CLOSED ;
    valvePos KITCHEN cf EQ HALF ;
    valvePos KITCHEN cf EQ OPEN   ]
```

The `Element` function is a wrapper to make it possible to type check a list of elements of type `order`. It is uninterpreted and takes an element of any type and returns a Boolean. The list of numbers indicates the position in the order of the Boolean variables used in the encodings of the expressions that follow.

The third type of substitution structure is a partition of a numeric range. For example:

```
PT  [ 3 ; 1 ; 2]
[   (C 5 < (dT KITCHEN - aT KITCHEN)) cf ;
    ((C 2 < (dT KITCHEN - aT KITCHEN)) AND
       ((dT KITCHEN - aT KITCHEN) <= C 5)) cf ;
    ((C -2 <= (dT KITCHEN - aT KITCHEN)) AND
       ((dT KITCHEN - aT KITCHEN) <= C 2)) cf ;
    ((C -5 <= (dT KITCHEN - aT KITCHEN)) AND
       ((dT KITCHEN - aT KITCHEN) < C -2)) cf ;
    ((dT KITCHEN - aT KITCHEN) < C -5) cf   ]
```

This substitution structure is similar to the constructor for finite types except the S+ expression is not isolated.

The constant defined by a list of these orders can be used as an argument to the command `%setorder <constant>`. This command creates the substitutions and declares the Boolean variables in this order.

To determine a good variable order to supply to Fusion for an expression, we chose to subcontract the problem to another tool. The Voss Verification System [Seg93] can return an optimised variable order. To link this capability with Fusion, we created a `%` command that identified and output the substitutions with an associated a Boolean variable name without building the BDD. This process also translated the S+ expression into the input language of Voss (called FL). Voss was then run with this input and the variable order produced was used with the list of substitutions matched with variable names to produce an order of S+ expressions, in the variable order suggested by Voss. This process is extremely valuable in dealing with the size of BDDs. However future implementations of Fusion could use a BDD package that carried out dynamic variable reordering sufficiently well to avoid the need for this process. The variable order generated by Voss is provided as input in the same way a user-created order would be. It does not affect the logical content of the specification.

After determining a good variable order, the next configuration relation for the heating system was built in one second (after evaluation[3]) on a dual-processor Ultra-Sparc 60 (300 MHz) with 1 GB of memory and is 2785 BDD nodes in size with 127 variables. Evaluation and BDD simplification of each component took three seconds, much of which was spent in rewriting of the `COND` operator found in the semantics for tables.

[3]Evaluation took very little time because the components had already been evaluated.

7.4 Choosing a mode for SFE

In Chapter 6, three modes of evaluation for SFE were presented. These modes are: evaluate, evaluate for rewrite simplification, and evaluate to the point of distinction. In this section, we discuss how an appropriate mode for SFE is chosen by a specifier.

The evaluate mode evaluates all arguments to uninterpreted functions. It is always possible to use this mode for any type of analysis.

The other two modes provide short cuts because full evaluation is not necessary when analysis uses Boolean abstraction. The extra information exposed by further evaluation is lost in the abstraction process. The most efficient mode is evaluate to the point of distinction.

If rewrite simplification is to be carried out (Sections 6.9.1 and 6.9.2), the evaluate for rewrite simplification or evaluate modes must be chosen. Rewriting is often necessary when a decision table is used because the semantics for decision tables use the `COND` operator. To expose information within the table for analysis, the `COND` operator must be "if-lifted".

Another reason for choosing the evaluate or evaluate for rewrite simplification modes over the most efficient mode is that by evaluating the arguments to uninterpreted functions the output may be more succinct. For example, the semantics for the CommEvent notation use the uninterpreted function `Msg`. The arguments to this function are often calculated values. The meaning of an expression including this function is more understandable once SFE has carried out the calculation.

7.5 Completeness, consistency and symmetry checking of tabular expressions

Completeness, consistency and symmetry checking are really three different analysis procedures, but because they are all applied to the same notation, they are considered together for presentation. The configuration space of a tabular specification is the possible assignments to the inputs of a table.

These procedures evaluate the relevant parts of a table using symbolic functional evaluation, then convert them to BDDs, and examine the properties of completeness, consistency and symmetry. The particular properties are presented in the relevant sections that follow. In stating these properties, we refer to a column as the conjunction of the conditions specified by the row entries in that column. A *case* is a combination of values for the row labels. One column in a table often covers multiple cases. The definitions used for completeness and consistency of tables are similar to those of Heimdahl and Leveson [Hei96], and the properties of coverage and disjointness of Heitmeyer et al. [HJL96]. Examples of the results are provided. Section 7.5.4 discusses how the results are output in a tabular form.

These procedures are invoked using the commands `%comp <constant> <env>`, `%cons <constant> <env>`, and `%sym <constant> <env>`. The `env` argument is an optional environmental constraint, which is discussed in Section 7.5.5.

7.5.1 Completeness

An expression in the notation TableExpr is *complete* if all possible combinations of values for the row labels are covered by at least one column of the table. A function table with a "Default" column is always complete. However, it is possible that more cases than intended fall into the "Default" column. When a "Default" column is used in a tabular specification, it is desirable to enumerate the cases included in that column as part of the validation effort.

A function table without a default column is complete if the disjunction of its columns is a tautology. A function table `FT`, equal to `Table rowlist resultrow`, and without a default column, is complete if

```
Exists (Columns rowlist) cf
```

is a tautology, where `Columns` is one of the semantics definitions for tables. Because a table is a lifted expression, its value must be considered in a configuration (`cf`). However if the constant representing the configuration is not defined (i.e., it is left uninterpreted), the property is checked for all possible configurations. If the property is not true, the analysis returns those cases not covered by the table, which are described by the expression:

$$\neg(\texttt{Exists (Columns rowlist) cf})$$

For a function table that has a "Default" column, the same completeness property as above is checked without including the "Default" column for the table. If the property is not true, useful output is a description of the cases that fall into the default column. If the property is true, then no cases fall into the "Default" column.

A predicate table is complete by definition since any cases not covered in the table return false. For a predicate table the analysis returns the cases that result in the predicate having the value false.

Figure 7.3 shows the results of checking the completeness of the table specifying the valve adjustment in the heating system (Table 4.4, which was previously presented on page 87 and is repeated in Table 7.2). The SFE mode of evaluate to the point of distinction was used for this example because no rewrite simplification is needed and the analysis uses Boolean abstraction. The listed cases are those not found as a column of the table (i.e., those described by ¬(`Exists (Columns rowlist) cf`)). The analysis is carried out for any instantiation of the parameters of the table. For the heating system, the result

Table 7.2: Valve position

dT i - aT i	_ < C -5	C -5 <= _ AND _ < C -2	C -5 <= _ AND _ < C -2	C -2 <= _ AND _ <= C 2	C 2 < _ AND _ <= C 5	C 2 < _ AND _ <= C 5	C 5 < _
valvePos i	.	_ = C OPEN	_ = C HALF	.	_ = C CLOSED	_ = C HALF	.
nextVp i	C CLOSED	C HALF	C CLOSED	valvePos i	C HALF	C OPEN	C OPEN

holds for any argument `i` of type `Room`. These results rely on automatically recognising the numeric partition provided by the row and the encoding of elements of the finite type `Valve_Pos`.

This output indicates that the table is not complete: no behaviour is specified for the two cases output. This error is corrected by adding a default column that returns the value `valvePos i` meaning that the valve position does not change.

This example demonstrates how information contained in the structure of the specification can be used to supplement BDD-based approaches to analysis by producing a more precise abstraction of the specification. Section 7.3.1 described a technique that uses the structure found in the rows of a table to determine a partition into ranges for a numeric value. This partition is encoded in BDDs. Without recognition of the partition found in first row of the `nextVp` table, the results would include an impossible case. The analysis would produce the result that the case where the difference between the desired and actual temperatures (`dT i - aT i`) does not fall into any of the ranges described by row one is not covered. Because the partition provided by row one is complete, this is an impossible case that would be produced by an overly conservative abstraction.

7.5.2 Consistency

A table is consistent if there is no overlap in the cases covered by columns that return different values for the function. A predicate table can never be inconsistent because all of its columns return the value true for the function.

```
>%comp nextVp

nextVp is:
(
  (Table
    [
      ((Row ((dT i) - (aT i)))
        [(\x.(x < (C -5)));
          (\x.(((C -5) <= x) AND (x < (C -2))));
          (\x.(((C -5) <= x) AND (x < (C -2))));
          (\x.(((C -2) <= x) AND (x <= (C 2))));
          (\x.(((C 2) < x) AND (x <= (C 5))));
          (\x.(((C 2) < x) AND (x <= (C 5))));
          (\x.((C 5) < x))]);
      ((Row (valvePos i))
        [Dc;(\x.(x = (C OPEN)));(\x.(x = (C HALF)));Dc;
          (\x.(x = (C CLOSED)));(\x.(x = (C HALF)));Dc])])
  [(C CLOSED);(C HALF);(C CLOSED);(valvePos i);(C HALF);
    (C OPEN);(C OPEN)])

The table is NOT complete.
Cases not covered:

Case 1
Row 1 : (((C -5) <= ((dT i) - (aT i))) AND
  (((dT i) - (aT i)) < (C -2)))
Row 2 : ((valvePos i) = (C CLOSED))

Case 2
Row 1 : (((C 2) < ((dT i) - (aT i))) AND
  (((dT i) - (aT i)) <= (C 5)))
Row 2 : ((valvePos i) = (C OPEN))
```

Figure 7.3: Completeness checking results for the table specifying valve adjustment

For a function table, `FT`, equal to `Table rowlist returnlist`, let i and j be indices to both the list of columns determined by `Columns rowlist` and to the elements in `returnlist`. Consistency analysis examines a series of properties for all pairings of i and j. Two columns are consistent if:

$$(i = j) \vee (\texttt{returnlist}_i\ \texttt{cf} = \texttt{returnlist}_j\ \texttt{cf}) \vee$$
$$\neg((\texttt{Columns rowlist})_i\ \texttt{cf} \wedge (\texttt{Columns rowlist})_j\ \texttt{cf})$$

If the above property is not satisfied for a pair of columns, the analysis procedure returns the cases that satisfy the following (for differing values of i and j):

$$(\texttt{Columns rowlist})_i\ \texttt{cf} \wedge (\texttt{Columns rowlist})_j\ \texttt{cf}$$

No inconsistencies were found in either table of the heating system specification.

7.5.3 Symmetry

In some specifications a function with two arguments may be symmetric in its arguments, i.e., the order of the two arguments is irrelevant. Functions specified by tables in the separation minima example should be symmetric. We developed a form of symmetry checking to check that the order of the two parameters to a table is irrelevant. For any table `K`, this property can be stated as

$$\texttt{K (A,B)} = \texttt{K (B,A)}$$

If the configuration `cf` is not defined then the above property is equivalent to `K (A,B) cf` $=$ `K (B,A) cf`, which is examined using the Boolean abstraction technique. Using only Boolean abstraction, it may not be possible to prove conclusively whether or not a table is symmetric. This analysis technique may rely on assumptions about the symmetry of the return values of the table if they contain uninterpreted constants. These assumptions are output.

7.5.4 Presentation of results

This section discusses how the results of completeness and consistency checking are presented in the same tabular form of their input as seen in Figure 7.3. Completeness and consistency checking report to the specifier the set of cases that are either not covered in the table or that are covered by multiple columns. Previous sections defined the properties that denote these cases. These properties are output to the user in a form where they can be viewed as a possible column in the table. This form of output is easy to interpret by the specifier since it matches the original specification in form and order of the rows in the table. This presentation is possible because SFE maintains the unevaluated versions of expressions. Tracing the source of inconsistencies through nested tables where the output is completely expanded was identified as a problem by Heimdahl [Hei96]. Likely the best approach would be one of initially providing the source of the inconsistencies in terms of the original row label (as we do) and then iteratively exploring the more detailed source of the problem.

Producing this output is more complicated than enumerating the paths found in a BDD because the condition specified by one row entry may be represented by a BDD of multiple variables. The original rows in order can be determined from the unevaluated representation of the table. This order is used for the output. The row entries in a row may not cover all possible related entries. For example, the row entries may contain only `True` and `Dc`. The row entry `False` is another possible entry for a row with a row label of Boolean type. To determine other possible related row entries that are not covered in the original table, but might be needed to express the cases to be output, the list of associated substitutions (those for finite types and partitions) is used. Our algorithm iterates over the list of rows and possible row entries producing the cases to be output as a potential column in the table. Row entries that involve constraints on multiple names, such as `False (AllOf [A;B] IsLevel)`, appearing in the output mean there is *some* way of satisfying the expression `False (AllOf [A;B] IsLevel)` that is included in the output case. Individual row entries are presented in their unevaluated form to match the row labels and entries in the original table.

In all cases in our example specifications, the possible row entries are mutually exclusive, although this is not required by the table notation. Consequently the above method may result in overlapping output cases but only when different row entries are not independent.

The cases produced using this method represent the full disjunctive normal form of the output. Just as the user will often take great advantage of the ability to put a "don't care" element in a row, the number of output cases can be summarised by using a `Dc` element to combine multiple cases.

We have developed a method of limiting the number of cases output using an approximation of the minimal sum of products (SOP) form[4]. For efficiency reasons, producing an approximation is more suitable than attempting to achieve the exact minimal SOP form. For review, it is also useful that the cases reported to the specifier are disjoint.

Briefly, our algorithm extracts the cases that can be grouped with a "don't care" value in row i by using the list of all possible row entries for row i to determine all cases that differ only in row i and that cover all possible entries for row i. For these cases a `Dc` value is used as the row entry. After extracting these cases, the remaining cases are output with particular entries for this row. The output does not produce overlapping groups of cases. The choice of "don't care" entries for rows is highly dependent on the order of the rows.

[4]The work on summarising results in Tablewise [HC95] is comparable here. However they search for output closer to the minimal sum of products.

7.5.5 Including the environment in completeness, consistency and symmetry checking

Sometimes rows in a table are not independent. For example in the separation minima specification, the predicates `IsLevel` and `InCruiseClimb` are used on different rows in a table. These predicates are time-varying properties of a flight, indicating its slope. A flight cannot have both of these attributes at the same time. This environmental constraint can be specified as:

```
forall (A:flight). NOT (IsLevel A AND InCruiseCLimb A);
```

Using the specialisation option described in Section 6.9.4, for two flights `A` and `B`, symbolic functional evaluation expands the application of this environmental constraint to the `cf` argument to:

```
~ (IsLevel A cf /\ InCruiseClimb A cf) /\
  ~ (IsLevel B cf /\ InCruiseClimb B cf)
```

The environmental constraint is conjoined with the expression describing the output to eliminate infeasible cases.

7.6 Separating previous and next configuration constraints

Returning again to the elements of the toolkit, this section describes a technique needed for analysis procedures that use the next configuration relation to explore iteratively the configuration space. After the Boolean abstraction step, names that are of infinite or uninterpreted types may exist in expressions such as `x cf' EQ (x cf PLUS 1)`. The abstraction phase considers this expression one Boolean variable. This section discusses how all expressions, including these expressions that reference multiple configurations, are split into lists of previous and next variables needed for configuration space analysis.

To ensure the generality of our framework for possible future uses, we define both the semantics of a notation and a specification in higher-order logic. Often configuration space exploration tools provide special constructs to assign a value to a name in the next configuration. For example, SMV [BCM$^+$90] uses "Next", while Ever [Hu95] and Voss [Seg93] use "becomes". We accomplish the same effect by formalising the concept of a configuration.

By using variables to represent the previous and next configurations and by making names be functions from configurations to values, we avoid the need to group explicitly the names in a record structure as has been done in PVS [Raj95]. Once a specification has been expanded by SFE, the variables representing configurations are used to determine the references to next and previous values of names.

Boolean abstraction of a next configuration relation produces a list of substitutions (constraints). The elements of the list must be separated into two lists of substitutions. One list consists of substitutions referring to the previous values of names. The second list consists of substitutions referring to the next values of names. Previous and next values for the same name are associated by the order of the two lists. We now describe the rules for constructing these two lists.

If an S+ expression associated with a substitution contains `cf` but no `cf'`, it is considered a constraint on the value of a name in the previous configuration. A search is carried out among the substitutions for the same S+ expression with `cf` replaced by `cf'`. If no match is found, then there are no constraints (that can be matched lexically) on that S+ expression in the next configuration. A placeholder is used in the list of next value constraints to ensure previous and next values of names hold corresponding positions in each list.

If a substitution has only a `cf'`, it is considered a constraint on the value of a name in next configuration. If no substitution exists that has the `cf'` replaced by `cf`, a substitution must be created with `cf'` replaced by `cf` to represent the constraint on

```
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf)
(
  (((C 2) < ((dT KITCHEN) - (aT KITCHEN))) AND
    (((dT KITCHEN) - (aT KITCHEN)) <= (C 5))) cf)
(
  (((C -2) <= ((dT KITCHEN) - (aT KITCHEN))) AND
    (((dT KITCHEN) - (aT KITCHEN)) <= (C 2))) cf)
(
  (((C -5) <= ((dT KITCHEN) - (aT KITCHEN))) AND
    (((dT KITCHEN) - (aT KITCHEN)) < (C -2))) cf)
((((dT KITCHEN) - (aT KITCHEN)) < (C -5)) cf)
((requestHeat KITCHEN) cf)
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) = (C 0)) cf)
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf))
((InBasicState (WAIT_FOR_COOL KITCHEN)) cf)
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) = coolDownTime) cf)
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf)
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf)
((InBasicState (IDLE_HEATING KITCHEN)) cf)
((tooCold KITCHEN) cf)
((tooHot KITCHEN) cf)
((InBasicState (WAIT_FOR_COOL LIVING_ROOM)) cf)
```

Figure 7.4: Partial list of previous configuration constraints in the heating system

the value in the previous configuration. The version of the substitution with `cf` must exist because iterations of the next configuration relation replace constraints on the next configuration with constraints on the previous configuration.

If a substitution includes both `cf` and `cf'`, it is treated as a constraint on the value of names in the next configuration. An S+ expression is created replacing occurrences of `cf` by `p_cf` to represent the configuration before the previous configuration, and occurrences of `cf'` by `cf`. This new expression is considered the matching constraint on the names in the previous configuration.

The result of this process is two lists of constraints, which correspond in order. One list has constraints on the value of names in the previous configuration and the other has constraints on the values of names in the next configuration (and may possibly include some placeholders). The list of constraints on previous configuration names of a next configuration relation can be examined using the command `%showprev <nc>`. Figure 7.4 shows a partial list of previous configuration constraints for the heating system.

```
:ctl :=
  At :BOOL |        /* atomic formula */
  AX :ctl |         /* for all paths, true in the next configuration */
  EX :ctl |         /* for some path, true in the next configuration */
  AG :ctl |         /* for all paths, true in all configurations */
  EG :ctl |         /* for some path, true in all configurations */
  AF :ctl |         /* for all paths, eventually true */
  EF :ctl |         /* for some path, eventually true */
  AU :ctl :ctl |    /* for all paths, arg 1 holds until arg 2 is true */
  EU :ctl :ctl |    /* for some path, arg 1 holds until arg 2 is true */
  CTLNOT :ctl |     /* arg is not true */
  CTLAND :ctl :ctl;/* arg 1 and arg 2 are true */

CTLOR A B := CTLNOT (CTLAND (CTLNOT A) (CTLNOT B));
A (_ CTLIMPL _) B := CTLOR (CTLNOT A) B;
```

Figure 7.5: CTL operators in S+

7.7 Symbolic CTL model checking

Clarke, Emerson and Sistla [CES86] introduced a technique called model checking for searching all possible configurations of a system to see if the system satisfies a particular temporal property. In CTL model checking, the properties are written in a propositional, branching time temporal logic called computational tree logic (CTL). The specification must be a finite state automaton that has a next configuration for every reachable configuration. The next configuration relation determined by the semantics of a model in Chapter 5 always has a next configuration even if there is no change in the configuration. The Boolean abstraction process produces a finite model.

CTL operators describe a property with respect to a particular starting set of configurations. CTL operators are combinations of path and configuration qualifiers. Each operator begins with A, meaning true in all paths starting from a configuration in the execution of the model, or E, meaning true in some path. The configuration qualifiers are X meaning true in the next configuration, G meaning true in every configuration along a path, F meaning eventually true and U which takes two formulae and means the first is true until the second becomes true. The CTL operators are declared in the S+ type definition found in Figure 7.5.

We draw on the presentation of Burch et al. [BCM+90] to show how the lists of previous (v) and next (v') configuration Boolean variables, and a next configuration relation (nc) can be used to produce a symbolic representation of the set of configurations that can be reached in one step from a set of configurations. Starting from a set of configurations $c1$, the set of configurations that can be reached from $c1$ is:

$$\exists v.c1 \wedge nc$$

This results in a condition in terms of the next configuration variables. Substituting the previous configuration variables for the next configuration variables, results in a symbolic representation of the set of configurations that can be reached in one step from $c1$. (This type of substitution is different from that used in Boolean abstraction to associate Boolean variables with S+ expressions.)

A representation of the set of reachable configurations can be constructed using the following operations of a BDD package:

- SUBSTITUTE(V,V',A) - This operation takes two lists of Boolean variables (V and V') and a BDD (A) and creates the BDD resulting from substituting the variables in V with the corresponding variable (determined by list position) in V' into A.
- ANDEXISTS(V,A,B) - This operation is an optimisation proposed by McMillan [McM92] to carry out the steps of existential quantification and conjunction concurrently when evaluating the expression EXISTS V . A AND B to reduce the intermediate size of the BDDs. This method is also called the relational product [Lon].

Using these BDD operations, the symbolic representation of the set of configurations reachable from $c1$ can be implemented as:

NEXTCONFIG (C1, NC, V, V') = SUBSTITUTE(V,V',ANDEXISTS(V,C1,NC))

$$
\begin{aligned}
&CTLAND\ p\ q = p \wedge q \\
&CTLNOT\ p = \neg p \\
\\
&EG\ p = \nu Y.(p \wedge EX\ Y) \\
&EU\ q\ p = \mu Y.(p \vee (q \wedge EX\ Y) \\
&EF\ p = \mu Y.(p \vee EX\ Y) \\
\\
&AX\ p = \neg EX \neg p \\
&AG\ p = \neg EF \neg p \\
&AU\ q\ p = \neg((EU\ \neg p\ (\neg q \wedge \neg p)) \vee EG \neg p) \\
&AF\ g = AU\ true\ g
\end{aligned}
$$

Figure 7.6: Definitions of CTL formula

The validity of a CTL formula is determined by iterating the next configuration relation until all relevant reachable states have been examined. Iteration is carried out using least (μ) and greatest (ν) fixed point computations, and EX, which is the NEXTCONFIG operator defined in the previous section. Figure 7.6 contains the definitions of the CTL formulae in these terms (as found in McMillan [McM92])[5].

Using a symbolic representation for a set of configurations, the fixed point computation checks at each iteration if any new configurations are encountered. When no new configurations are encountered the algorithm can halt. Because BDDs have a canonical form, this check takes constant time.

Fusion includes an implementation of CTL model checking. The command `%ctlmc <ctl_formula> <nc> <ic> <env>` takes constants with definitions that are 1) a CTL formula, 2) a next configuration relation (a model applied to the constants `cf` and `cf'`), 3) an initial condition, and 4) an optional environmental constraint described in the next section. Internally the expression representing the CTL formula is decomposed to invoke procedures based on the definitions of the component formulae. Each atomic formula is evaluated to the set SFE evaluation level and converted to a BDD. The previous and next configuration variables are determined from the next configuration relation and

[5] EF can be defined in terms of EU.

the initial condition. If all configurations satisfying the initial condition are in the set of configurations satisfying the CTL formula then this process returns that the formula is satisfied. Otherwise it returns that the formula is not satisfied.

A key feature of model checkers is their ability to produce a counterexample when a formula is not satisfied. Fusion includes counterexample generation for the CTL formulae EF and AG. Both of these formulae are defined as least fixed point calculations. A counterexample can be created by keeping track of the new previous configurations encountered in each step of the fixed point calculation.

The statechart specification of a room in the heating system (parameterised by the room `i`) is found in Figure 7.7[6]. We would like to ensure that whenever a room is in the `HEAT_REQUESTED` state, the synchronisation name `requestHeat` is also true for that room. For all rooms, this property is:

```
safe :=
 (forall (i:Room).
 (NOT
  (InState (HEAT_REQUESTED i) heatingSystemScStruct AND
     NOT (requestHeat i)))) cf;
```

Fusion's model checking showed that this formula is valid, taking one second to search the state space. The SFE mode of evaluate for rewrite simplification was chosen because rewriting is used to expand the tables of the specification.

The conservative abstraction may admit more possible reachable configurations than the original specification. An "A" CTL formula shown to be valid in the abstracted model is also true in the original specification. An "E" formula valid in the abstracted model may not be true of the specification.

[6]The textual representation of the room statechart is found in Figure 4.9 on page 103

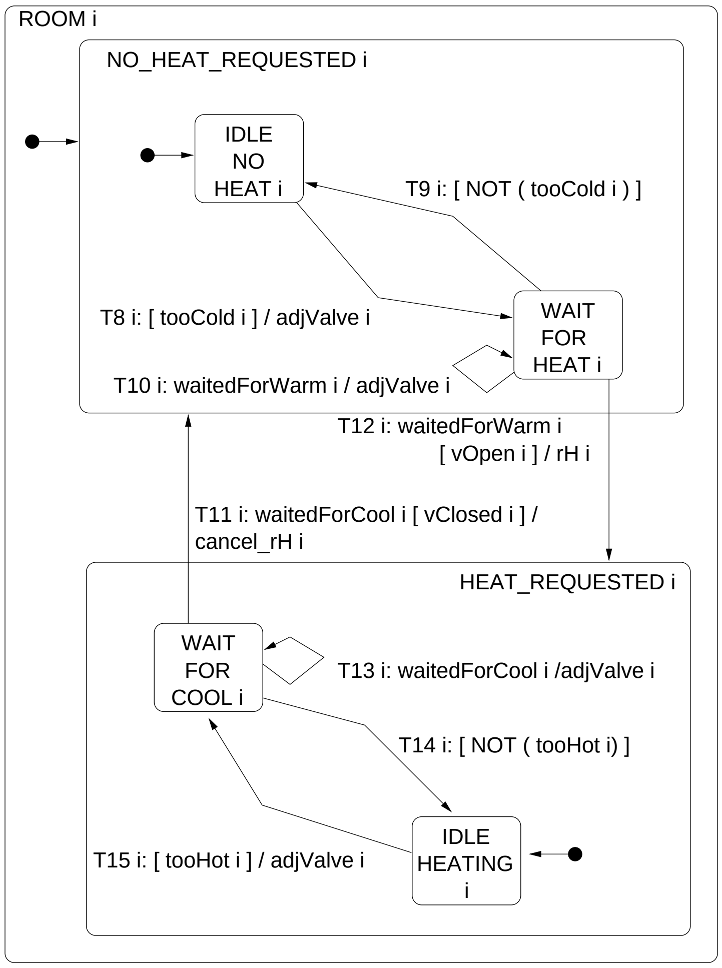
ROOM i
NO_HEAT_REQUESTED i
IDLE NO HEAT i
T9 i: [NOT (tooCold i)]
T8 i: [tooCold i] / adjValve i
WAIT FOR HEAT i
T10 i: waitedForWarm i / adjValve i
T12 i: waitedForWarm i [vOpen i] / rH i
T11 i: waitedForCool i [vClosed i] / cancel_rH i
HEAT_REQUESTED i
WAIT FOR COOL i
T13 i: waitedForCool i /adjValve i
T14 i: [NOT (tooHot i)]
T15 i: [tooHot i] / adjValve i
IDLE HEATING i

Figure 7.7: Room statechart

7.7.1 Including the environment in model checking

Adding environmental constraints to model checking can help reduce inaccurate results produced as a consequence of the Boolean abstraction of the model[7]. An environmental constraint limits the set of reachable configurations. In CTL model checking, the environmental constraint is conjoined to each atomic property in the formula, the next configuration relation (thereby limiting the possible previous configurations), and the initial condition. Furthermore, anytime a negation of the set of configurations described by a CTL formula is carried out, such as for *CTLNOT* or *AG*, the environmental constraint must be conjoined to the result to stay within the set of reachable configurations specified by both the next configuration relation and the environment.

For example, in the heating system it is important to know that transition `T8` of the room statechart cannot be interfered with by any other behaviour of the specification. Furthermore, we want to check that the valve is adjusted correctly when the room is too cold.

Inspired by the specification pattern work of Avrunin, Corbett and Dwyer [DAC98], we define the property:

```
P (_ ImmediateResponse _)  Q :=
    AG ((At P) CTLIMPL (AX (At Q)));
```

This definition means that whenever `P` is true, `Q` is always true in the next configuration[8].

[7]In the context of CTL model checking, fairness constraints are often discussed. A fairness constraint is a temporal property. The environmental constraints described here are limitations on the set of previous configurations of a step and cannot be used to describe a fairness constraint.

[8]The specification patterns work contains many similar patterns to help make temporal logic formulae more readable but not this exact formula.

A property of transition T8 can be stated as:

```
p :=
  ((InBasicState (IDLE_NO_HEAT KITCHEN) AND
   (valvePos KITCHEN = C CLOSED) AND
   ((C 2 < dT KITCHEN - aT KITCHEN) AND
          (dT KITCHEN - aT KITCHEN <= C 5))) cf)
  ImmediateResponse
  (((valvePos KITCHEN = C HALF) AND
       InBasicState (WAIT_FOR_HEAT KITCHEN)) cf);
```

To show that this property holds in the heating system specification, the analysis procedure needs to recognise the relationship between the trigger of transition `T8`, which is the condition `tooCold KITCHEN`, and the difference between the desired and actual temperatures. Our analysis tool does not currently have the capability to make this logical deduction. However, using an environmental constraint, these two conditions can be related to show that the property is valid. The suitable environmental constraint is[9]:

```
env :=
  (forall i.
    let delta := dT i - aT i in
    (tooCold i =
         ( ((C 2 < delta) AND (delta <= C 5)) OR
         (C 5 < delta))) AND
    (tooHot i =
         ( (delta < C -5) OR
         ((C -5 <= delta) AND (delta < C -2))))) cf;
```

This constraint states that the condition `tooCold` is true if and only if the difference in

[9]This constraint was previously mentioned in Section 4.16 on page 108.

the temperatures is between two and five degrees or greater than five degrees. It includes a similar constraint on the condition `tooHot`.

The property `p` also relies on the knowledge that the range partitions are mutually exclusive. To validate this property, the desired temperature was left as an uninterpreted constant and rewriting was applied to expressions involving the conditional operator to break apart the if-then-else construct of the valve adjustment table. Fusion took less than a second to search the state space to validate this property.

This property does not depend on any of the timeout events. To prove that the system eventually responds to a room being too cold for a finite duration, the timeout delays must be finite, which can be specified using the `TmB` keyword instead of `Tm`.

7.8 Simulation

A non-exhaustive form of configuration space exploration is simulation. Simulation normally allows the user to set values of names and then take a step and observe the next value of the names. New inputs to the system can be provided at each step. In a nondeterministic specification, the analysis procedure can arbitrarily choose a next value based on the possibilities. The presence of uninterpreted constants in the specification forces the simulation to be symbolic. The values are dependent on the unknown values of the uninterpreted constants.

There are two ways that simulation can be applied to a specification in Fusion. The first is similar to previous efforts at symbolic simulation in theorem provers as mentioned in Section 2.3. A deterministic specification can be written as a function that takes a configuration and produces a configuration. By creating an expression in S+ that iteratively applies this function to its own results, the result of executing a finite sequence of steps of the specification is denoted. Evaluating this expression using SFE determines the configuration resulting from the execution of the sequence of steps. SFE provides an efficient means of carrying out symbolic simulation of functional specifications.

For specifications that are models, the semantics produce a relation. These specifications may be nondeterministic. The Boolean abstraction process creates a finite abstraction of the specification. The Boolean variables of this abstraction are separated into those representing constraints on the previous configuration and those of the next configuration. By choosing an assignment of truth values for the Boolean variables of an initial configuration, and then applying the NextConfig operation, a BDD describing the possible assignments to Boolean variables of the next configuration is produced. Again a particular assignment from this set is chosen, and this assignment becomes the previous configuration for the next iteration. By choosing a particular assignment each time, this form of simulation does not encounter the configuration space explosion problem as in model checking.

The user can constrain the set of assignments possible for the initial configuration and each subsequent configuration using a list of conditions. This list of conditions is input to the simulator, instructing it to produce a sequence of configurations where $\texttt{constraint}_i$ is true in configuration i of the sequence, using the `%simulate <nc> <list>` command.

The script of a session simulating the heating system is provided in Figure 7.8. The SFE mode of evaluate for rewrite simplification was chosen because rewriting is used to expand the tables of the specification. Where "..." appears in the figure, parts of the output have been removed for presentation. Figure 7.8 shows how an occurrence of the `heatSwitchOn` event results in the controller moving from the `OFF` state to the `IDLE` state. The heating system involves many timeout events, which result in many constraints on the timer history functions. This output is automatically filtered to print only the substitutions associated with the Boolean variables assigned the value true (`T`) (all others have the value false). As the number of Boolean variables representing a configuration grows large, this filtering mechanism is very valuable. The entire output and several other simulations are found in Appendix L.

```
>%load_bdd nsr heating_nsr_bddfile
Bdd of nsr, size 2785 successfully loaded.
>ic := InitialCond (heating_system_sc_struct) cf;
ic := ((InitialCond heating_system_sc_struct) cf);
>sim1 := [
  ic;
  heatSwitchOn cf;
  T
];
sim1 := [ic;(heatSwitchOn cf);T];
>%simulate nc sim1

Configuration 0:
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T

calculating next config took: 0 sec bdd size: 48
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(heatSwitchOn cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
...
calculating next config took: 0 sec bdd size: 48
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 2:
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
...
```

Figure 7.8: Simulation of the heating system

A sequence of steps may not exist that satisfies the listed conditions. This case occurs when there are no next configurations that satisfy the condition for the results of that step. It is possible that an arbitrary choice of a particular configuration that satisfies a condition made early in the simulation results in a satisfying sequence of steps not being found when one does exist. To help with this problem, an alternative simulation process carries out "one-lookahead" (`%simulate_one_ahead`). At each step, it chooses a configuration that satisfies the applicable condition and has a next configuration that satisfies the next condition in the list. Some examples in the Appendix L demonstrate the use of this command.

In simulation, environmental constraints are conjoined with each constraint in the list.

7.9 Summary

This chapter has presented the architecture of our implementation of the framework proposed in this dissertation. We have described our solutions to the problems of linking automated analysis to the framework, reporting analysis results in terms of the original specification, and exploiting structure.

Our architecture consists of a parser and a type checker that produce a parse tree representation of the specification. Analysis procedures manipulate this parse tree to determine information about the specification. In working with one common base formalism we identified a re-usable toolkit of techniques such as symbolic functional evaluation and Boolean abstraction for multiple analysis techniques. Examples of five automated analysis procedures were described in this chapter. Analysis procedures such as simulation and model checking are available to specifications in any notational styles in the model category (and potential new styles). The toolkit and parse tree interface make it easy to add analysis procedures. For example, our implementation of a symbolic CTL model checker (without counterexample generation) is approximately 330 lines of C code.

Being able to apply multiple analysis procedures to the heating system specification, which consists of multiple notations (tables, statecharts, and higher-order logic) and uninterpreted constants, demonstrates the power of the framework. Model checking and simulation analysis relied on information found in multiple notations in the heating system specification.

Analysis results are presented to the specifier in terms of the original specification through a combination of reversing the Boolean abstraction process and using the unevaluated form of an expression maintained by the SFE step.

We demonstrate the use of structure in the way a partition found in the specifier's arrangement of items in a row can form the basis for an encoding of a numeric value in a BDD. This technique produces a more precise abstraction for analysis than strictly BDD-based approaches.

Fusion is an implementation of our framework. It is an example of a second generation analysis tool. First generation tools are those that closely couple the notation with the analysis capabilities of the tool. The identification of a toolkit of common techniques that bridge the gap between the general-purpose logic and the input to analysis techniques facilitates re-use of code and ease of extension. Our architecture is suitable for adding techniques to broaden their applicability to specifications written in multiple notations.

Chapter 8

Examples

This chapter presents two major examples that illustrate the use of our framework: the separation minima for aircraft in the North Atlantic region and the Aeronautical Telecommunications Network (ATN)[1]. The first of these examples was previously presented in Day, Joyce and Pelletier [DJP97a][2]. A discussion of the formalisation of the second was previously presented in Andrews, Day, and Joyce [ADJ97].

This chapter provides a sense of how the framework is used to tackle real specifications. In both cases, multiple notations are used for different aspects of the specification. The analysis results indicate that much can be learned about a specification using automated analysis, even when that analysis is conservative. Furthermore there are useful types of analysis between type checking and model checking that do not encounter space and time limitations for large problems.

The separation minima example shows the combination of decision tables with higher-order logic. The tables are used as entries and results in other tables interchangeably with other forms of expressions. This specification includes many uninterpreted

[1]Although the author of this dissertation has made every effort to ensure the technical accuracy of the presentations of these systems, they should not be regarded as anything more than illustrative examples.

[2]A tutorial and the complete analysis results can be found in Day, Joyce and Pelletier [DJP97b].

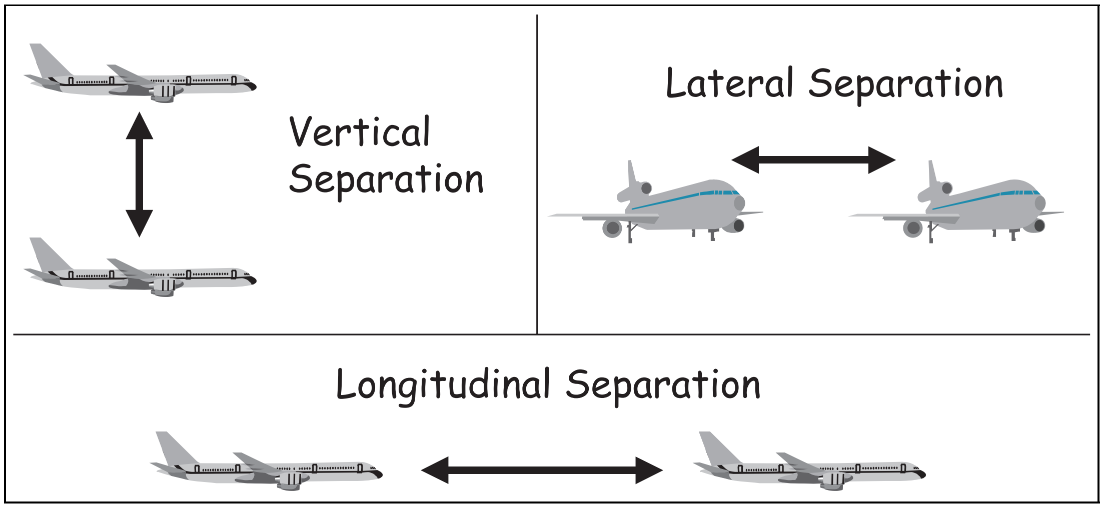

Figure 8.1: Separation minima

constants whose definitions are irrelevant to the analysis of the specification. The names of the uninterpreted constants have meaning to air traffic control experts. Completeness and consistency checking, along with the new analysis technique of symmetry checking, are applied to the separation minima. Our structure-based technique improves the analysis results compared to a strictly BDD-based approach.

The ATN example combines statecharts with higher-order logic. This combination allows the statechart specifications of the ATN components to be parameterised without any changes to the statecharts notation or its semantics. This example uses the new notations of CommAction and CommEvent for directed communication among statechart components. Model checking and the new analysis technique of simulation are demonstrated for the ATN.

The times for all results are from execution on a dual-processor Ultra-Sparc 60 (300 MHz) with 1 GB RAM running SunOS 5.6.

8.1 Separation minima for the North Atlantic Region

The first example is the specification of the separation minima for aircraft in the North Atlantic region (NAT). This specification provides guidance to air traffic controllers managing the region of oceanic airspace between Europe and North America. It is also used as the basis for the development of computer-based systems that support the management of the NAT region. For example, it would be used to plan a flight from New York to London to check whether the route is free from separation conflicts with other aircraft expected to be in the NAT region at the same time. This example specification was chosen as a good test of the use of the framework to check for the application-independent properties of completeness, consistency and symmetry in a real system. The development of symmetry checking was motivated by this example and using the toolkit of Fusion was easily implemented.

We begin by describing the specification and the effort to create it. The introduction and use of environmental assumptions in the analysis are also discussed. Our formal specification was developed in consultation with Gerry Pelletier of Hughes International Airspace Management Systems who is an expert in the development of computer-based systems that support the management of air traffic.

The conventional method for validating the informal specification is manual review. The results of our analysis, presented in Section 8.1.2, showed three places where the specification of the separation minima is inconsistent that had not previously been identified.

This example demonstrates the value of integrating notations (tabular style and predicate logic) and of using uninterpreted types and constants to maintain a level of abstraction. The restructuring of results to produce output in terms of the original specification, possible through the information retained in the process of symbolic functional evaluation, made the output of the analysis easily reviewable by our domain expert. The use of environmental constraints, as well as the structure found in the `Row` keyword, produced a more precise abstraction of the specification. The better abstraction made the results of analysis more accurate. As with all analysis procedures in the framework, the semantic definitions are used directly in the analysis.

8.1.1 Formal specification

Our formal specification is based on a document published by Transport Canada on behalf of the International Civil Aviation Organization (ICAO). This document describes the official North Atlantic separation minima published by ICAO. The document is an informal (English) specification that has been scrutinised by the NATSPG (NAT Systems Planning Group). NATSPG members are air traffic control specialists from the NAT countries. Most of them maintain and use automated systems that implement these rules.

The formal representation of the separation minima is written in a mixture of the tabular style presented in Section 4.8 and S+. It can be found in Appendix M. S+ is used to declare the uninterpreted types and constants of the specification, such as:

```
:FLIGHT;
:flight == config -> flight;
FlightLevel : flight -> num;
```

The use of uninterpreted types and constants ensured the formalisation did not add any details not present in the original informal specification. The informal specification was written for an audience of air traffic control domain experts whose knowledge includes the meaning of these uninterpreted terms. Also the definition of how the flight level of an aircraft is calculated is irrelevant to the specification of the separation minima. Carrying out analysis over uninterpreted elements is one distinguishing feature of our approach over previous work on completeness and consistency analysis. S+ is also used for parts of the specification that are not well suited to tables such as the top-level definition of separation,

```
AreSeparated(A:flight,B:flight,t:time) :=
    /* A and B are vertically separated based on flight level */
    (ABS(FlightLevel A - FlightLevel B) > VerticalSeparationRequired(A,B))
    OR

    /* A and B are laterally separated based on either position in degrees
       of latitude or position in miles */
    (if (LatitudeEquivalent(A,B))
    then
      (ABS(LateralPositionInDegrees A - LateralPositionInDegrees B) >
           "LateralSeparation RequiredInDegrees" (A,B))
    else
      (ABS(LateralPositionInMiles A - LateralPositionInMiles B) >
           "LateralSeparation RequiredInMiles" (A,B)))
    OR

    /* A and B are longitudinally separated based on time
       depending on whether the two flights are in the approximate
       same or opposite direction */
    (if (AngularDifferenceGreaterThan90Degrees
            (RouteSegment A, RouteSegment B))
    then     /* opposite direction */
        NOT (WithinOppDirNoLongSepPeriod(A,B,t))
    else     /* same direction */
        ABS(TimeAtPosition A - TimeAtPosition B) >
               LongSameDirSepRequired(A,B));
```

Figure 8.2: Top-level specification of separation

which can be found in Figure 8.2[3].

A tabular style was chosen because the specification consists of complex decision logic describing predicates and functions. The tabular style of Section 4.8 allows related conditions to be placed in the same row, which is not allowed in more conventional forms of AND/OR tables. The use of TableExpr aided in the review of the formal specification as well as providing a structure that could be exploited in analysis to produce more precise results.

The tabular approach is modular in that a table can be split into multiple tables

[3]AreSeparated is a lifted definition. It may seem strange that "time" is a lifted variable, but it is a value that changes through multiple configurations.

Table 8.1: Function table for longitudinal separation required between same direction flights

			Default
`AllOf [A;B] IsSupersonic`	`True`	`False`	
`AllOf [A;B] IsTurbojet`	`Dc`	`True`	
`LongSameDirSepRequired (A,B)`	`ssSameDir LongSep(A,B)`	`turbojetSameDir LongSep (A,B)`	`otherSameDir LongSep(A,B)`

Table 8.2: Function table for longitudinal separation required between same direction supersonic flights

			Default
`ssSubcondition(A,B)`	`True`	`True`	
`SameOrDivergingTracks(A,B)`	`True`	`True`	
`ReportedOverCommonPoint(A,B)`	`True`	`Dc`	
`AppropriateTimeSepAtCommonPoint(A,B)`	`Dc`	`True`	
`ssSameDirLongSep(A,B)`	`C 10`	`C 10`	`C 15`

when it grows too large in the number of columns or rows needed. A table represents an expression so it can be used anywhere in another table. Tables 8.1, 8.2, and 8.3 are examples of tables that reference other tables in the separation minima specification. Table 8.1 is a definition of the expression `LongSameDirSepRequired(A,B)` used in Figure 8.2.

The value of using a packaged embedding to facilitate the use of multiple notations within one specification is demonstrated in the way expressions in S+ and tables are used interchangeably. If a deep embedding had been used, when a table referenced another table, the semantic function would have to appear explicitly in the specification. For example, the first row of Table 8.2 uses `ssSubcondition`, which is specified by a table. In a deep embedding, the semantic function would have to appear there. The alternative would be to have a fixed combination of notations whose concrete syntaxes are known in advance.

The specification includes environmental constraints, which help increase the ac-

Table 8.3: Predicate table for conditions relating to supersonic flights

ssSubcondition(A,B)		
AllOf [A;B] IsLevel	True	Dc
SameMachNumber(A,B)	True	Dc
SameType(A,B)	Dc	True
AllOf [A;B] InCruiseClimb	Dc	True

curacy of the output. Three categories of environmental constraints were useful. The first relates the truth values of uninterpreted predicates, as in:

```
forall (A:flight).  NOT (IsLevel (A) AND InCruiseClimb (A))
```

The second describes the symmetry of some uninterpreted constants such as:

```
forall (A:flight) (B:flight).
    ReportedOverCommonPoint(A,B) = ReportedOverCommonPoint(B,A)
```

The third describes relationships between uninterpreted constants representing numeric comparisons such as[4]:

```
forall A.
    if "LatChange Per10DLong LessThanOrEq2" (A)
    then "LatChange Per10DLong LessThanOrEq3" (A)
```

Pelletier reviewed and edited the specification after minimal explanation of the notations being used. The specification went through several iterations as we clarified concepts in the informal document. All but one section of the informal specification was formalised; the one section left out was not of immediate interest to our domain expert for investigating our techniques. The resulting specification consists of 15 tables (both predicate and function tables), 18 definitions in S+, and 40 uninterpreted constants.

[4]As inequalities rather than uninterpreted constants, these relationships could have been determined automatically by a decision procedure.

The largest table consisted of eight rows and four columns. The formal specification is approximately 350 lines, which includes both table rows and S+ definitions.

8.1.2 Analysis and results

All of the tables in our formal specification of the separation minima have been checked for the properties of completeness, consistency and symmetry. A summary of these results is found in Table 8.4. This section highlights some of these results. The maximum time for any of these checks was one second. The largest BDD had 71 nodes and depended on 43 Boolean variables.

Analysis parameters

For the analysis of the separation minima, the evaluate to the point of distinction mode was chosen for symbolic functional evaluation because the analysis technique relies on Boolean abstraction. Rewrite simplification was not necessary and the output was sufficiently succinct without further evaluation. Rewriting was not used[5]. If it had been used, function tables within tables would have had their conditionals expanded. The use of rewriting could have made a difference if any nested tables relied on the same predicates as the table, which is not the case in any of the separation minima tables. Any remaining function applications in the evaluated expression have uninterpreted functions at their tip. The specialisation option was used in symbolic functional evaluation to expand the environmental constraints to apply to the two flights that are parameters to each table.

Completeness

Completeness analysis revealed the cases that are covered by the default column of function tables and the cases that return false in the predicate tables. All the function

[5]The "if-then" construct found in some environmental constraints is defined using implication rather than the conditional operator.

Table 8.4: Summary of analysis results of the separation minima

Table	Rows	Cols	Type	Compl	Cons	Symm
VerticalSeparationRequired	4	4	F,D	4	0	YES
LateralSeparation RequiredInDegrees	8	4	F,D	16	4	YES
LateralSeparation RequiredInMiles	8	4	F,D	16	4	YES
LatitudeEquivalent	5	6	P	17	n/a	NO
LongSameDirSepRequired	2	2	F,D	1	0	YES-A
OppDir NoLongSepPeriod	2	2	F,D	1	0	YES-A
ssOppDir NoLongSep	1	2	F	0	0	YES-A
ssSameDirLongSep	4	2	F,D	3	0	YES
ssSubcondition	4	2	P	3	n/a	YES
turbojetSameDir LongSep	2	4	F	0	0	YES-A
turbojetOppDir NoLongSepPeriod	2	4	F	0	0	YES-A
MNPSSameDir LongSep	3	5	F,D	3	0	NO
WATRSCondition	6	2	P	8	n/a	YES
genSameDir LongSep	6	3	F,D	5	0	NO
otherSameDirLongSep	3	2	F,D	2	1	YES

Type: F = function table; P = predicate table; D = has default column
Compl (completeness): number of cases not covered in table using "don't cares"
Cons (consistency): number of overlapping cases; n/a means "not applicable" if it is a predicate table
Symm (symmetry): whether the analysis could determine if the table is symmetric; YES-A means "yes, with assumptions"

```
>%comp VerticalSeparationRequired

VerticalSeparationRequired is:
     < omitted for presentation >

The following cases
yield the default value of (C 2000)
Case 1
Row 1 : (((C 280) < (FlightLevel A)) AND ((FlightLevel A) <= (C 450)))
Row 2 : ((FlightLevel B) > (C 450))
Row 3 : Dc
Row 4 : Dc

Case 2
Row 1 : ((FlightLevel A) > (C 450))
Row 2 : (((C 280) < (FlightLevel B)) AND ((FlightLevel B) <= (C 450)))
Row 3 : Dc
Row 4 : Dc

Case 3
Row 1 : (((C 280) < (FlightLevel A)) AND ((FlightLevel A) <= (C 450)))
Row 2 : (((C 280) < (FlightLevel B)) AND ((FlightLevel B) <= (C 450)))
Row 3 : Dc
Row 4 : Dc

Case 4
Row 1 : ((FlightLevel A) > (C 450))
Row 2 : ((FlightLevel B) > (C 450))
Row 3 : (False (IsSupersonic A))
Row 4 : (False (IsSupersonic B))
```

Figure 8.3: Output of completeness analysis of vertical separation table

tables that did not have default columns are complete. Figure 8.3 shows the output of completeness analysis for Table 8.5, which specifies the vertical separation required between two aircraft. The results are summarised using "don't care" values in some rows. This output illustrates the use of the `Row` construct to partition the numeric range. By exploiting the information in the `Row` structure, the missing range in the table (`(C 280 < FlightLevel A) AND (FlightLevel A <= C 450)`) is determined and is used in the output.

For some tables there are a great number of cases (e.g., 16) that are covered by the default column relative to the number of columns in a table. Our domain expert did not find any errors in the specification from these results.

The results produced are all output using the unexpanded version of the row la-

Table 8.5: Vertical separation

					Default
FlightLevel A	_ < = C 280	.	_ > C 450	_ > C 450	
FlightLevel B	.	_ < = C 280	_ > C 450	_ > C 450	
IsSupersonic A	.	.	_ = C T	.	
IsSupersonic B	.	.	.	_ = C T	
VerticalSeparation Required (A,B)	C 1000	C 1000	C 4000	C 4000	C 2000

```
>%comp ssSubcondition env

ssSubcondition is:
(PredicateTable
  [((Row ((AllOf [A;B]) IsLevel)) [True;Dc]);
    ((Row (SameMachNumber (A , B))) [True;Dc]);
    ((Row (SameType (A , B))) [Dc;True]);
    ((Row ((AllOf [A;B]) InCruiseClimb)) [Dc;True])])

The predicate is false
for the following cases:

Case 1
Row 1 : (False ((AllOf [A;B]) IsLevel))
Row 2 : Dc
Row 3 : (False (SameType (A , B)))
Row 4 : Dc

Case 2
Row 1 : (True ((AllOf [A;B]) IsLevel))
Row 2 : (False (SameMachNumber (A , B)))
Row 3 : Dc
Row 4 : (False ((AllOf [A;B]) InCruiseClimb))

Case 3
Row 1 : (False ((AllOf [A;B]) IsLevel))
Row 2 : Dc
Row 3 : (True (SameType (A , B)))
Row 4 : (False ((AllOf [A;B]) InCruiseClimb))
```

Figure 8.4: Completeness checking of supersonic subcondition

bels. For example, `FlightLevel A > C 450` appears rather than its evaluated form of `FlightLevel A cf GREATER_THAN 450`.

Figure 8.4 shows the results of completeness checking of the supersonic subcondition found in Table 8.3. This analysis used the environmental constraints relating the uninterpreted predicates `InCruiseClimb` and `IsLevel`. It also shows how the predicate `AllOf` is presented in its unexpanded form.

Consistency

The most significant result of the analysis of the separation minima was the discovery of three tables that are inconsistent. Our domain expert concluded that these results are cases where the informal specification (i.e., the original source document) is ambiguous.

Table 8.6 specifies the number of minutes that must exist between two aircraft

Table 8.6: Longitudinal separation required between same direction flights of "other" types

			Default
ReportedOverCommonPoint(A,B)	True	Dc	
SameOrDivergingTracks(A,B)	True	Dc	
AllOf [A;B] (IsOnRoute Routes3)	Dc	True	
otherSameDirLongSep(A,B)	C 15	C 20	C 30

```
>%cons otherSameDirLongSep env

otherSameDirLongSep is:
(
  (Table
    [((Row (ReportedOverCommonPoint (A , B))) [True;Dc]);
      ((Row ("SameOr Diverging Tracks" (A , B))) [True;Dc]);
      ((Row ((AllOf [A;B]) (IsOnRoute Routes3))) [Dc;True])]
    ) [(C 15);(C 20);(C 30)])

Columns 1 and 2 conflict in the following:
Case 1
Row 1 : (True (ReportedOverCommonPoint (A , B)))
Row 2 : (True ("SameOr Diverging Tracks" (A , B)))
Row 3 : (True ((AllOf [A;B]) (IsOnRoute Routes3)))
```

Figure 8.5: Output of consistency analysis of Table 8.6

(that are not both turbojet or both supersonic) flying in the same direction for them to be considered longitudinally separated. The analysis output in Figure 8.5 identifies that, for the case where two aircraft have reported over a common navigation point, are on the same or diverging tracks, and are both on a particular set of routes that have special criteria, the specification is ambiguous as to whether there should be 15 or 20 minutes of separation between them. These results are output using the unexpanded form of the row labels, such as `(True ((AllOf [A;B] (IsOnRoute Routes3)))`.

The other two tables with inconsistencies describe requirements for lateral separation. These two tables are the same except that the first table returns a value in degrees of latitude and the second table returns a value in miles. These tables have eight rows and four columns. Four inconsistent pairs of columns were found for each table. The

```
>%sym "ssOppDir NoLongSepPeriod" env

"ssOppDir NoLongSepPeriod" is:
(
  (Table [((Row (ReportedOverCommonPoint (A , B))) [True;False])]
    )
  [(P ((ept (A , B)) , ((ept (A , B)) + (C 10))));
    (P (((ept (A , B)) - (C 15)) , ((ept (A , B)) + (C 15)))
      )])

Assumption:
((P ((ept (A , B)) , ((ept (A , B)) + (C 10)))) cf)
  =
((P ((ept (B , A)) , ((ept (B , A)) + (C 10)))) cf)

Assumption:
((P (((ept (A , B)) - (C 15)) , ((ept (A , B)) + (C 15))))
  cf)
  =
((P (((ept (B , A)) - (C 15)) , ((ept (B , A)) + (C 15))))
  cf)


The table is symmetric.
```

Figure 8.6: Symmetry checking of the supersonic opposite direction no longitudinal separation period

inconsistent cases involve special provisions for particular routes that overlap with the more general criteria. The results clearly reveal cases in the informal specification that are ambiguous as to the amount of lateral separation required between aircraft.

Symmetry

Twelve of the fifteen tables in the separation minima were automatically shown to be symmetric under assumptions about the equality of return values of the tables. These results relied heavily on the environmental assumptions about the symmetry of uninterpreted constants. Figure 8.6 presents an example of the output of symmetry checking for a simple table. Fusion is unable to determine the equivalence of values in the result row so corresponding columns are compared to provide the user with some helpful feedback. These assumptions can be examined by manual review.

Symmetry checking of the table describing the conditions for “latitude equivalence” pointed out that in one row a condition was written `x < 58` and in another row it was written `58 > x`. The analysis carried out by the tool is based on syntax so it is not able to show these terms are equal. However if the table is amended to use only one form of this expression, symmetry analysis would show the table is symmetric.

In general, symmetry analysis highlighted information about the primitive terms that might not be known by an implementor of the separation minima in software.

8.2 Aeronautical Telecommunications Network

The Aeronautical Telecommunications Network (ATN) is a specification for a global telecommunications network for air traffic control systems. The ATN is being developed to allow aircraft and ground stations to exchange data. The various software components of the ATN reside in aircraft or ground station computers, and interact with human users and with each other to perform this data exchange (Figure 8.7). The communications protocols used by the software components are defined in International Civil Aviation Organization (ICAO) documents referred to as Standards and Recommended Practices (SARPs) [ATN96].

A conventional method of validating this specification would likely involve its implementation in a programming language followed by extensive simulation and testing. As an alternative to this conventional method, we created a formal specification of aspects of the ATN SARPs using a combination of the notations that have been embedded within our framework. We hoped to show that this approach would require less effort than conventional methods and be more effective in the discovery of potential problems or shortcomings in the specification. In particular, we expected that the ability to use uninterpreted constants in our specification would avoid some of the effort that would otherwise be required in a prototyping approach to "flesh out" irrelevant details. We also expected that the choice of a specialised notation that matched the original informal spec-

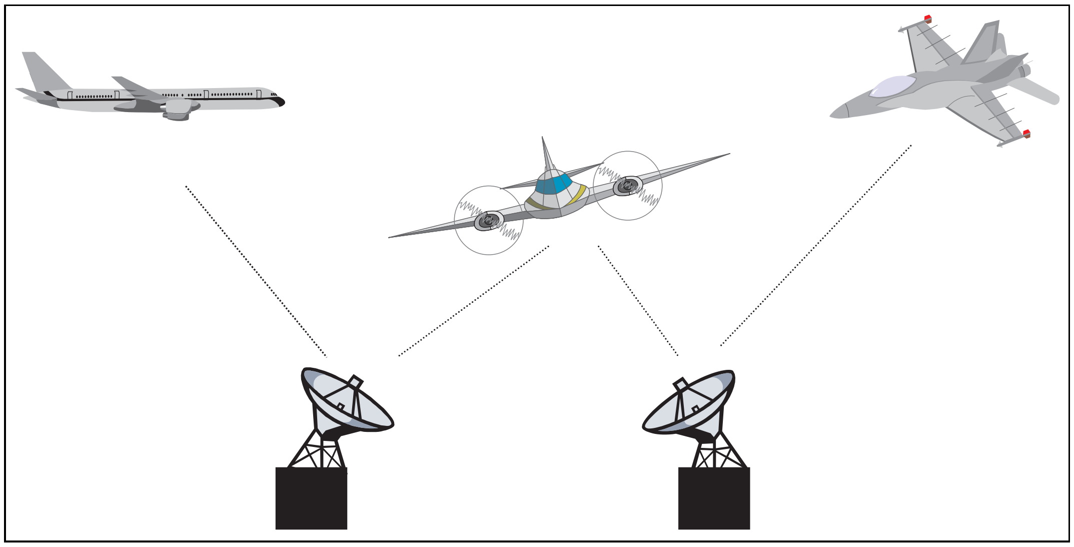

Figure 8.7: Aeronautical Telecommunications Network

ification well, namely statecharts, would be more economical than using the “flat” control structures (e.g., if-then-else) of a programming language. Finally, we expected that model checking and simulation of the formal specification itself would be a more effective way to explore the configuration space than testing a prototype implementation.

Aspects of a draft version of the ATN SARPs were formally specified by a team of researchers at the University of British Columbia (Jamie Andrews (project lead), Kendra Cooper, Michael Donat, and Ken Wong), and the University of Victoria (Dilian Gurov and Bruce Kapron), who were advised by members of Hughes International Airspace Management Systems (Ayman Farahat, Jeff Joyce, Alec MacKay, Ofelia Moldovan, Greg Saccone, and Robert Taylor). The author of this dissertation provided the framework and tool support during the specification effort. There were two goals for this effort: first, to help validate the SARPs protocols; and second, to provide a formal description of the SARPs that can potentially act as a basis for validating implementations of the ATN. The first phase of the effort involved all project members and consisted of writing and type checking a formal specification and doing some informal validation. Some problems in the draft SARPs were identified as a result of this work. The second phase was carried out by the author of this dissertation and involved configuration space exploration analysis of the specification. At the time of the formal specification effort, the type of analysis to be carried out had not been determined.

This example demonstrates the combination of the notations: statecharts, CoreEvent, CoreAction, CommAction, CommEvent and S+. The use of uninterpreted constants, such as `Msg`, which is associated with the semantics of the `Send` and `Receive` keywords (Section 5.11 on page 156), reduced the number of declarations needed in the specification. The use of S+ allowed the specification authors to partition the specification and use auxiliary declarations to make the specification more compact and readable. It also allowed the parameterisation of statechart specifications without any extension to the notation, which eliminated the need to specify the same behaviour for multiple components.

The analysis demonstrates the scalability of symbolic functional evaluation as this specification is quite large. Simple efforts, such as examining the list of previous variables for its next configuration relation, uncovered some errors in the specification. Building the BDD for the next configuration relation involved the auxiliary BDD reordering tool and partitioning the specification based on the structure of the statechart. We also adjusted the level of detail in the specification by using the `Send` and `Receive` keywords rather than `SendData` and `ReceiveData`, which involved changes to only two lines in the specification. Simulation and model checking were carried out on the formal specification of the ATN. In general, our goal was to determine the scalability of the framework by testing it on a large example.

8.2.1 Formal specification

The ATN is structured as a set of concurrent components that interact by means of messages. The top-level components of the ATN, which human users interact with, are referred to as “applications”. Applications do not communicate directly with each other; rather,

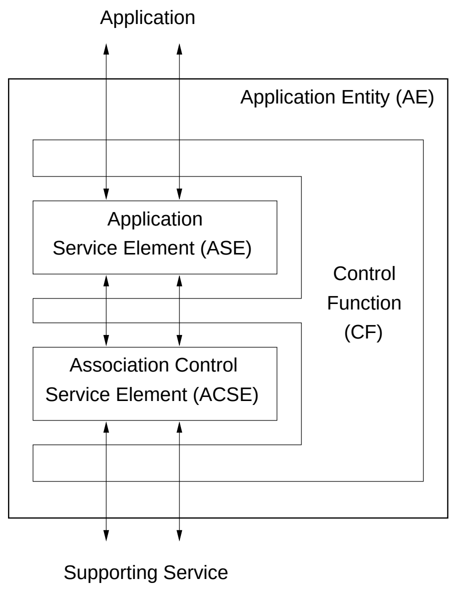

Figure 8.8: Structure of the ATN

they use a number of Application Entities (AEs) found in the OSI application layer [Tan88] to provide them with communication services. The four types of AEs formally specified are the Automated Dependent Surveillance (ADS), Context Manager (CM), Controller Pilot Data Link Communication (CPDLC), and Flight Information Service (FIS). There are two versions of each type of AE: a ground version, which resides in ground stations, and an air version, which resides in aircraft. The AEs communicate with each other by means of the supporting service found in the OSI presentation layer.

As shown in Figure 8.8, each AE consists of three entities. The Application Service Element (ASE) performs the duty of receiving messages from the application and translating them into OSI-standard messages. The Association Control Service Element (ACSE) allows its AE to form associations with other (peer) AEs. The Control Function (CF) mediates all communication amongst the ASE, the ACSE, the application, and the supporting service. Each type of AE contains a unique type of ASE, but the CF and the ACSE have the same behaviour for all types of AEs.

The SARPs consist of on the order of 1000 pages of text, containing detailed

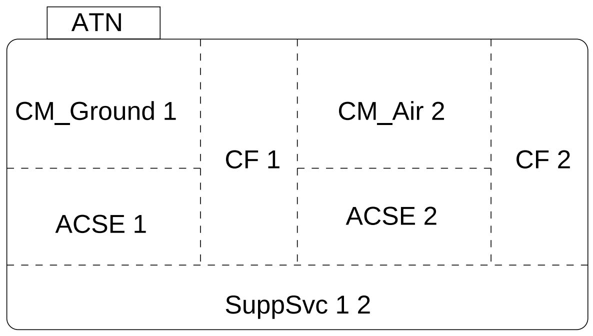

Figure 8.9: Statechart structure of example ATN

specifications of the four types of ASEs and the CF, along with requirements for the lower OSI layers and various less formal guidelines documents. The ACSE is described in a separate 40-page document, ISO 8650 [Int94].

Each entity (component) is described using an informal state transition table as well as text in paragraphs. Statecharts were a good match to the informal model. The compact representation of concurrent components of statecharts helped avoid the state explosion of a flat finite state machine. Each cell of the informal state tables mapped directly to one transition of a statechart.

Figure 8.9 shows the top-level decomposition of a two application entity ATN. Each AE consists of an application control service element (ACSE), a control function (CF), and a specific application service element. In this case the application service elements are the context manager for the ground version (CM_Ground 1) and the context manager for the air version (CM_Air 2). The formal specification includes an abstract specification of the supporting service, which simply translates messages from one AE to the other (SuppSvc 1 2).

Part of the informal state table for the ACSE component is presented in Table 8.7 from the SARPs [Int94]. The whole table consists of nine columns, four of which are shown. The rows that have no entries for the three columns shown have entries for other columns. The ACSE can be in one of several status states (e.g., Idle, Awaiting AARE).

Table 8.7: Part of the ACSE state table

STATE ⇒ EVENT ⇓	STA0 Idle- Unassoc	STA1 Awaiting AARE	STA2 Awaiting A-ASCrsp	...
A-ASCreq	p1 AARQ STA1			
A-ASCrsp+		AARE+ STA5		
A-ASCrsp-		AARE- STA0		
AARQ	p1 A-ASCind STA2 ˜p1 AARE- STA0			
AARE+		A-ASCconf+ STA5		
AARE-		A-ASCconf- STA0		
A-RLSreq				
A-RLSrsp+				
A-RLSrsp-				
RLRQ				
RLRE+				
RLRE+				
RLRE-				
A-ABRreq		ABRT STA0	ABRT STA0	
ABRT		A-ABRind STA0	A-ABRind STA0	
P-PABind		A-PABind STA0	A-PABind STA0	

The possibles states are listed across the top of the table. The SARPs also use the names STA0 and STA1 for these states but the formal specification used the more descriptive names. Each of these status states was modelled by a basic state, and these basic states were put together in an OR-state to define the overall component.

The first column of the informal state table is a list of messages that can be received by the component. Receiving a message triggers a transition. Each cell of the table represents a transition. These transitions can have guarding conditions, such as "p1", which are defined elsewhere in the documentation. For example, "p1" is "can support requested connection". The action of a transition consists of sending a message, such as AARQ. The destination state of the transition is specified in the last line of the cell. The blank cells also represent transitions. In the ACSE state table, the blank cells mean an "error" message should be sent to the Error state and the component should move into the Idle state.

The messages that are both received and sent are listed in these tables using short forms and are described in separate tables in the SARPs. The specifiers followed the trail through the informal specification to use a smaller set of more descriptive message names. For example, "AARE+" is specified using the message name `P_CONNECT_cnf_pos`.

By combining statecharts with S+, it was possible to make extensive use of parameterisation in specifying the components without complicating the semantics of statecharts. Each component was parameterised by their AE number. One specification was written for each of the CF and ACSE components and multiple instantiations of the specification were used to create the two AE ATN example.

Components were specified by different authors. Each author exploited the integration with S+ to customise their formal specification to suit the component and their particular specification style. An example of an auxiliary definition for a "transition constructor" used in the ACSE specification is found in Figure 8.10. The function `ACSE_TRANS` maps five parameters, `i` (the AE number), `sourceState`, `outMessage`, `destState` and `inMessage`, to an instance of a transition denoted by a five-tuple of the form (transition label, source state, event, action, destination state).

The use of auxiliary definitions made the specification more compact and readable. Our approach of integrating textual notations differs considerably from a graphical approach to statecharts specification where the specifier would likely have a fixed palette of constructs for the composition of a specification. The advantage of not having a fixed set of constructs stems from the generality of S+ coupled with the use of SFE in the preliminary stage of analysing a specification within our framework. No new infrastructure is required for the customisations chosen by specifiers.

The `SendData` action and `ReceiveData` event, described in Sections 4.14 and 4.15, were used extensively in this specification. Messages, such as `A_ASSOCIATE_req`, are constructors of type `msg`. The functions `ATNSend` and `ATNReceive` were defined to be `SendData` and `ReceiveData` respectively, limited to messages of the correct type. This customisation helps reduce the chance of erroneously sending messages of a different type.

Figure 8.11 shows a typical section of the ACSE specification that lists the transitions from the state `ACSE_Awaiting_AARE`. This list corresponds to the third column of Table 8.7 under the heading "STA1 Awaiting AARE". Part of the statechart that uses these transitions is found in Figure 8.12. The graphical version is provided only for illustration; our specification was developed using a textual representation of statecharts. Figure 8.11 shows the eight basic states that are substates in the ACSE component. To match clearly a column of the original informal state table, transitions leaving each state were grouped together in a definition in the specification of this component. Lines 7 to 12, 14 to 19, and 21 to 23 of Figure 8.11 each represent one transition that matches one cell in the original informal state table. The comments provided in Figure 8.11 show the correspondence with the transitions labelled in Figure 8.12. There are ten transitions of the form of the transition labelled `te` in Figure 8.12, which all send an error message and return to the idle state. The complete formal specification of the ACSE component can be found in Appendix N.

In the definition of Transitions_From_Awaiting_AARE (Figure 8.11), ACSE_TRANS is used in a let-definition of the local function TRANS_CELL. In TRANS_CELL, the function ACSE_TRANS is partially evaluated when it is applied to two values, i and Awaiting_AARE, as arguments for the first two of the five parameters of ACSE_TRANS. TRANS_CELL denotes transitions that always originate from the state Awaiting_AARE. TRANS_CELL is parameterised by the remaining three parameters of ACSE_TRANS, namely outMessage, destState and inMessage.

In the list of transitions, the element at the beginning of each dot expression is the triggering message. For example, transition t1 specified by line 11 in Figure 8.11 is a transition triggered by the event P_CONNECT_cnf_pos. Expanding the use of TRANS_CELL results in the transition:

```
( PTrans ((ACSE i).ACSE_Awaiting_AARE) P_CONNECT_cnf_pos,   /* trans label */
  (ACSE i).ACSE_Awaiting_AARE                               /* source state */
  ATNReceive s (CF i) P_CONNECT_cnf_pos (ACSEData i),       /* event */
  ATNSend s (CF i) A_ASSOCIATE_cnf_pos (ACSEData i),        /* action */
  (ACSE i).ACSE_Associated)                                 /* destination state */
```

The function PTrans produces an appropriate label for the transition. The triggering event is the message P_CONNECT_cnf_pos from the CF component. The action is sending the message A_ASSOCIATE_cnf_pos to the CF component.

The ATN SARPs include timers for purposes such as timing out dropped connections. The SARPs do not usually give details of the duration before timing out. Consequently, timers were specified as external events that could occur at any time, rather than using the Tm event.

Table 8.8 shows the effort required to create the formal specification and its size. The two CM components were created by the same specifier so their hours and lines are grouped together. The formal specification analysed is approximately 43 pages. Compared with the 1000 pages of text in the SARPs, the formal specification that was analysed did

```
ACSE_TRANS (s:(stateName,transName)sc_struct) i sourceState (outMessage:msg)
           (destState: stateName -> stateName) (inMessage:msg) :=
  ( (PTrans ((ACSE i).sourceState) inMessage),
    ((ACSE i).sourceState),
    (ATNReceive s (CF i) inMessage (ACSEData i)),
    (ATNSend s (CF i) outMessage (ACSEData i)),
    ((ACSE i).destState)
  );
```

Figure 8.10: Transition constructor in predicate logic

```
Transitions_From_ACSE_Awaiting_AARE s i :=
  /* From ACSE_Awaiting_AARE state (STA1) */
  let Error_Cell := (ACSE_error s i ACSE_Awaiting_AARE) in
  let TRANS_CELL := (ACSE_TRANS s i ACSE_Awaiting_AARE) in

  [ /* Making connection */
     A_ASSOCIATE_req     . Error_Cell;    /* te */
     A_ASSOCIATE_rsp_pos . Error_Cell;    /* te */
     A_ASSOCIATE_rsp_neg . Error_Cell;    /* te */
     P_CONNECT_ind       . Error_Cell;    /* te */
     P_CONNECT_cnf_pos   . (TRANS_CELL A_ASSOCIATE_cnf_pos ACSE_Associated);  /* t1 */
     P_CONNECT_cnf_neg   . (TRANS_CELL A_ASSOCIATE_cnf_neg ACSE_Idle);        /* t2 */
     /* Releasing connection normally */
     A_RELEASE_req       . Error_Cell;    /* te */
     A_RELEASE_rsp_pos   . Error_Cell;    /* te */
     A_RELEASE_rsp_neg   . Error_Cell;    /* te */
     P_RELEASE_ind       . Error_Cell;    /* te */
     P_RELEASE_cnf_pos   . Error_Cell;    /* te */
     P_RELEASE_cnf_neg   . Error_Cell;    /* te */
     /* Releasing connection abnormally */
     A_ABORT_req         . (TRANS_CELL P_U_ABORT_req       ACSE_Idle);  /* t3 */
     P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind         ACSE_Idle);  /* t4 */
     P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind       ACSE_Idle)   /* t5 */
  ];
```

Figure 8.11: Typical section of the ACSE specification

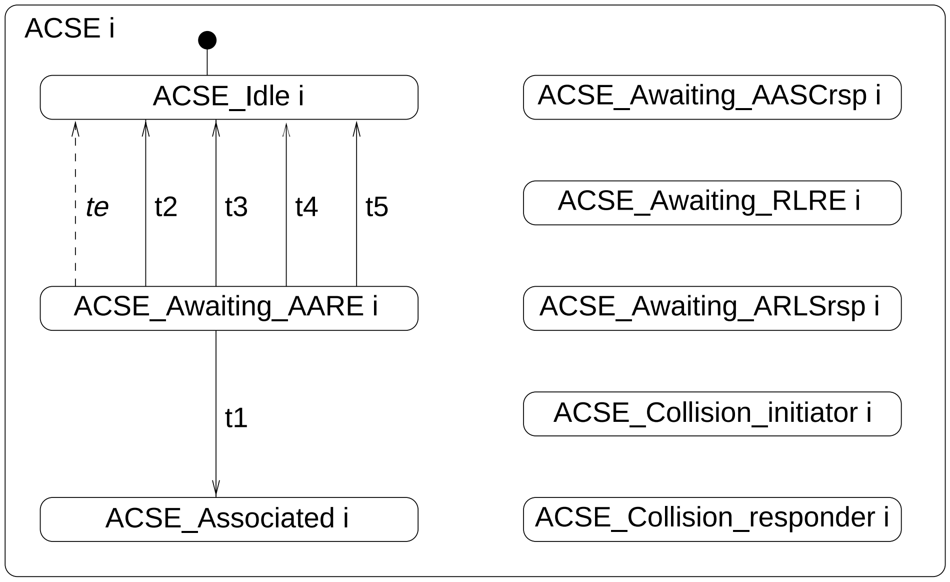

Figure 8.12: Specification of part of the ACSE component

not include the ADS, CPDLC, or FIS. The SARPs also contain explanatory text and detailed requirements on the contents of the data sent in messages, which were not formally specified. Minor changes were made after the original specification effort because our framework had evolved. An example of a change needed was to declare state names and transition labels as constructors. The group effort had left these labels as uninterpreted constants that returned elements of the types `stateName` and `transName`. For the most part, the uninterpreted constants could be defined as returning the appropriate type constructor so the specification changed little beyond the addition of the `stateName` and `transName` type definitions. Another change was to "lift" many of the data types and a few of the operators. Problems with the draft SARPs uncovered as a result of the specification and type checking analysis can be found in Andrews, Day and Joyce [ADJ97].

8.2.2 Analysis and results

Simulation and model checking of the two AE example ATN in Figure 8.9 were carried out to demonstrate the framework presented in this dissertation.

Table 8.8: Size and time of formal specification effort

Component	# of basic states	# of trans	worker hours	lines
common			32	200
SuppSvc	1	10	5	50
CF	5	96	24	680
ACSE	7	123	20	400
CM_Air	5	70	44	1790
CM_Ground	8	168		
Totals	26	467	125	3120

Analysis parameters

With the size of the ATN specification, both in terms of S+ nodes and BDDs[6], we chose not to keep the unexpanded version of expressions. This choice did not affect the readability of the output because the elements of the ATN specification appearing in the output are mainly the status of the basic states and the messages. We chose the SFE mode of evaluate for rewrite simplification. This mode reduces the arguments of uninterpreted constants, such as `Msg`, to their most compact form for readability. No rewriting was needed for this specification.

The functions `ATNSend` and `ATNReceive` were changed to use the less detailed `Send` action and `Receive` event as conservative abstractions of `SendData` and `ReceiveData`, because the properties checked did not depend on the data. This abstraction reduced the size of the configuration space and was sufficient to produce interesting results in the analysis.

Symbolic functional evaluation

The first step in both simulation and model checking is to evaluate the next configuration relation. Our implementation of symbolic functional evaluation performed well taking 39

[6]Table 8.9 provides details on these sizes.

seconds for the largest component of the ATN, which demonstrates the scalability of the technique.

Evaluation uncovered two errors in the formal specification. The first was found in the specification of the CF component, which contained the following:

```
AmInitiator : NUM -> bool;

AmResponder i := NOT(AmInitiator i);


CFT cf_sc i STA0 (SuppSvc, P_CONNECT_ind, dataSS)
    STA1
      (Both
         (ATNSend (ACSE i) P_CONN_ind (ACSEofSS dataSS))
         (Asn (AmResponder i) (C T)));
```

`CFT` is an auxiliary definition for building transitions used by the author of the CF component. SFE halted in evaluation indicating that `AmResponder` is an illegal left-hand side for an assignment statement. This error was uncovered because the `NAME` function used in the action semantics was applied to an illegal expression. The correction for this problem is to use `Asn (AmInitiator i) (C F)` as the second action of the `Both` statement.

The second error discovered was in the specification of the supporting service, which consists of one basic state `SuppSvc 1 2`. The following auxiliary definition was specified:

```
Translation (s:(stateName,transName)sc) i j
       (outMessage:msg) inMessage :=
  ( (SSTrans i j inMessage),                    /* transition label */
    (SuppSvc i j).Translate,                    /* source state */
    ATNReceive s (CF i) inMessage SuppSvcData, /* event */
    ATNSend s (CF j) outMessage SuppSvcData,   /* action */
    (SuppSvc i j).Translate                     /* destination state */
  );
```

The constant `Translation` was used as in:

```
P_CONNECT_rsp_neg . (Translation s j i P_CONNECT_cnf_neg);
```

The swapping of the two AE designator numbers (`i` and `j`) resulted in the use of the state name `Translate (SuppSvc 2 1)`, which does not exist. This error was discovered when SFE could not evaluate the next configuration relation for the supporting service.

Building the BDD

Because of the size of the configuration space, building the BDD representing the next configuration relation for the example ATN could only be accomplished by partitioning the specification component-wise. The ATN satisfied the requirements for the components of the AND-state being independent as described in Section 5.13. The next configuration relation for each component was evaluated and Boolean simplification was used to remove the existentially quantified transition flags. No information was lost in this process. A suitable variable order for each component was found and input to build the intermediate BDDs (Section 7.3.4).

The ATN next configuration relation was then created as the conjunction of the simplified next configuration relations for each component. Another variable order was required to build the final BDD. Table 8.9 presents the sizes and times for the symbolic functional evaluation phase ("SFE time"), the construction of the BDD ("abs time"), and the process of turning it back into an S+ expression ("rev abs")[7]. The last row of the table gives the times and sizes for building the ATN next configuration relation, which consists of the other components. The sizes of the collapsed S+ expressions are presented because they are in canonical form. The reverse abstraction process also produces a canonical form. The number of Boolean variables used for the ATN is equal to the sum of the Boolean variables for each component (with two CFs and two ACSEs), indicating the independence of the components, which is required to build the next configuration relation in parts. A message sent by one component to another denotes a constraint on the next configuration, while the same message received by another component is a constraint on the previous configuration.

The statistics of Table 8.9 are for the final version of the ATN used in analysis. As errors were uncovered, the specification was modified and, at times, the process of determining a new variable order had to be iterated. The next configuration relation BDD was saved in a file so it did not need to be recalculated for each analysis run.

The process of building the BDD resulted in an error being discovered in the CF specification. When the number of Boolean variables used in all the components added up to greater than the number in the ATN, smaller combinations of components were investigated to see where the overlap in variables existed. It was discovered that the data declarations of the CF component had not been parameterised. Consequently, the multiple CF components incorrectly constrained the same data. This problem was easily fixed by parameterising five data declarations in the CF and adding let-definitions to instantiate

[7]The abstraction process for the CM_Ground component took much longer than the rest because it is the largest and its variable order had not been optimised.

Table 8.9: Times and sizes for constructing the ATN next configuration relation

Component	SFE time (sec)	collapsed size (nodes)	abs time (sec)	BDD size	# Bool vars	rev abs time (sec)	rev abs size (nodes)
SuppSvc	0	415	0	97	22	0	282
CF	15	2748	1	2024	80	0	4284
ACSE	17	4406	7	823	50	0	1795
CM_Air	8	2347	2	1660	44	0	3186
CM_Ground	39	5429	75	20183	69	2	37082
ATN	0	52076	32	15553	395		

the uses of the data in the CF statecharts for the particular AE number.

Messages

After an initial version of the BDD for the next configuration relation was constructed, an attempt at simulation was made. It was discovered that there were no external messages that could be used to initiate a simulation. The CM ASE components assumed messages from their application would come via the CF. The CF component had a top-level interface of receiving messages from the CM ASE components (the ASE lower level interface) because the specification of the upper interface is left ASE-independent in the informal specification. After consulting the project lead, the most suitable resolution of this difficulty was to drop the top exchange between the CF and the two CM ASEs and allow the CM ASEs to communicate directly with the application. The role of the CF in between the ASE and the application had only been translation. This revised structure is illustrated in Figure 8.13.

The next step in analysis was to determine an appropriate initialisation constraint that ensures no internal messages exist in the initial configuration. A list of the messages used in the ATN could be obtained by sorting through the results of the `%showprev` command on the ATN next configuration relation (Section 7.6). Two messages were found

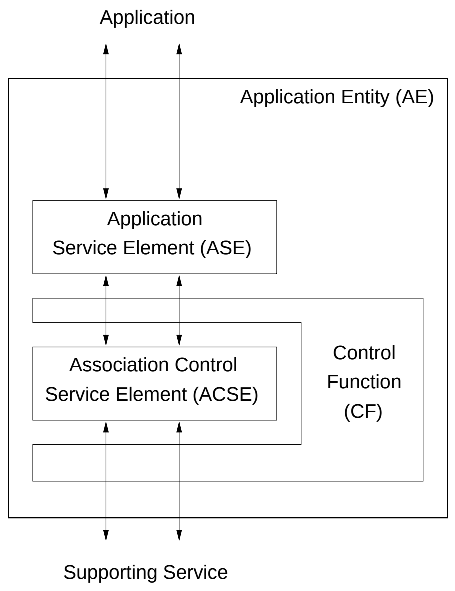
Application
Application Entity (AE)
Application Service Element (ASE)
Association Control Service Element (ACSE)
Control Function (CF)
Supporting Service

Figure 8.13: Simplified structure of the ATN

that originated at the CF and were destined for the CF. These messages were the result of typing errors in the formal specification effort and required changes to eight transitions.

Another result of this process was a list of four messages that are sent from the CF to the ASE but are never received in the CM Ground and CM Air ASEs. They may be messages not applicable to certain ASEs.

Simulation

The SARPs contain a number of message sequence charts[8] to describe the typical behaviour of the protocol. Simulation can be used to demonstrate this behaviour. An example of a message sequence chart in the SARPs is found in Figure 8.14. A D-START request from a dialogue initiator should be followed in time by a D-START indication being received by the dialogue responder. Then the dialogue responder can reply with a D-START response, which is received by the dialogue initiator as a D-START confirmation. A simulation of this sequence chart was produced using the list of constraints,

[8]Message sequence charts are part of the Specification and Description Language (SDL) developed by the CCITT (the International Telegraph and Telephone Consultative Committee).

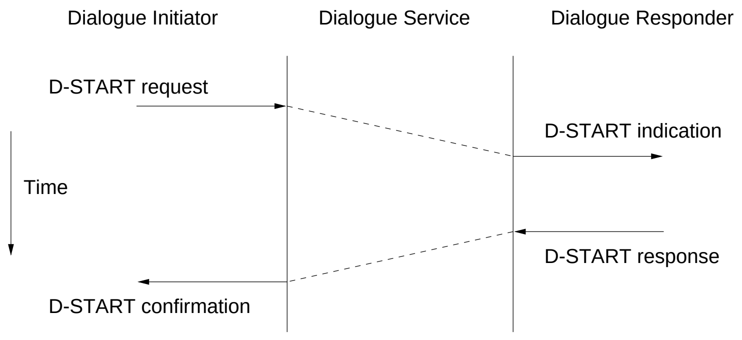

Figure 8.14: D-START message sequence chart

which is found in Figure 8.15, on each configuration of the simulation. The simulation begins from a configuration that satisfies the initial condition of the ATN, `atn_ic`, which includes being in the default basic states and no internal messages existing. The initial constraint also sends a message from the application user to the CM ground ASE. The `CM_UPDATE_req` message is a dialogue initiation message. In the next configuration the `D_START_req` is sent from the CM ground ASE to the appropriate CF. Figures 8.16 and 8.17 show highlights of the simulation output giving only the changes between the configurations. The complete simulation can be found in Appendix O.

Extra information was needed in the constraints producing the simulation to resolve nondeterminism. This extra information is not described in the message sequence chart, such as `((ACSEVersionSupported 1) = (C T)) cf` in configuration 2. The simulation also highlighted side effects of the sequences, such as `((AmInitiator 1) cf) EQ T`, which is an assignment that occurs in the CF component. It also showed that not all transitions result in state changes.

Model checking

Model checking analysis was used to check the reachability of states in the ATN by checking properties such as:

```
EF (InBasicState (CM_Ground_CONTACT_DIALOGUE 1) cf)
```

```
page2_4 :=
[
/* cf0 */  atn_ic /\ Msg (AppUser 1) (CM_Ground 1) (CM_UPDATE_req) cf;
/* cf1 */  Msg (CM_Ground 1) (CF 1) D_START_req cf;
/* cf2 */  ((ACSEVersionSupported 1) = (C T)) cf;
/* cf3 */  T;
/* cf4 */  T;
/* cf5 */  T;
/* cf6 */  ((ACSEVersionSupported 2) = (C T)) cf;
/* cf7 */  T;
/* cf8 */  (dataCM_USER 2 = C CM_UPDATE_ind) cf;
                /* Msg (CF 2) (CM_Air 2) D_START_ind cf */
/* cf9 */  T;
/* cf10 */ T;
/* cf11 */ T;
/* cf12 */ T;
/* cf13 */ T;
/* cf14 */ T;
/* cf15 */ T;
/* cf16 */ T
];
```

Figure 8.15: Simulation constraints for message sequence chart

Model checking this property returns a witness that is the shortest path to the state from the initial configuration. One case showed a path that had not been possible in the original informal state tables. This case was an error in the destination state of a transition in the CM ground component in the formal specification. This particular model checking run produced a witness of a sequence of five configurations in one second.

The ATN specification includes “cannot occur” transitions. These transitions trigger on the reception of internal messages that are not supposed to occur in a particular state. The SARPs indicate that if these messages are sent, there is an error in the implementation of the ATN. Therefore, the specification should not allow behaviour where these transitions are triggered. The specification also included transitions triggered on certain messages from the application user that are “not permitted”. We anticipated that the “cannot occur” message would only occur if a “not permitted” message had been sent.

```
>%simulate atn_nc page2_4

Configuration 0:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA0 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((((Msg (AppUser 1)) (CM_Ground 1)) CM_UPDATE_req) cf) EQ T
((InBasicState (CM_Ground_IDLE 1)) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

Configuration 1:
((((Msg (CM_Ground 1)) (CF 1)) D_START_req) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T

Configuration 2:
((InBasicState (CF_STA1 1)) cf) EQ T
((ACSEVersionSupported 1) cf) EQ T
((((Msg (CF 1)) (ACSE 1)) A_ASSOCIATE_req) cf) EQ T

Configuration 3:
((((Msg (ACSE 1)) (CF 1)) P_CONNECT_req) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T

Configuration 4:
((((Msg (CF 1)) ((SuppSvc 1) 2)) P_CONNECT_req) cf) EQ T

Configuration 5:
((((Msg ((SuppSvc 1) 2)) (CF 2)) P_CONNECT_ind) cf) EQ T

Configuration 6:
((InBasicState (CF_STA1 2)) cf) EQ T
((((Msg (CF 2)) (ACSE 2)) P_CONNECT_ind) cf) EQ T
((ACSEVersionSupported 2) cf) EQ T

Configuration 7:
((((Msg (ACSE 2)) (CF 2)) A_ASSOCIATE_ind) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
```

continued in Figure 8.17

Figure 8.16: Simulation output for message sequence chart: Part 1

continued from Figure 8.16

```
Configuration 8:
((((Msg (CF 2)) (CM_Air 2)) D_START_ind) cf) EQ T
(((dataCM_USER 2) cf) EQ CM_UPDATE_ind) EQ T

Configuration 9:
((((Msg (CM_Air 2)) (CF 2)) D_START_rsp_pos) cf) EQ T
((((Msg (CM_Air 2)) (AppUser 2)) CM_UPDATE_ind) cf) EQ T

Configuration 10:
((((Msg (CF 2)) (ACSE 2)) A_ASSOCIATE_rsp_pos) cf) EQ T

Configuration 11:
((((Msg (ACSE 2)) (CF 2)) P_CONNECT_rsp_pos) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T

Configuration 12:
((((Msg (CF 2)) ((SuppSvc 1) 2)) P_CONNECT_rsp_pos) cf) EQ T

Configuration 13:
((((Msg ((SuppSvc 1) 2)) (CF 1)) P_CONNECT_cnf_pos) cf) EQ T

Configuration 14:
((((Msg (CF 1)) (ACSE 1)) P_CONNECT_cnf_pos) cf) EQ T

Configuration 15:
((((Msg (ACSE 1)) (CF 1)) A_ASSOCIATE_cnf_pos) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 1))) cf) EQ T

Configuration 16:
((AmInitiator 1) cf) EQ T
((((Msg (CF 1)) (CM_Ground 1)) D_START_cnf_pos) cf) EQ T
```

Figure 8.17: Simulation output for message sequence chart: Part 2

The formal specification included these "cannot occur" transitions, such as:

```
CM_UTN i CM_Ground_IDLE
  D_START_cnf_neg
  (CM_send_error CannotOccur)
  CM_Ground_IDLE;
```

We used model checking to search for ways to reach a configuration where a component sends a `CannotOccur` message to the error state, as in:

```
EF (Msg (CM_Ground 1) Error CannotOccur
```

The first result of this analysis showed that timers modelled as external events were too conservative of an abstraction of the ATN behaviour. These external events could occur in the first configuration and immediately trigger a `CannotOccur` message. This model checking run took one second. This difficulty had been previously noted by Jamie Andrews by inspection.

Limiting the applicability of the timers was accomplished by adding Boolean flags associated with each timer to indicate whether the timer is active or not. Wherever the SARPs indicated that a timer is started, this flag is set to true. Likewise wherever a timer is stopped, the appropriate timer is set to false. This change affected 66 transitions in the CM air and ground components. These flags were initialised to false.

At this point the model checking times and BDD sizes grew large, which considerably slowed down (and possibly would have made it impossible) continuing to check for `CannotOccur` messages. To work around this limitation, four of the seven ATN components were left uninterpreted as in:

```
suppsvc, cf1, acse1, cm_ground : model;

atn_nc :=

    cm_air_nc /\

    cf2_nc /\

    acse2_nc /\

    suppsvc (cf,cf') /\

    cf1 (cf,cf') /\

    acse1 (cf,cf') /\

    cm_ground (cf,cf');
```

This specification is a conservative abstraction of the ATN specification and therefore contains behaviours not found in the original specification. But it produced results much more quickly than model checking the complete ATN next configuration relation. The model checking results included constraints such as `acse1(p_cf,cf) EQ T` in each configuration of the witness.

Using only half of the ATN specification, output for the model checking property was produced in 19 seconds. This output pointed out another flaw in the specification. Timers were not explicitly turned off when a timeout occurred. The informal specification should have contained explicit "stop timer" actions for these transitions. This error was fixed by setting the timer flag to false in these transitions. Greg Saccone at Raytheon[9] reported that this error had been corrected in a more recent version of the SARPs.

With this correction, the search for `CannotOccur` messages being sent in half of the ATN was continued. A run taking 1083 seconds produced a case that seemed to indicate messages coming in from the application user and from the supporting service at the same time could cause this behaviour. To ensure that this behaviour was not a consequence of the conservative abstraction created by using only half of the ATN, a

[9]During the time of this project, Hughes International Airspace Management Systems became Raytheon.

simulation of the complete ATN was created demonstrating this behaviour. A similar case had been discovered by Dilian Gurov and Bruce Kapron using a CCS (Milner's Calculus Communicating Systems) specification of the ATN [Kap97]. In their work, they used only the CM_Ground and CM_Air components and therefore could not ensure that the CF and ACSE did not stop this behaviour. Our approach has demonstrated this case is a behaviour of the ATN.

The model checking output is found in Figure 8.18. The output of the simulation is found in Appendix P and illustrated using a message sequence chart-like diagram in Figure 8.19. Each component is represented by a vertical bar. A message being sent from one component to another is illustrated by a dashed line and label. The horizontal bars represent configurations. This sequence of messages cannot actually occur in the system due to the following two paragraphs found in the SARPs that had been unknown to several of the specifiers:

> Note 6.– This CF specification assumes that the embedded ASEs (ATN-App ASE and ACSE) are modelled as atomic entities, such that when an input event is invoked by the CF, that event is processed to completion by the ASE and the CF responds to any resulting output events from the ASE, all within the same logical processing thread. This model avoids the need to specify further transient states within the CF. It does not imply any particular implementation architecture.
>
> Note 9 – For the purposes of this specification, the ATN-App AE is modelled such that a new instance of communication (effectively a new AE invocation) is implicitly created (a) for each request from the AE-User that will require a new association (i.e., that will result in a D-START request being invoked), and (b) for each indication from the underlying communications service that a new connection is requested. The AE invocation ceases to exist when the underlying communications service connection is disconnected and the CF is idle (i.e., in the NULL state).

Thus the ATN dynamically creates components. The analysis could be continued by holding all but one external input (from the application or the supporting service) constant using an environmental constraint. Another possibility would be to incorporate a notion of microsteps into our statechart semantics.

```
Configuration 0:
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

Configuration 1:
((((Msg ((SuppSvc 1) 2)) (CF 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
(suppsvc (p_cf , cf)) EQ T
(cf1 (p_cf , cf)) EQ T
(acse1 (p_cf , cf)) EQ T
(cm_ground (p_cf , cf)) EQ T

Configuration 2:
((InBasicState (CF_STA1 2)) cf) EQ T
((ACSEVersionSupported 2) cf) EQ T
((((Msg (CF 2)) (ACSE 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
(suppsvc (p_cf , cf)) EQ T
(cf1 (p_cf , cf)) EQ T
(acse1 (p_cf , cf)) EQ T
(cm_ground (p_cf , cf)) EQ T

Configuration 3:
((((Msg (ACSE 2)) (CF 2)) A_ASSOCIATE_ind) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
((((Msg (AppUser 2)) (CM_Air 2)) CM_LOGON_req) cf) EQ T
(suppsvc (p_cf , cf)) EQ T
(cf1 (p_cf , cf)) EQ T
(acse1 (p_cf , cf)) EQ T
(cm_ground (p_cf , cf)) EQ T

Configuration 4:
((((Msg (CM_Air 2)) (CF 2)) D_START_req) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((((Msg (CF 2)) (CM_Air 2)) D_START_ind) cf) EQ T
(((dataCM_USER 2) cf) EQ CM_UPDATE_ind) EQ T
((InBasicState (CM_Air_LOGON 2)) cf) EQ T
(CM_TIMEOUT_logon_active cf) EQ T
(suppsvc (p_cf , cf)) EQ T
(cf1 (p_cf , cf)) EQ T
(acse1 (p_cf , cf)) EQ T
(cm_ground (p_cf , cf)) EQ T

Configuration 5:
((InBasicState (CF_STA1 2)) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Air_LOGON 2)) cf) EQ T
(CM_TIMEOUT_logon_active cf) EQ T
((((Msg (CM_Air 2)) Error) CannotOccur) cf) EQ T
(suppsvc (p_cf , cf)) EQ T
(cf1 (p_cf , cf)) EQ T
(acse1 (p_cf , cf)) EQ T
```

Figure 8.18: Model Checking output of EF (Msg (CM_Ground 1) Error CannotOccur)

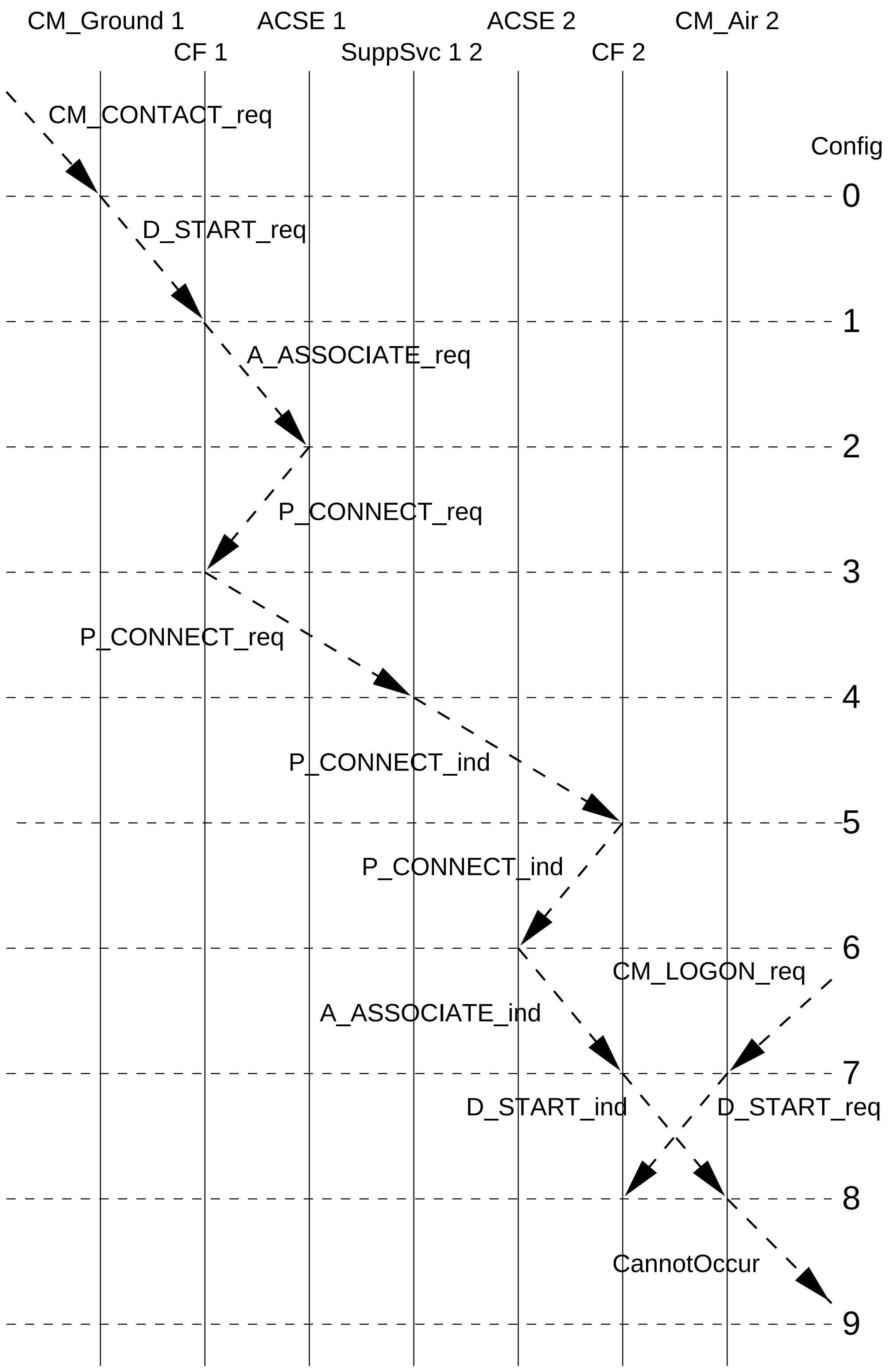

Figure 8.19: Possible behaviour of the ATN

8.3 Summary

This chapter has presented examples of using our framework to specify and analyse two real specifications, namely, the separation minima for aircraft in the North Atlantic region and an instantiation of the Aeronautical Telecommunications Network. In each case, notations were chosen to suit the different aspects of the problem. Multiple notations were easily combined in the specifications, allowing parameterisation and modularisation to make the specifications more compact and readable. Extensive use was also made of uninterpreted constants to maintain the level of abstraction of the original informal specifications.

In both cases, useful analysis results were produced that were unknown to the specifiers at the beginning of the analysis phase. Compared to conventional methods, these results were generated automatically from the specification itself. The effort of creating a prototype implementation, which might have required the addition of unnecessary details, was avoided and more exhaustive analysis techniques were applied.

In the separation minima, some of the results were actually problems in the original specification. In the ATN, many of the results uncovered errors in the formalisation process. This review is important if the formalisation is to be used as the basis for simulation, code generation, or development. Many of the errors were found through the steps leading up to model checking thus demonstrating the value of a range of analysis procedures that can be applied to the same specification.

Chapter 9

Conclusions and Future Work

We may now summarize Aristotle's motivation in inventing logic. First, there is the desire to know the truth about the nature of argument, an intellectual curiosity which needs no further account or justification. Second, there is the desire to know the conditions under which something is proved. ... Third, there is the desire to refute opponents. ..., logic is vastly more comprehensive and useful than merely as a device which may be used to show that an opponent is wrong. Yet we should not overlook the egotism and spirit of competitiveness which marked its origin. (Howard Delong, "A Profile of Mathematical Logic" [DeL71]).

This chapter marks the successful completion of our attempt to validate the conjectures put forward in the introduction. We summarise our approach and the results presented in this dissertation. Questions of the generality of the approach are then addressed. Our contributions are outlined and suggestions are provided for future research. The chapter concludes with a final word on the lasting impact.

9.1 Summary

This dissertation has addressed the problem of how to analyse specifications, written in multiple formal notations describing different aspects of a system, in a systematic, rigorous, and extensible manner.

Many requirements specification methodologies use multiple notations for different aspects of the system's description. CASE (Computer-Aided Software Engineering) tools often attempt to capture the best of many approaches by providing integrated editors to link the syntax of multiple notations. The Object Modeling Technique (OMT) [R$^+$91] and the Unified Modeling Language (UML) [Rat] use different notations for each of the dynamic, functional, and object views of the system. Examples of formal languages that contain a set of notations are the Requirements State Machine Language (RSML) [LHHR94] and the Software Cost Reduction (SCR) [Hen80] methods. The need for integration of notations is clear.

We have studied the problem of how the integration of notations can be systematised to provide flexibility in the choice of notations for specifying a system and to link multi-notation specifications with formal analysis techniques. We provide a framework that integrates the meaning of notations, not just their syntax.

In particular, we have studied model-oriented specifications that include uninterpreted constants. Uninterpreted constants are useful in system specifications for filtering non-essential details and improving readability. They can also be used to create conservative abstractions of specifications based on the user's knowledge of the system.

We have presented a framework for multi-notation requirements specification and analysis. The framework uses higher-order logic as the base formalism. Higher-order logic was chosen because of its expressibility, generality, and ability to include uninterpreted constants. The meaning of notations is defined by operational semantics. The use of packaged embeddings allows notations to be integrated based only on type consistency.

Chapter 4 discussed notations and how they can be integrated in specifications. Semantic categories of notations were identified, namely models, actions, events and lifted expressions. Each category has an associated type signature. The embedding of a notation in higher-order logic is called a notational style. Keywords of the notation denote elements of the type signature of a category. Arguments to the keywords have types that match other categories. These matches are called join points and allow type checking to regulate how notations can be used together in specification.

Chapter 5 presented the semantic functions for our example notations. In a packaged embedding, the keywords of a notation are defined in higher-order logic. These definitions are semantic functions that associate a meaning with a specification written in the notation. Once written, they can be re-used for every specification written in that notation.

We have demonstrated the integration of a variant of the statecharts notation, higher-order logic, and a tabular style of specification, along with notations for events and actions.

For analysis, a technique called symbolic functional evaluation was presented in Chapter 6. This technique directly evaluates the semantic functions to produce a semantically equivalent representation of a specification. We have defined five distinct levels of evaluation for expressions in higher-order logic. We used an algorithm based on techniques from functional programming languages extended to handle uninterpreted constants for carrying out symbolic functional evaluation. These techniques include evaluation in place and lazy evaluation.

Chapter 7 presented the architecture of our implementation of the framework. Notation is de-coupled from analysis. Through the use of semantic functions and symbolic functional evaluation, the analysis is parameterised by the set of notations used in the specification. Analysis procedures are applied to the expanded representations of specifications produced by symbolic functional evaluation. We used a lightweight parse tree interface and toolkit of common, re-usable techniques to allow multiple automated analysis procedures to be applied to the specification within the same tool.

Our implementation includes a toolkit of techniques common to many configuration space analysis procedures. These include Boolean abstraction, binary decision diagrams, and automatic identification of constraints on previous and next configurations. We have illustrated how multiple analysis techniques can be applied in the framework by giving implementations of completeness, consistency, and symmetry checking of tables, simulation, and symbolic CTL model checking. Multiple analysis procedures were applied to each example specification.

In Chapter 7, we demonstrated that the structure of the specification has relevance for automated analysis. This use of structure contrasts with approaches that consider only the semantics of the specification. In particular, the structure of the specification found in the specifier's arrangement of items in a row in a tabular specification can be used to create a more precise abstraction for automated analysis.

The ease with which the most suitable notation for a part of the specification can be used in the framework is demonstrated through three examples: the heating system, the separation minima for aircraft in the North Atlantic Region, and the Aeronautical Telecommunications Network (ATN). The heating system was used throughout the dissertation for illustration. The separation minima and ATN examples were discussed in Chapter 8. Automated analysis techniques were applied to the three examples demonstrating this work. Analysis of the separation minima found three inconsistencies in the specification previously unknown to domain experts. Our implementation of symbolic functional evaluation performed well for specifications of substantial size, such as the ATN where the largest component (consisting of 18 pages of formal specification) took only 39 seconds to evaluate on an Ultra-Sparc 60. Analysis of the ATN illustrated errors in the formal specification and uncovered assumptions about the system previously unknown to the specifiers.

9.2 Generality

This section discusses the generality of our framework both in terms of adding new notations and adding new types of analysis.

9.2.1 Notations

Formal notations have a semantics. In our approach, we make direct use of these semantics to ensure that all forms of analysis use the same meaning for a specification. Existing descriptions of the semantics of many notations could be "codified" in higher-order logic to extend the framework to include these notations. For example, the work of Wang et al. [WRC97] on formalising and integrating the dynamic and object models of OMT could aid in bringing OMT into our framework.

Our framework could be extended to include Z [Spi88] based on the work of Bowen and Gordon [BG94], which describes a shallow embedding of Z in the object language of the HOL theorem prover [GM93]. Z is a well-known model-oriented specification notation, which has been used in a number of industrial examples. The predicates of a Z schema are relations between the previous and next configurations and therefore this notation falls into the model category. They use the ML interface language of HOL to translate ASCII representations of schemas in Z to higher-order logic. Schema-combining operators are also programmed in ML. Z is based on sets and Bowen and Gordon's embedding uses a representation of set theory within higher-order logic. Their method provides a means of integrating parts of a specification in Z with other notations in the framework. For example, Z could be used to describe one member of a set of concurrent components.

Our approach is limited to textual, non-symbolic representations of notations. There would be a greater gap between the Z notation and its textual representation than found in the example notations used so far in the framework.

Configuration space exploration analysis can be applied to Z as demonstrated in the Nitpick model checker [DJJ96]. In Z, the references to the values of previous and next configuration names are usually indicated through primed and non-primed identifiers of the same name. For analysis, an approach other than distinction based on configuration lifting would be needed in the toolkit to separate these constraints. The result would be that current analysis procedures implemented in the framework, such as model checking, could immediately be applied. The optimisations already demonstrated in Nitpick, such as reductions based on symmetry, could be added to extend this analysis. These extensions might be applicable to specifications written in other notations and become part of the toolkit. More general theorem proving reasoning for Z embedded in HOL has been implemented in the ICL ProofPower tool [PP].

The semantics for our example notation of a variant of statecharts could form the basis for including many state-transition notations in our framework. Our variant deals with multiple levels of hierarchy, concurrency, and priority. SCR includes only one level of hierarchy. RSML does not include priority of transitions at different levels in the hierarchy. The meaning of other notations, such as marked Petri nets, is often defined in terms of transition systems (e.g., Manna and Pnueli [MP92]). Many of the elements of the semantics of statecharts could be re-used to build the semantic functions for other notations to extend the framework.

Another example of a useful extension to the current set of notations of the framework would be an imperative language for actions, which would appeal to those more familiar with programming than specification. The use of uninterpreted constants lessens the chance of adding in too many design details when using a code-like specification notation.

9.2.2 Analysis

In terms of analysis, the choice of higher-order logic as the base formalism places few limits on the range of formal analysis methods that can be applied. Most formal methods-based analysis techniques are justifiable using deduction in higher-order logic. A toolkit of re-usable techniques helps bridge the gap between the general-purpose formalism and automated techniques, allowing for the easy integration of new analysis techniques. For example, the decision procedure of the Stanford Validity Checker [BD94, JDB95, BDL96] for a quantifier-free logic of equality with uninterpreted functions could be integrated into the framework. An abstraction technique other than Boolean abstraction would be needed since their decision procedure is applicable to more than just propositional logic. This addition would be immediately useful for completeness and consistency checking of tables and could improve the accuracy of output.

9.3 Limitations

This section discusses some limitations of our approach.

The framework is designed for notations that can be described using an operational semantics. Most model-oriented notations can be given an operational semantics. We have not yet explored the use of notations designed to specify complex data relationships. The approach will be less effective for notations that are not easily represented textually until appropriate interfaces to editors for those notations have been created.

Our framework inherits the limitations of the formal analysis techniques. Many of these techniques are restricted by time and space complexities for large specifications. The framework is mainly designed for analysis techniques that manipulate expressions and next configuration relations. We have not explored analysis techniques for the detection of inconsistencies among multiple models used together in a specification written in different notations, such as statecharts and Petri Nets.

The implementations of analysis tools potentially contain errors, which may detract from the rigour of the approach. The implementations of some analysis tools achieve rigour by using a central mechanism, such as a core set of inference rules, to minimise opportunities for implementation errors. Our tool, Fusion, achieves rigour by centralising the translation function for all notations in symbolic functional evaluation. The SFE step can be viewed as one large, core inference rule.

Application-independent checks provide a means of determining if a specification has certain standard properties. It is impossible to ensure that the set of application-dependent checks carried out are sufficient because requirements can only be compared to a person's intention for a system. Thus, no approach can eliminate all errors in the requirements specification.

Finally, use of the framework requires having a familiarity with the particular specification notations and at least limited knowledge of higher-order logic. Those familiar with functional programming languages would only require an additional understanding of uninterpreted constants. To carry out application-dependent checks, knowledge of the notation used to express the properties, such as CTL, is necessary. Formal methods expertise is required to change the semantics or to add a new notation or analysis technique.

9.4 Contributions

The main contribution of this work is an extensible, systematic, and rigorous framework for analysis of multi-notation, model-oriented specifications, which may include uninterpreted constants. Our work provides specifiers with the means to explore new options in the combinations of notations with immediate access to well-known automated analysis techniques. Furthermore, we achieve this result without using the infrastructure of a theorem prover.

Our framework achieves the desirable qualities through the use of operational semantics in higher-order logic. The choice of this core technology led us to use type checking as a mechanism for regulating combinations of notations. It also led us to develop symbolic functional evaluation, which is a rigorous method for determining the meaning of a specification in any notation. Thus, the key ingredients for creating our framework are higher-order logic, operational semantics, type checking and symbolic functional evaluation. The choice of these general-purpose techniques avoids a multiplicity of special-purpose tools for notations and analysis.

The framework is systematic in the regulation of the combinations of notations provided by the categorisation of notations and join points. It is also systematic in the separation of concerns between integrating the meaning of notations and carrying out analysis. It is rigorous in the direct use of semantic functions in higher-order logic. It is extensible in notations because of the packaging of notations with their semantics in embeddings. The toolkit of common analysis components aids in the extensibility of the architecture for analysis.

We have achieved a balance between the more general approach of Zave and Jackson [ZJ93], which considers all types of notations and overlapping specifications, and the more state-transition notation focus of Pezzè and Young [PY97]. Our method has the following advantages over automated translation techniques and other approaches to multi-notation analysis. The use of higher-order logic as the base formalism allows the use of uninterpreted constants and provides a formal foundation for analysis techniques to be applied to both specifications and semantics of notations. Packaged embeddings of notations and the semantic categories provide an extensible and systematic way to add new notations to the framework. Direct use of the semantics ensures all analysis techniques rely on the same meaning for the specification. Structure is preserved in symbolic functional evaluation and used in analysis. Finally, our architecture is easily extensible to new analysis techniques.

Our framework links a general-purpose formalism with automated analysis techniques, which allows us to de-couple specific notations from analysis. We obtain the benefits of multiple notations in a rigorous and extensible manner together with the benefits of automatic analysis.

Our framework eases the introduction of new notations and new combinations of notations for analysis. There does not appear to be any evidence to conclude that an ideal general-purpose requirements specification notation will ever be developed. Parnas and Madey note, "If history is a good predictor of the future, we will continue to invent improved forms of expressions as we discover new classes of functions to be of interest." [PM95].

9.5 Suggestions for future research

The research area of analysis of formal specifications is extremely active. This section discusses immediate questions resulting from this work.

9.5.1 Data specification

This work concentrates on model-oriented specifications. It has addressed the areas of combining control with functional behaviour. Many systems are dominated by their data complexity. Our demonstration of the framework is lacking in notations for data specification. We currently rely on higher-order logic to specify the data involved in the system using constant declarations. While S+ includes a close approximation of the specification of arrays and records, more structured notations such as ASN.1 and entity-relationship diagrams would be useful additions to the framework. Many languages include notations for data description. For example, LOTOS, a protocol description language, uses ACT ONE [CB87] for data descriptions. A means of capturing object hierarchies is necessary for notations such as UML and ObjectCharts [CHB92].

An open question is how well predicate logic can accommodate data specifications. As noted in Lamport and Paulson [LP97], set theory offers more flexibility than a typed notation. For example, sets consisting of objects of different types cannot be directly specified in higher-order logic. While typed systems offer the benefits of catching errors in the specification through type checking, set theory may be more convenient for specifying elements of object hierarchies such as classes and methods. Gordon offers a discussion on set theory and higher-order logic and possible combinations of the two [Gor96]. Gilmore's impredicative simple theory of types (ITT) [Gil98] offers an interesting alternative as a base formalism for our framework since it combines features of set theory and Church's higher-order logic. ITT makes a distinction between the use and mention of predicate names. Cooper's research investigates specification of more complex data relationships [Coo].

9.5.2 Symbolic simulation

A second area for exploration is that of simulation. Here we have presented a simple type of simulation that proved useful, but it does not provide all the facilities that we usually think of in terms of simulation. It involves the user knowing the number of steps to simulate at the beginning and there is no interaction during the simulation. Techniques that combine verification and simulation in one environment, such as symbolic trajectory evaluation (STE), rely on a functional presentation of the model where the behaviour of each variable is a function of previous values of variables [SB95]. However, STE limits the types of properties that can be analysed to gain efficiency. Symbolic model checking relies instead on a next configuration relation and can check properties of unknown duration. A next configuration relation and functional presentation of the specification often capture the same information. An interesting question for study is how to carry out interactive simulation and model checking within the same environment for specifications including uninterpreted constants.

9.5.3 Structure

This work has demonstrated in a small way that structure, not just semantics, can play a role in analysis by producing a more precise abstraction. The work on completeness and consistency checking by Heimdahl and Leveson [HL96] and by Heitmeyer et al. [HJL96] rely on structure to compose results of their analysis and to provide an overall statement of the completeness or consistency of a specification. Bharadwaj and Heitmeyer [BH97a] use structure-based techniques to reduce the size of the configuration space for model checking of specifications in SCR. Further work on using structure to compose results, reduce the size of configuration space, or aid in the choice of abstraction holds promise for handling scalability of formal analysis techniques.

9.5.4 Methodology

Research in formal methods is close to reaching a point of being readily applicable in industry. While scalability is still an issue for use on real problems, formal methods are being selectively applied to great advantage. Type checking and checking for application-independent properties help to isolate quickly areas of concern. As pointed out in the results of a survey of industry applications of formal methods, we need to determine how formal methods contribute to the overall development of systems [CGR95]. Guidelines for answers to the questions of "how much?", "how deeply?" and "where?" in the application of formal methods need to be provided. The use of uninterpreted constants can be a key ingredient in limiting the depth needed to apply formal methods. Contributions towards addressing these questions for application-independent properties can be found in Jaffe et al. [JL89, JLHM91], which describes criteria for checking whether specifications are complete. Simple efforts such as linking specifications with hypertext browsers, as we did automatically for the separation minima example, contribute towards the usability of formal methods [DJP97a, YAG98].

For the use of application-dependent techniques, there is currently less documented methodology. Perhaps the most pressing concern in applying techniques such as model checking is the lack of methodology. Work on specification patterns by Dwyer, Avrunin and Corbett [DAC98] begins to address this problem by providing a taxonomy of properties and mappings from informal statements of properties to formal ones in a variety of temporal logics. However, much still needs to be done in determining the relevant properties for a specification. The questions of what to check, what order to use for carrying out the checks, and when sufficient analysis has been done are important questions that when answered, even for particular classes of systems, will allow formal methods to take the next step forward.

9.6 A final word

We believe that the discipline of formal methods is in the midst of an evolutionary change that will be seen in the years to come as a new generation in its application to system development.

A decade ago, there was a wide chasm between specialised automated methods such as model checking, specification-intensive methods such as the use of Z, and general proof-based reasoning found in tools such as HOL. This first generation consisted of many exciting developments in the use of formal analysis to help in the validation and verification of a wide range of systems. Much of this work was based on early efforts by logicians and computer scientists. The benefits and limitations of these approaches were explored. Specific notations have proved suitable and been used successfully for real hardware verification efforts. However, requirements specifications necessitate the ability to express easily more abstract concepts. Research in requirements specification resulted in the development of precise, readable formal notations. Structured, graphical notations are being accepted and used in industry. To bridge the gap between requirements specification notations and notation-specific analysis techniques, in some cases translators were constructed.

The key characteristic of the second generation of formal methods-based analysis is the de-coupling of notation from analysis. The use of suitable, understandable notations is critical for formal methods to become accepted in industry. Even if an analysis technique is only applicable to an abstraction of a specification, a great deal of benefit can be derived from its use in our quest to improve system development through analysis. The second generation will include efforts to link multiple kinds of analysis to a range of systems specified using different notations. Having made significant progress in developing useful techniques in the first generation, researchers will explore the range of applicability of these techniques and provide guidelines for their use in system development. The second generation opens up formal methods to non-specialists in providing an extensible set of notations together with automated analysis.

As a contribution to the second generation, we provide a framework that is parameterised by notation through the use of semantic functions. A general-purpose notation provides the formal basis for their integration. Symbolic functional evaluation exposes the meaning of the specification for analysis. Using a toolkit of re-usable components, automated analysis can be applied to requirements specifications. Extending the framework with new notations provides immediate access to automated analysis. We achieve the benefits of multiple notations for specification without sacrificing automation in analysis.

Bibliography

[AB96] Joanne M. Atlee and Michael A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[ABB+96] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, 1996.

[ACD97] George S. Avrunin, James C. Corbett, and Laura K. Dillon. Analyzing partially-implemented real-time systems. In *19th International Conference on Software Engineering*, pages 228–238. ACM Press, 1997.

[ADJ97] J. H. Andrews, N. A. Day, and J. J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *FORTE/PSTV'97 (1997 IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification)*, November 1997.

[AG93] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–

40, January 1993.

[AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[AL93] Martín Abadi and Leslie Lamport. Conjoining specifications. Technical Report 118, Digital Systems Research Center, December 1993.

[And97] James H. Andrews. Executing formal specifications by translating to higher order logic programming. In *1997 International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, number 1275 in Lecture Notes in Computer Science, pages 17–32. Springer Verlag, August 1997. Bell Labs, New Jersey.

[ATN96] ATN Panel, Working Group 3, International Civil Aviation Organization. *Aeronautical Telecommunications Network Panel: Draft SARPs and Guidance Material for ATN Upper Layers for the CNS/ATM-1 Package*, 1996.

[BCM+90] J. R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *6th International Conference, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–79, Stanford, California, June 1994. Springer-Verlag.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture*

Notes in Computer Science, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[BG94] Jonathan P. Bowen and Michael J. C. Gordon. Z and HOL. In J. Anthony Hall, editor, *Proceedings of the 8th Annual Z User Meeting*, Workshops in Computing, St. John's College, University of Cambridge, Cambridge, England, 29–30 June 1994. BCS-FACS, Springer-Verlag, London.

[BGG$^+$92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129–156, Amsterdam, 1992. North-Holland.

[BH97a] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state space exploration. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[BH97b] Ramesh Bharadwaj and Constance Heitmeyer. Applying the SCR requirements method to a simple autopilot. In *Lfm97: Fourth NASA Langley Formal Methods Workshop*, pages 87–99. NASA Conference Publication 3356, September 1997.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*, volume 23. Academic Press, Inc., Boston, 1988.

[Boo91] Grady Booch. *Object Oriented Design.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Cam88] Albert John Camilleri. Simulation as an aid to verification using the HOL theorem prover. Technical Report 150, University of Cambridge Computer Laboratory, October 1988.

[CB87] The SPECS Consortium and J. Bruijning. Evaluation and integration of specification languages. *Computer Networks and ISDN Systems*, 13(2):75–89, 1987.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGR95] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21:90–98, February 1995.

[CHB92] Derek Coleman, Fiona Hayes, and Stephan Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[Che80] Brian F. Chellas. *Modal logic: an introduction.* Cambridge University Press, Cambridge, 1980.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[COHH92] Rachel Cardell-Oliver, Roger Hale, and John Herbert. An embedding of timed transition systems in HOL. In *Higher Order Logic Theorem Proving and its Applications*, pages 263–278, Leuven, Belgium, Sept 1992.

[Coo] Kendra Cooper. PhD work in preparation entitled *SPECL: A Formal Specification Language*, Department of Electrical Engineering, University of British Columbia, expected completion 1999.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of FMSP'98. The Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, March 1998.

[Day93] Nancy Day. A model checker for statecharts. Master's thesis, Department of Computer Science, University of British Columbia, 1993. Available as Technical Report 93-35.

[DeL71] Howard DeLong. *A Profile of Mathematical Logic.* Addison-Wesley, Reading, Massachusetts, 1971.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[DJJ96] Craig A. Damon, Daniel Jackson, and Somesh Jha. Checking relational specifications with binary decision diagrams. In *Proceedings of the Fourth ACM SIGSOFT Symposim on the Foundations of Software Engineering*, pages 70–80, 1996.

[DJP97a] Nancy A. Day, Jeffrey J. Joyce, and Gerry Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Lfm97:*

Fourth NASA Langley Formal Methods Workshop, pages 35–49. NASA Conference Publication 3356, September 1997.

[DJP97b] Nancy A. Day, Jeffrey J. Joyce, and Gerry Pelletier. Formalization and analysis of the separation minima for the North Atlantic Region: Complete specification and analysis results. Technical Report 97-12, Department of Computer Science, University of British Columbia, October 1997.

[Don98] Michael R. Donat. *A Discipline of Specification-Based Test Derivation.* PhD thesis, Department of Computer Science, University of British Columbia, 1998.

[DS95] Heping Dai and C. Keith Scott. AVAT, a CASE tool for software verification and validation. In *Proceedings of the IEEE 7th International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 358–367, 1995.

[Dur98] David M. Durham. JPL Virtual Technical Tour (presentation handout). In *8th Annual International Symposium of the International Council on Systems Engineering*, July 1998.

[EC97] Steve Easterbrook and John Callahan. Formal methods for V & V of partial specifications: An experience report. In *Third IEEE International Symposium on Requirements Engineering (RE'97)*, pages 160–168, Annapolis, MD, 1997.

[EK94] Dirk Eisenbiegler and Ramayya Kumar. Evaluation techniques as a part of the verification procress. In *1994 International Conference on Higher Order Logic Theorem Proving and its Applications: Supplementary Proceedings*, 1994.

[Fau95] Stuart R. Faulk. Software requirements: A tutorial. Technical Report NRL/MR/5546–95-7775, Naval Research Laboratory, November 1995.

[GG77] David Gries and Narain Gehani. Some ideas on data types in high-level languages. *Communications of the ACM*, 20(6):414–420, June 1977.

[Gil98] Paul Gilmore. An impredicate simple theory of types. In *14th Workshop on the Mathematical Foundations for Programming Systems*, 1998.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[Goo93] K.G.W. Goossens. Operational semantics based formal symbolic simulation. In *Higher Order Logic Theorem Proving and its Applications*, pages 487–506. North-Holland, 1993.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

[Gor85] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, September 1985.

[Gor87] Mike Gordon. A proof generating system for higher-order logic. Technical Report No. 103, University of Cambridge Computer Laboratory, January 1987.

[Gor88a] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In Graham M. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[Gor88b] Michael J. C. Gordon. *Programming Language Theory and its Implementation.* International Series in Computer Science. Prentice Hall, New York, 1988.

[Gor96] Mike Gordon. Set theory, higher order logic or both? In *The 1996 International Conference on Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 191–202. Springer-Verlag, 1996.

[Han87] Peter Hancock. Polymorphic type-checking. In *The Implementation of Functional Programming Languages*, International Series in Computer Science, chapter 8, pages 139–162. Prentice Hall, New York, 1987.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 109–122, New York, June 1995. IEEE.

[HC95] D.N. Hoover and Z. Chen. Tablewise, a decision table tool. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 97–108, New York, June 1995. IEEE.

[HDDY93] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop, 1992*. Springer-Verlag, 1993.

[Hei94] Mats Per Erik Heimdahl. *Static Analysis of State-Based Requirements Analysis for Completeness and Consistency*. PhD thesis, University of California, Irvine, 1994.

[Hei96] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 79–83, January 1996. San Diego.

[Hen80] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HL$^+$90] David Harel, H. Lachover, et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HL93] Constance L. Heitmeyer and Bruce G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report NRL/FR/5540-93-9586, United States Naval Research Laboratory, Washington, D.C., December 1993.

[HL96] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[HLCM92] C.L. Heitmeyer, B.G. Labaw, P.C. Clements, and A.K. Mok. Engineering CASE tools to support formal methods for real-time software development. In *5th International Workshop on Computer-Aided Software Engineering*, pages 110–113, 1992.

[HLK95] Constance Heitmeyer, Bruce G. Labaw, and Daniel Kiskis. Consistency checks of SCR-style requirements specifications. In *Proceedings International Symposium on Requirements Engineering*, pages 56–63, March 1995. York, England.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming. In *OOPSLA*, pages 411–428, 1993.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, New York, June 1987.

[Hu95] Alan John Hu. *Techniques for efficient formal verification using binary decision diagrams.* PhD thesis, Department of Computer Science, Stanford University, 1995.

[i-L91] i-Logix Inc., Burlington, MA. *The Semantics of Statecharts*, January 1991.

[Int94] International Organization for Standardization. *ACSE Protocol, ITU-T Rec. X. 227 – ISO/IEC 8650-1: Edition 2*, 1994.

[Jac95] Michael Jackson. *Software Requirements and Specifications: a lexicon of practice, principles and prejudices.* Addison-Wesley Publishing Company, Wokingham, England, 1995.

[JDB95] Robert B. Jones, David L. Dill, and Jerry R. Burch. Efficient validity checking for processor verification. In *Proceedings of the 1995 International Conference on Computer-Aided Design*, 1995.

[JDD94] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *7th International Workshop on Higher Order*

Logic Theorem Proving and Its Applications, pages 285–299, Valletta, Malta, September 1994. Springer-Verlag.

[JL89] Matthew S. Jaffe and Nancy G. Leveson. Completeness, robustness, and safety in real-time software requirements specification. In *11th International Conference on Software Engineering*, pages 302–311, May 1989. Pittsburgh, Pennsylvania.

[JLHM91] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–259, March 1991.

[JM94] Farnam Jahanian and Aloysius K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, New York, 1987.

[Joy89] Jeffrey Joyce. *Multi-Level Verification of Microprocessor Based Systems*. PhD thesis, University of Cambridge Computer Laboratory, 1989. Technical Report 195.

[JS88] Farnam Jahanian and Douglas A. Stuart. A method for verifying properties of modechart specifications. In *1988 Symposium on Real-Time Systems*, pages 12–21, 1988.

[JS93] J. Joyce and C-J. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th Design Automation Conference*. IEEE Computer Press, June 1993.

[Kap97] Bruce Kapron. Private communication, February 1997.

[KM97] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[LBH+95] Nancy Leveson, Ken Bauer, Mats Heimdahl, Wayne Ohlrich, Kurt Partidge, Vivek Rata, and Jon Reese. A CAD environment for safety-critical software. University of Washington Safety-Critical Systems Project, July 1995.

[LHHR94] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[Lon] David E. Long. bdd - a binary decision diagram (BDD) package. Man page.

[LP97] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? Technical Report 147, Digital Systems Research Center, May 1997.

[Lut97] Robyn R. Lutz. Reuse of a formal model for requirements validation. In *Lfm97: Fourth NASA Langley Formal Methods Workshop*, pages 75–85, September 1997. NASA Conference Publication 3356.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[Nes93] Monica Nesi. Value-passing CCS in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, pages 352–365. LNCS 780, Springer-Verlag, 1993.

[NVG92] Sanjiv Narayan, Frank Vahid, and Daniel D. Gajski. Modeling with speccharts. Technical Report 90-20, Dept. of Information and Computer Science, University of California, Irvine, October 1992.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[ORS96] S. Owre, J.M. Rushby, and Nataranjan Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, April 1996.

[ORS97] Sam Owre, John Rushby, and Natarajan Shankar. Integration in PVS: Tables, types, and model checking. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 336–383. Spinger-Verlag, April 1997.

[Par92] David Lorge Parnas. Tabular representations of relations. Technical Report 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, October 1992.

[Par93a] D. L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[Par93b] David Lorge Parnas. Some theorems we should prove. In *Higher Order Logic Theorem Proving and Its Applications*, pages 155–162. LNCS 780, Springer-Verlag, 1993.

[Par94] David Parnas. Inspection of safety-critical software using program-function tables. In *13th IFIP World Computer Congress*, pages 270–277, 1994.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Plo81] Gordon D. Plotkin. A structural approach to operation semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, September 1981. Reprinted April 1991.

[PM95] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[PP] Proofpower server. Send email to ProofPower-server@win.icl.co.uk.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, vol.526, pages 244–264. Springer-Verlag, 1991.

[PY97] Mauro Pezzè and Michal Young. Constructing multi-formalism state-space analysis tools. In *19th International Conference on Software Engineering*, pages 239–249. ACM Press, 1997.

[R$^+$91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Raj93] P. Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In *Higher Order Logic Theorem Proving and its Applications*, pages 527–535. North-Holland, 1993.

[Raj95] P. Sreeranga Rajan. *Transformations on Data Flow Graphs: Axiomatic Specification and Efficient Mechanical Verification.* PhD thesis, University of British Columbia, 1995.

[Rat] Rational Software. *UML Notation Guide: Version 1.1.* Available at "www.rational.com/uml/html/notation".

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *Seventh Workshop on Computer-Aided Verification*, Liege, Belgium, July 1995.

[Rus97] John Rushby. Subtypes for specifications. In *Fifth ACM Foundations of Software Engineering*, September 1997. Revised version of invited paper.

[SA96a] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements: A case study. Technical Report CS96-05, Department of Computer Science, University of Waterloo, 1996.

[SA96b] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements: A case study. In *Proceedings of the 11th Annual Conference on Computer Assurance*, June 1996.

[SB95] C.-J. H. Seger and R. E. Bryant. Formal verification of partially-ordered trajectories. *Formal Methods in Systems Design*, 6:147–189, March 1995.

[Seg93] Carl-Johan H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, December 1993.

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, pages 15–24, November 1990.

[SJ92] Carl-Johan H. Seger and Jeffrey J. Joyce. A mathematically precise two-level formal hardware verification methodology. Technical Report 92-34, University of British Columbia, Department of Computer Science, December 1992.

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag, Berlin, 1968.

[Spi88] J.M. Spivey. *Understanding Z.* Cambridge University Press, Cambridge, 1988.

[Sta94] Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[Tan88] Andrew S. Tanenbaum. *Computer Networks, Second Edition.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Tas93] John P. Van Tassel. A formalization of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications*, pages 359–374. North-Holland, 1993.

[vdB94] Michael von der Beeck. A comparison of statecharts variants. In *Proc. of 3rd Int. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT '94)*, volume 863, pages 128–148. LNCS, Springer, 1994.

[VG93] Myra VanInwegan and Elsa Gunter. HOL-ML. In *Higher Order Logic Theorem Proving and Its Applications*, pages 61–74. LNCS 780, Springer-Verlag, 1993.

[Win90] P. J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambrige, Massachusetts, 1993.

[WR27] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, England, 1927. 2nd edition.

[WRC97] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *19th International Conference on Software Engineering*, pages 45–55. ACM Press, May 1997.

[YAG98] Richard Yates, James H. Andrews, and Phil Gray. Practical experience applying formal methods to air traffic management software. In *International Council on Systems Engineering (INCOSE'98)*, July 1998.

[ZJ93] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

Appendix A

Heating System Specification

```
/*
 Specification of Heating System
 (loosely based on Chapter 8, "Object Oriented Design"
      by Grady Booch,
      The Benjamin/Cummings Publishing Company,
      1991)
*/

/* import all semantic definition files */
%addpath ..
%include all.s+

/* type declarations and definitions */
: Room := KITCHEN | LIVING_ROOM | BEDROOM;

: Behaviour :=
    NOT_OCCUPIED |
    EXPECT_SOON |
    EXPECT_NOW ;
: behaviour == config -> Behaviour ;

: Valve_Pos := OPEN | HALF | CLOSED ;
: valve_Pos == config -> Valve_Pos;

/* outputs of the system */
valvePos : Room -> valve_Pos;

/* internal synchronization between components */
requestHeat : Room -> bool;
```

```
/* user inputs */
livingPattern : Room -> behaviour;
setTemp : Room -> num;
heatSwitchOn, heatSwitchOff, userReset : simpleEvent;

/* system specific parameters */
/* time to wait before adjusting valve again */
warmUpTime, coolDownTime : num;
furnaceStartupTime: num;

/* inputs from the environment */
actualTemp : Room -> num;
occupied : Room -> bool;

activate, deactivate,
furnaceFault, furnaceReset,
furnaceRunning, furnaceNotRunning : simpleEvent;

/* states and transition of the system */
: stateName :=
 HEATING_SYSTEM |
    FURNACE |
       FURNACE_NORMAL |
          FURNACE_OFF |
          FURNACE_ACTIVATING |
          FURNACE_RUNNING |
       FURNACE_ERROR |

    ROOM :Room |
       NO_HEAT_REQUESTED :Room |
          IDLE_NO_HEAT :Room |
          WAIT_FOR_HEAT :Room |
       HEAT_REQUESTED :Room |
          IDLE_HEATING :Room |
          WAIT_FOR_COOL :Room |

    CONTROLLER |
       OFF |
       ERROR |
       CONTROLLER_ON |
          IDLE |
          HEATER_ACTIVE |
               ACTIVATING_HEATER |
               HEATER_RUNNING ;
```

```
:transName :=
   T1 | T2 | T3 | T4 | T6 | T7 |
   T8 :Room | T9 :Room | T10 :Room | T11 :Room | T12 :Room |
   T13 :Room | T14 :Room | T15 :Room |
   T16 | T17 | T18 | T19 | T20 | T21 | T22  ;

/* room statechart */

/* shorthands */
sT := setTemp;
aT := actualTemp;

desiredTemp (i:Room) :=
Table
[ Row (occupied i)  [ True; False ; False ];
  Row (livingPattern i)
     [  Dc;
        (\x.x = C EXPECT_NOW) ;
        (\x.x = C EXPECT_SOON)] ]
[ sT i; sT i; sT i - C 5; sT i - C 10 ];

dT := desiredTemp;

/*
 optionally leave dT unintepreted, replace above with
 dT : Room -> num;
*/

/* shorthands */
tooCold i := (dT i - aT i) > C 2;
tooHot i := (aT i - dT i) > C 2;
vOpen i := valvePos i = C OPEN;
vClosed i := valvePos i = C CLOSED;

/* setting the valve position */
nextVp i :=
Table
[Row (dT i - aT i)
      [ (\x.x < C -5);
        (\x. C -5 <= x AND x < C -2);
        (\x. C -5 <= x AND x < C -2);
        (\x. C -2 <= x AND x <= C 2);
        (\x. C 2 < x AND x <= C 5);
        (\x. C 2 < x AND x <= C 5) ;
        (\x. C 5 < x)];
 Row (valvePos i)
      [ Dc ;
        (\x. x = C OPEN) ;
        (\x. x = C HALF) ;
        Dc ;
        (\x. x = C CLOSED) ;
        (\x. x = C HALF) ;
        Dc] ]
[C CLOSED; C HALF; C CLOSED; valvePos i; C HALF; C OPEN; C OPEN];

/* shorthands */
rH i := Asn (requestHeat i) (C T);
cancelrH i := Asn (requestHeat i) (C F);
adjValve i := Asn (valvePos i) (nextVp i);

roomSc (i:Room) :=
let waitedForWarm :=
      \i.Tm (En (WAIT_FOR_HEAT i) (roomSc i)) warmUpTime in
let waitedForCool :=
      \i. Tm (En (WAIT_FOR_COOL i) (roomSc i)) coolDownTime in
   OrState (ROOM i) (NO_HEAT_REQUESTED i)
   [OrState (NO_HEAT_REQUESTED i) (IDLE_NO_HEAT i)
      [BasicState (IDLE_NO_HEAT i);
       BasicState (WAIT_FOR_HEAT i)]
      [(T8 i,IDLE_NO_HEAT i,EvCond NonEvent (tooCold i),
           adjValve i,WAIT_FOR_HEAT i);
       (T9 i,WAIT_FOR_HEAT i,EvCond NonEvent (NOT (tooCold i)),
           NoAction, IDLE_NO_HEAT i);
       (T10 i, WAIT_FOR_HEAT i, waitedForWarm i,
           adjValve i, WAIT_FOR_HEAT i)] ;
    OrState (HEAT_REQUESTED i) (IDLE_HEATING i)
      [BasicState (IDLE_HEATING i);
       BasicState (WAIT_FOR_COOL i)]
      [(T15 i,IDLE_HEATING i,EvCond NonEvent (tooHot i),
           adjValve i,WAIT_FOR_COOL i);
       (T14 i,WAIT_FOR_COOL i,EvCond NonEvent (NOT(tooHot i)),
           NoAction, IDLE_HEATING i);
       (T13 i, WAIT_FOR_COOL i, waitedForCool i,
           adjValve i, WAIT_FOR_COOL i)] ]
   [(T11 i, WAIT_FOR_COOL i, EvCond (waitedForCool i) (vClosed i),
            cancelrH i, NO_HEAT_REQUESTED i);
    (T12 i, WAIT_FOR_HEAT i, EvCond (waitedForWarm i) (vOpen i),
           rH i, HEAT_REQUESTED i)];
```

```
/* furnace statechart */

furnaceSc :=
    OrState FURNACE FURNACE_NORMAL
    [BasicState FURNACE_ERROR;
     OrState FURNACE_NORMAL FURNACE_OFF
     [BasicState FURNACE_OFF;
      BasicState FURNACE_ACTIVATING ;
      BasicState FURNACE_RUNNING  ]
     [(T1,
        FURNACE_OFF,
        Ev activate,
        NoAction,
        FURNACE_ACTIVATING);
      (T2,
        FURNACE_ACTIVATING,
        Ev deactivate,
        NoAction,
        FURNACE_OFF);
      (T3,
        FURNACE_ACTIVATING,
        Tm (En FURNACE_ACTIVATING furnaceSc) furnaceStartupTime,
        Gen furnaceRunning,
        FURNACE_RUNNING);
      (T4,
        FURNACE_RUNNING,
        Ev deactivate,
        NoAction,
        FURNACE_OFF) ] ]
     [(T7,
        FURNACE_NORMAL,
        Ev furnaceFault,
        NoAction,
        FURNACE_ERROR);
      (T6,
        FURNACE_ERROR,
        Ev furnaceReset,
        NoAction,
        FURNACE_NORMAL)];


/* central controller */

/* shorthands */
```

```
roomNeedsHeat := exists i. requestHeat i;
noRoomsNeedHeat := NOT (roomNeedsHeat);

controllerSc :=
  OrState CONTROLLER OFF
  [BasicState OFF;
   BasicState ERROR;
   OrState CONTROLLER_ON IDLE
      [BasicState IDLE;
       OrState HEATER_ACTIVE ACTIVATING_HEATER
       [BasicState ACTIVATING_HEATER;
        BasicState HEATER_RUNNING]
       [(T21, ACTIVATING_HEATER, Ev furnaceRunning,
              NoAction, HEATER_RUNNING)] ]
      [(T20, IDLE, EvCond NonEvent (roomNeedsHeat),
              Gen activate, HEATER_ACTIVE);
       (T22, HEATER_ACTIVE, EvCond NonEvent (noRoomsNeedHeat),
              Gen deactivate, IDLE ) ] ]
   [(T16, ERROR, Ev userReset, Gen furnaceReset, OFF);
    (T17, OFF, Ev heatSwitchOn, NoAction, CONTROLLER_ON);
    (T18, CONTROLLER_ON, Ev heatSwitchOff, Gen deactivate, OFF);
    (T19, CONTROLLER_ON, Ev furnaceFault, NoAction, ERROR)];

/* Heating System */

heatingSystemScStruct :=
  AndState HEATING_SYSTEM
  [ roomSc (KITCHEN);
    roomSc (BEDROOM);
    roomSc (LIVING_ROOM);
    furnaceSc;
    controllerSc];

HeatingSystem := Sc heatingSystemScStruct;
```

Appendix B

Built-in constants of S+

```
(:ty) LET (x:ty) := x;
(:ty)FORALL :(ty -> BOOL) -> BOOL;
(:ty)EXISTS :(ty -> BOOL) -> BOOL;
(_ /\ _) : BOOL -> BOOL -> BOOL;
(_ \/ _) : BOOL -> BOOL -> BOOL;
~ : BOOL -> BOOL;
a (_ AND _) b (cf:config) := (a cf) /\ (b cf);
a (_ OR _) b (cf:config) := (a cf) \/ (b cf);
NOT a (cf:config) := ~(a cf);
(:ty) COND (T) (a:ty) b := a | COND (F) a b := b;
a (_ ==> _) b := (NOT a) OR b+
(ty) C (x:ty) (cf:config) := x;
(:ty1,:ty2) FST :(ty1#ty2)->ty1;
(:ty1,:ty2) SND :(ty1#ty2)->ty2;
(_ PLUS 12 _) : NUM->NUM->NUM;
a (_ + 12 _) b (cf:config) := (a cf) PLUS (b cf);
(_ MULT 14 _) : NUM->NUM->NUM;
a (_ * 14 _) b (cf:config) := (a cf) MULT (b cf);
(_ MINUS 12 _) : NUM->NUM->NUM;
a (_ - 12 _) b (cf:config) := (a cf) MINUS (b cf);
(_ DIV 14 _) : NUM->NUM->NUM;
a (_ / 14 _) b (cf:config) := (a cf) DIV (b cf);
(_ iMOD 14 _) : NUM->NUM->NUM;
a (_ MOD 14 _) b (cf:config) := (a cf) iMOD (b cf);
(:ty) (_ EQ 2 _) : ty->ty->BOOL;
(:ty) (a:config->ty) (_ = 2 _) b cf:= (a cf) EQ (b cf);
(:ty) (a:config->ty) (_ != 2 _) b := NOT (a = b);
(_ GREATER_THAN 10 _) : NUM -> NUM -> BOOL;
a (_ > 10 _) b (cf:config) := (a cf) GREATER_THAN (b cf);
```

```
(_ LESS_THAN 10 _) : NUM -> NUM -> BOOL;
a (_ < 10 _) b (cf:config) := (a cf) LESS_THAN (b cf);
(_ GREATER_THAN_OR_EQUAL 10 _) : NUM -> NUM -> BOOL;
a (_ >= 10 _) b (cf:config) := (a cf) GREATER_THAN_OR_EQUAL (b cf);
(_ LESS_THAN_OR_EQUAL 10 _) : NUM -> NUM -> BOOL;
a (_ <= 10 _) b (cf:config) := (a cf) LESS_THAN_OR_EQUAL (b cf);
iABS: NUM -> NUM;
ABS a (cf:config) := iABS (a cf);
(:ty)NAME:ty->STRING;
```

Appendix C

Types for Semantic Categories

```
/*
 Types for semantic categories
*/

/* model */
:step == config # config;
Prev (x:step) := FST x;
Next (x:step) := SND x;
:model == step -> BOOL;

/* step extended to include extra information */
:(label)ext_step == (config # config) # (label->BOOL);
(:label)ExtPrev (x:(label)ext_step) := FST (FST x);
(:label)ExtNext (x:(label)ext_step) := SND (FST x);

/* expressions  -------------------------------------- */
:(ty)exp == config -> ty;

/* events -------------------------------------------- */

:(label)eventinfo ==
     (bool #                            /* Occurred */
     (BOOL->(label)ext_step -> BOOL) #  /* Update */
     ((label)ext_step -> BOOL) #        /* Occurs */
     (BOOL-> bool) #                    /* Init */
     bool);                             /* OccursAtInit */

: eventLabelPath := S | L | R;
: path == (eventLabelPath)list;

: (label) event ==
     label -> path -> (label)eventinfo;

(:label)
Occurred (ev:(label)event) (lab:label) (p:path)
    := FST (ev lab p);

(:label)
Update (ev:(label)event) (lab:label) (p:path)
    := FST(SND (ev lab p));

(:label)
Occurs (ev:(label)event) (lab:label) (p:path)
    := FST(SND(SND (ev lab p)));

(:label)
Init (ev:(label)event) (lab:label) (p:path)
    := FST(SND(SND(SND (ev lab p))));

(:label)
OccursAtInit (ev:(label)event) (lab:label) (p:path)
    := SND(SND(SND(SND (ev lab p))));

/* event labelling */


(:label)Sub path := append path [S];
(:label)Left path := append path [L];
(:label)Right path := append path [R];

/* actions ---------------------------------------------*/

: mod == (STRING # model # model);

: action == (mod )list;


/* useful shorthand */
(:t)NoConstraintcfcf' (step:(t)ext_step) := T;
```

Appendix D

Basic definitions

```
/*
  basics.s+
*/

(:ty)UNKNOWN:ty;

/* Operations on Lists */

(:ty)
length (NIL) := 0 |
length (CONS (e:ty) l) := 1 PLUS length l;

(:ty)
hd (CONS (e:ty) l) := e;

(:ty)
tl (NIL) := NIL |
tl (CONS (e:ty) l) := l ;

(:ty1,:ty2)
map (CONS a b) (f:ty1->ty2) := CONS (f a) (map b f) |
map (NIL) f := NIL;

(:ty)
append NIL l := l |
append (CONS (e:ty) l) j := CONS e (append l j) ;

(:ty)
flatten (NIL) := NIL |
flatten (CONS (h:(ty)list) t) := append h (flatten t);

/* Operations on lists whose elements are lifted */

(:ty)
everyAux (NIL) (p:ty->BOOL) := T |
everyAux (CONS e l) p := (p e) /\ everyAux l p;

(:ty)
every (p:ty->BOOL) l := everyAux l p;

(:ty)
anyAux (NIL) (p:ty->BOOL) := F |
anyAux (CONS e l) p := (p e) \/ anyAux l p;

(:ty)
any (p:ty->BOOL) l := anyAux l p;

(:ty)
(a:ty) (_ member _) b :=
    COND  (b EQ NIL)
    F
    (COND (a EQ hd b) T (a member (tl b)));

(:ty)
lastElement (CONS (a:ty) b) :=
    COND (b EQ NIL) a (lastElement b);

(:ty)
repeats NIL := NIL |
repeats (CONS (a:ty) b) :=
    COND (a member b)
    (CONS a (repeats b))
    (repeats b);

(:ty)
allFalse NIL (charfcn:ty->BOOL) := T |
allFalse (CONS (a:ty) t) charfcn :=
    ~(charfcn a) /\ allFalse t charfcn;

/* know there is at least one element */
(:ty)
sizeIsOneAux NIL (charfcn:ty->BOOL) := T |
sizeIsOneAux (CONS (a:ty) t) charfcn :=
    COND (t EQ NIL) (charfcn a)
    (((charfcn a) /\ allFalse t charfcn) \/
      (~(charfcn a) /\ sizeIsOneAux t charfcn));

(:ty)
sizeIsOne NIL (charfcn:ty->BOOL) := F |
sizeIsOne (CONS a b) charfcn :=
    sizeIsOneAux (CONS a b) charfcn;

/* Lifting constructors and pairs  */

(:ty)
C (a:ty) (cf:config) := a;

(:ty1,:ty2)
P (a:config->ty1,b:config->ty2) (cf:config) := (a cf, b cf);

/* Operations on lists whose elements are lifted */

(:ty)
EveryAux NIL p := C T |
EveryAux (CONS a b) (p:ty->bool) := p a AND EveryAux b p;

(:ty)
Every (p:ty->bool) l := EveryAux l p;

(:ty)
AnyAux NIL p := C F |
AnyAux (CONS a b) (p:ty->bool) := p a OR AnyAux b p;

(:ty)
Any (p:ty->bool) l := AnyAux l p;

/*
 Operations over sets that are represented by
 explicit characteristic functions as lists of (*,bool) pairs
 These are all lifted operations (i.e. relative to a particular
 configuration).
*/

/* creating a set of this form */
(:ty)
MakeCharFcnSet NIL (cfcn:ty->bool) := NIL |
MakeCharFcnSet (CONS h t) cfcn :=
     CONS (h,cfcn h) (MakeCharFcnSet t cfcn);
```

```
(:ty)
SubsetAux NIL (a:ty#bool) := NOT(SND a) |
SubsetAux (CONS h t) a :=
    COND (FST a EQ FST h) (SND a ==>  SND h) (SubsetAux t a);

(:ty)
(a:(ty#bool)list) (_ Subset _) b :=
   Every (\el.SubsetAux b el) a;

(:ty)
Empty NIL := C T |
Empty (CONS (h:ty#bool) t) := (NOT (SND h)) AND  Empty t ;

(:ty)
IntersectAux (NIL) (a:ty#bool) := NIL |
IntersectAux (CONS h t) a :=
    COND (FST a EQ FST h)
    [(FST a,SND a AND SND h)]
    (IntersectAux t a);

(:ty)
(a:(ty#bool)list) (_ Intersect _) b :=
    COND (a EQ NIL)
    NIL
    (append (IntersectAux b (hd a)) ((tl a) Intersect b));
```

Appendix E

TableExpr

```
/*
 TableExpr notation
 Produces (ty)exp's
 Keywords:
     row labels: AllOf, AtLeastOneOf
     row entries: Dc, True, False
     rows: Row
     tables: Table, PredicateTable
*/

: (ty) rowlabel == (ty)exp;

: (ty) rowentry == ((ty)exp)->bool;

: row == (bool)list;

/* produces a rowentry */
(:ty)
Dc := \(x:(ty)exp).C T;

/* produces a rowentry */
True := \x.(x = (C T));

/* produces a rowentry */
False := \x.(x = (C F));

/* produces a rowlabel */
(:ty)
AllOf (rl:((ty)exp)list) (p:(ty)exp->bool) := Every p rl;
```

```
/* produces a rowlabel */
(:ty)
AtLeastOneOf (rl:((ty)exp)list) (p:(ty)exp->bool) :=
    Any p rl;

(:ty)
RowAux NIL (rl:(ty)rowlabel) := NIL |
RowAux (CONS (re:(ty)rowentry) res) rl :=
CONS (re rl) (RowAux res rl);

/* produces a row */
(:ty)
Row (rl:(ty)rowlabel) (res:((ty)rowentry)list) :=
   RowAux res rl;

/* produces a list of columns, type: (bool)list */
Columns (rowMatrix:(row)list) :=
     /* last column is a list of empty lists */
    COND ((hd rowMatrix) EQ NIL) NIL
    (CONS (Every (hd) rowMatrix) (Columns (map rowMatrix (tl))));

(:ty)
TableAux (NIL) (resultRow:((ty)exp)list) :=
    COND (resultRow EQ NIL) (UNKNOWN) (hd resultRow)  |
TableAux (CONS col cols) resultRow :=
    if col
    then (hd resultRow)
    else TableAux cols (tl resultRow);

(:ty)
Table (rowMatrix:(row)list) (resultRow:((ty)exp)list) :=
    TableAux (Columns rowMatrix) resultRow;

PredicateTable (rowMatrix:(row)list) :=
    Any (\x.x) (Columns rowMatrix);

(:ty)
WellFormedTable (rowMatrix:(row)list) (resultRow:((ty)exp)list) :=
    ~(rowMatrix EQ NIL) /\
    (let lenFirstRow := length (hd rowMatrix) in
       /* all rows are the same length */
    every (\x.length x EQ lenFirstRow) (tl rowMatrix) /\
       /* result row is same length as rows, or one more (default) */
    ((length resultRow EQ lenFirstRow)
         \/ (length resultRow EQ (lenFirstRow PLUS 1))));

WellFormedPredicateTable (rowMatrix:(row)list) :=
    let lenFirstRow := length (hd rowMatrix) in
    /* all rows are the same length */
    every (\x.length x EQ lenFirstRow) (tl rowMatrix);
```

Appendix F

CoreEvent

```
/*
  CoreEvents: event notational style
  Keywords: NonEvent, Ev, EvCond, OrE, AndE, Tm, TmB
*/


/* Primitives */

(:label)
TimeEventLastOccurred : label -> path -> config -> NUM;

(:label)
Changed : label -> path -> bool;

/* NonEvent ----------------------------------------------- */

(:label)
NonEventUpdate (flag:BOOL) (step:(label)ext_step) := T;

NonEventInit (flag:BOOL) (cf:config) := T;

(:label)
NonEvent (lab:label) (p:path) :=
    (C T,
     (:label)NonEventUpdate,
     (:label)NoConstraintcfcf',
     NonEventInit,
     C T);
```

```
/* Ev simpleEvent --------------------------------------- */

/* wellformedness constraints on this */

: simpleEvent == bool;

(:label)
EvOccurs (ev:simpleEvent) (step:(label)ext_step) := ev (ExtNext step);

(:label)
EvUpdate (flag:BOOL) (step:(label)ext_step) := T;

EvOccursAtInit (ev:simpleEvent) (cf:config) := ev cf;

EvInit (ev:simpleEvent) (flag:BOOL) (cf:config) := T;

(:label)
Ev (ev:simpleEvent) (lab:label) (p:path) :=
    (ev,
     ((:label)EvUpdate),
     ((:label)EvOccurs) ev,
     EvInit ev,
     EvOccursAtInit ev);

/* Ch condition -------------------------------------------- */

(:label)
ChOccurs (cond:bool) (step:(label)ext_step) :=
    ~(cond (ExtPrev step) EQ cond (ExtNext step));

(:label)
ChOccurred (lab:label) (p:path) (cf:config) :=
    Changed lab p cf;

(:label)
ChUpdate (cond:bool) (lab:label) (p:path)
  (flag:BOOL) (step:(label)ext_step) :=
    (~flag) \/
    (Changed lab p (ExtNext step) EQ ChOccurs cond step);

(:label)
Ch (cond:bool) (lab:label) (p:path) :=
   (ChOccurred lab p,
    ChUpdate cond lab p,
    (:label)ChOccurs cond,
    \(flag:BOOL).\(cf:config).T,
    \(cf:config). F);

/* EvCond --------------------------------------------------- */

(:label)
EvCondOccurred (ev:event) (b:bool)
 (lab:label) (p:path) (cf:config) :=
   ((:label)Occurred) ev lab (Sub p) cf /\ b cf;

(:label)
EvCondOccurs (ev:event) (b:bool) (lab:label) (p:path)
          (step:(label)ext_step) :=
   Occurs ev lab (Sub p) step /\ b (ExtNext step);

(:label)
EvCondOccursAtInit (ev:(label)event) (b:bool) (lab:label) (p:path) :=
    ((:label)OccursAtInit) ev lab (Sub p) AND b;

(:label)
EvCond (ev:(label)event) (b:bool) (lab:label) (p:path) :=
    (EvCondOccurred ev b lab p,
     Update ev lab (Sub p),
     (:label)EvCondOccurs ev b lab p,
     Init ev lab (Sub p),
     ((:label)EvCondOccursAtInit) ev b lab p);

/* AndEv ---------------------------------------------------- */

(:label)
AndOccurred (ev1:event) (ev2:event) (lab:label) (p:path) :=
    ((:label)Occurred) ev1 lab (Left p) AND
    ((:label)Occurred) ev2 lab (Right p);

(:label)
AndUpdate (ev1:event) (ev2:event) (lab:label) (p:path) (flag:BOOL)
      (step:(label)ext_step) :=
    ((:label)Update) ev1 lab (Left p) flag step /\
    ((:label)Update) ev2 lab (Right p) flag step;

(:label)
AndOccurs (ev1:event) (ev2: event) (lab:label) (p:path)
        (step:(label)ext_step) :=
    ((:label)Occurs) ev1 lab (Left p) step /\
    ((:label)Occurs) ev2 lab (Right p) step;
```

```
(:label)
AndInit (ev1:event) (ev2:event) (lab:label) (p:path) (flag:BOOL) :=
    ((:label)Init) ev1 lab (Left p) flag AND
    ((:label)Init) ev2 lab (Right p) flag;

(:label)
AndOccursAtInit (ev1:event) (ev2:event) (lab:label) (p:path) :=
    ((:label)OccursAtInit) ev1 lab (Left p) AND
    ((:label)OccursAtInit) ev2 lab (Right p);

(:label)
AndE (ev1:event) (ev2:event) (lab:label) (p:path) :=
    (AndOccurred ev1 ev2 lab p,
     AndUpdate ev1 ev2 lab p,
     (:label)AndOccurs ev1 ev2 lab p,
     AndInit ev1 ev2 lab p,
     AndOccursAtInit ev1 ev2 lab p);


/* OrEv -------------------------------------------------- */

(:label)
OrOccurred (ev1:event) (ev2:event) (lab:label) (p:path) :=
    (:label)Occurred ev1 lab (Left p) OR
    (:label)Occurred ev2 lab (Right p);

(:label)
OrOccurs (ev1:event) (ev2: event) (lab:label) (p:path)
  (step:(label)ext_step) :=
   Occurs ev1 lab (Left p) step \/
   Occurs ev2 lab (Right p) step;

(:label)
OrOccursAtInit (ev1:event) (ev2:event) (lab:label) (p:path) :=
    (:label)OccursAtInit ev1 lab (Left p) OR
    (:label)OccursAtInit ev2 lab (Right p);


(:label)
OrE (ev1:event) (ev2:event) (lab:label) (p:path) :=
    (OrOccurred ev1 ev2 lab p,
     ((:label)AndUpdate) ev1 ev2 lab p,    /* same as for AND */
     ((:label)OrOccurs) ev1 ev2 lab p,
     AndInit ev1 ev2 lab p,
     OrOccursAtInit ev1 ev2 lab p);

/* Tm ------------------------------------------------------ */

(:label)
TmOccurred (n:num) (lab:label) (p:path) (cf:config) :=
    (TimeEventLastOccurred lab p = n) cf ;

(:label)
TmUpdate (ev:event) (lab:label) (p:path) (flag:BOOL)
    (step:(label)ext_step) :=
    ((Occurs ev lab (Sub p) step /\
        ((TimeEventLastOccurred lab p = (C 0)) (ExtNext step))))
    \/
    (~(Occurs ev lab (Sub p) step) /\
       (TimeEventLastOccurred lab p (ExtNext step) EQ
         ((TimeEventLastOccurred lab p + (C 1)) (ExtPrev step))))
    /\
    Update ev lab (Sub p) F step;

(:label)
TmOccurs (d:num) (lab:label) (p:path) (step:(label)ext_step) :=
   (TimeEventLastOccurred lab p = d ) (ExtNext step);

(:label)
TmOccursAtInit (ev:event) (n:num) (lab:label) (p:path) (cf:config) :=
    ((n cf EQ 0) /\ ((:label)OccursAtInit ev lab (Sub p) cf));

(:label)
TmInit (ev:event) (n:num) (lab:label) (p:path)
       (flag:BOOL) (cf:config) :=
    (TmOccursAtInit ev n lab (Sub p) cf EQ
          ((TimeEventLastOccurred lab p = (C 0)) cf ))
    /\ (:label)Init ev lab (Sub p) F cf;

(:label)
Tm (ev:event) (n:num) (lab:label) (p:path) :=
    ((:label)TmOccurred n lab p,
     (:label)TmUpdate ev lab p,
     (:label)TmOccurs n lab p,
     TmInit ev n lab p,
     TmOccursAtInit ev n lab p);

/* TmB ------------------------------------------------- */
/* uses some bit vector operations defined elsewhere */

(:label)
TimeEventLastOccurredBPos: label -> path -> NUM -> bool;

(:label)
TimeEventLastOccurredB (n:NUM) (lab:label) (p:path) (cf:config) :=
    COND (n EQ 0) NIL
     (CONS (TimeEventLastOccurredBPos lab p n cf)
        (TimeEventLastOccurredB (n MINUS 1) lab p cf));

BV NIL (cf:config) := NIL |
BV (CONS (h:bool) t) cf :=
    CONS (h cf) (BV t cf);

(:label)
TmBEventOccurred (n:NUM) (delay:(bool)list) (lab:label) (p:path)
   (cf:config) :=
    /* counter is not at its maximum value */
    (~(MaxValueBV (TimeEventLastOccurredB n lab p cf))) /\
    ((TimeEventLastOccurredB n lab p cf) EqualBV (BV delay cf));

(:label)
TmBOccurs (n:NUM) (delay:(bool)list) (lab:label) (p:path)
    (step:(label)ext_step) :=
     TmBEventOccurred n delay lab p (ExtNext step);

Inc (b:(BOOL)list) (c:(BOOL)list):=
    ((MaxValueBV b) /\ (b EqualBV c)) \/
    ((~(MaxValueBV b)) /\ (c EqualBV (IncBV b)));

(:label)
TmBUpdate (n:NUM) (ev:event) (lab:label) (p:path)
    (step:(label)ext_step) :=
    (Occurs ev lab (Sub p) step /\
      ResetBV (TimeEventLastOccurredB n lab p (ExtNext step)))
    \/
    ((~(Occurs ev lab (Sub p) step)) /\
        Inc (TimeEventLastOccurredB n lab p (ExtPrev step))
            (TimeEventLastOccurredB n lab p (ExtNext step)));

(:label)
TmBEventOccursAtInit (n:NUM) (ev:event) (delay:(bool)list)
  (lab:label) (p:path) (cf:config) :=
    ((ResetBV(BV delay cf)) /\ (OccursAtInit ev lab p cf));
```

```
(:label)
TmBInit (n:NUM) (ev:event) (delay:(bool)list) (lab:label) (p:path)
   (flag:BOOL) (cf:config) :=
    (TmBEventOccursAtInit n ev delay lab p cf EQ
       (ResetBV(TimeEventLastOccurredB n lab p cf)))
    /\ Init ev lab (Sub p) F cf;

(:label)
TmB (n:NUM) (ev:event) (delay:(bool)list) (lab:label) (p:path) :=
    ((:label)TmBEventOccurred n delay lab p,
     (:label)TmBUpdate n ev lab p,
     TmBOccurs n delay lab p,
     (:label)TmBInit n ev delay lab p,
     (:label)TmBEventOccursAtInit n ev delay lab p);
```

Appendix G

CoreAction

```
/*
  CoreActions
  Keywords: NoAction, Asn, Both, Gen
*/

/* NoAction ------------------------------------------ */

NoAction := ((:STRING # model # model)NIL);

/* Assignment ---------------------------------------- */

/* wellformedness constraints limit v */

(:ty)
AsnChange (v:(ty)exp) (exp:(ty)exp) (step:step) :=
     v (Next step) EQ exp (Prev step);

(:ty)
AsnNoChange (v: (ty)exp) (step:step) :=
     v (Next step) EQ v (Prev step);

(:ty)
Asn (v:(ty)exp) (exp:(ty)exp) :=
     [(NAME v,
       (:ty)AsnChange v exp,
       (:ty)AsnNoChange v
      )];

/* Both ---------------------------------------------- */
```

```
Both (a1:action) (a2:action) :=
    append a1 a2;

/* Gen ---------------------------------------------- */

Gen (ev:bool) :=
    [(NAME ev,
     \(step:step).ev (Next step) EQ T,
     \(step:step).ev (Next step) EQ F)];
```

Appendix H

CoreSc

H.1 Accessor functions

```
/*
  CoreSc: statechart notational style
  Accessor functions for parts of structure
  Keywords: AndState, OrState, BasicState
*/

/* Textual Representation */

/* specialized version of events - label is transName */

: (stateName,transName) trans ==
    transName#
    stateName#
    (transName)event #
    action #
    stateName;

/* optimisation - associate flag with transition */

: (stateName,transName) transrec ==
    transName#
    stateName#
    (transName)event #
    action #
    stateName #
    BOOL;
```

```
: (stateName,transName) sc_struct :=
    OR_STATE :stateName
             :stateName
             :((stateName,transName)sc_struct)list
         :((stateName, transName)trans)list
  | AND_STATE :stateName :((stateName,transName)sc_struct)list
  | BASIC_STATE :stateName;

(:stateName,:transName)OrState := (:stateName,:transName)OR_STATE;
(:stateName,:transName)AndState := (:stateName,:transName)AND_STATE;
(:stateName,:transName)BasicState := (:stateName,:transName)BASIC_STATE;

/* Accessor Functions for elements of a transition ------------------ */

(:stateName,:transName)
transLabel (a:(stateName,transName)transrec) := FST a;

(:stateName,:transName)
transLabel1 (a:(stateName,transName)trans) := FST a;

(:stateName,:transName)
transSrc (a:(stateName,transName)transrec) := FST(SND a);

(:stateName,:transName)
transSrc2 (a:(stateName,transName)trans) := FST(SND a);

(:stateName,:transName)
transEvent (a: (stateName,transName)transrec) := FST(SND(SND a));

(:stateName,:transName)
transEvent1 (a: (stateName,transName)trans) := FST(SND(SND a));

(:stateName,:transName)
transAction (a:(stateName,transName)transrec) := FST(SND(SND(SND a)));

(:stateName,:transName)
transDest (a:(stateName,transName)transrec) :=
    FST(SND (SND (SND (SND a))));

(:stateName,:transName)
transDest2 (a:(stateName,transName)trans) :=
    SND (SND (SND (SND a)));

(:stateName,:transName)
transFlag (a:(stateName,transName)transrec) :=
    SND (SND (SND (SND (SND a))));

/*
 Information about state and state hierarchy
 given a state description return the following information about it
*/

(:stateName,:transName)
isBasicState
    (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := F |
isBasicState (AND_STATE stn substates) := F |
isBasicState (BASIC_STATE stn) := T;

(:stateName,:transName)
isAndState
    (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := F |
isAndState (AND_STATE stn substates) := T |
isAndState (BASIC_STATE stn) := F;

(:stateName,:transName)
isOrState
    (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := T |
isOrState (AND_STATE stn substates) := F |
isOrState (BASIC_STATE stn) := F;

(:stateName,:transName)
stateName
    (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := stn |
stateName(AND_STATE stn substates) := stn |
stateName(BASIC_STATE stn) := stn ;

/* returns list of state descriptions */
(:stateName,:transName)
stateSubstates
    (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := substates |
stateSubstates(AND_STATE stn substates) := substates ;

/*
 returns a state description
*/
```

```
(:stateName,:transName)
getStateFromStateList NIL fcn def := NIL |
getStateFromStateList (CONS (h:(stateName,transName)sc_struct) t)
 fcn def :=
    COND (stateName h EQ def)
       [h]
       (COND (fcn h def EQ NIL)
          (getStateFromStateList t fcn def)
          (fcn h def));

/*
 given a state name returns the state description with this
 state at the root
*/

(:stateName,:transName)
stateAux(OR_STATE stn def substates
  (trans:((stateName,transName)trans)list)) stName :=
    COND (stName EQ stn)
    [(OR_STATE stn def substates trans)]
(getStateFromStateList substates stateAux stName) |
stateAux (AND_STATE stn substates) stName :=
    COND (stName EQ stn)
    [(AND_STATE stn substates)]
(getStateFromStateList substates stateAux stName) |
stateAux (BASIC_STATE stn) stName :=
    COND (stName EQ stn)
    [(BASIC_STATE stn)]
    NIL;

(:stateName,:transName)
state (s:(stateName,transName)sc_struct) stn := hd (stateAux s stn);

(:stateName,:transName)
StateDefault
 (OR_STATE stn def substates (trans:((stateName,transName)trans)list))
    := hd(getStateFromStateList substates stateAux def);

/* --------------------------------------------------------------- */

(:stateName,:transName)
findPathinList NIL findPathfcn stn := NIL |
findPathinList (CONS (a:(stateName,transName)sc_struct) b)
  (findPathfcn:(stateName,transName)sc_struct
                  -> stateName -> (stateName)list)
    (stn:stateName) :=
    COND (findPathfcn a stn EQ NIL)
       (findPathinList b findPathfcn stn)
       (findPathfcn a stn);

(:stateName,:transName)
findPath
    (OR_STATE stn def substates
         (trans:((stateName,transName)trans)list)) st :=
    COND (stn EQ st)
      [stn]
      (COND ((findPathinList substates findPath st) EQ NIL)
         NIL
         (CONS stn (findPathinList substates findPath st))) |
findPath (AND_STATE stn substates) st :=
    COND (stn EQ st)
      [stn]
      (COND ((findPathinList substates findPath st) EQ NIL)
         NIL
         (CONS stn (findPathinList substates findPath st))) |
findPath (BASIC_STATE stn) st :=
    COND (st EQ stn) [st] NIL;

/* -------------------------------------------------------------- */

(:stateName)
scope (CONS (h:stateName) t) destpath :=
    COND (~(hd t EQ hd (tl destpath)) \/
           (tl t EQ NIL) \/ (tl(tl destpath) EQ NIL))
[h]
        (CONS h (scope t (tl destpath)));

(:stateName,:transName)
transScope(tr:(stateName,transName)transrec)
         (s:(stateName,transName)sc_struct) :=
    let path1 := findPath s (transSrc tr) in
    let path2 := findPath s (transDest tr) in
    scope path1 path2 ;

/* ------------------------------------------------------------- */

(:stateName,:transName)
basicStates
    (OR_STATE stn def substates
                 (trans:((stateName,transName)trans)list)) :=
    flatten (map substates basicStates) |
basicStates (AND_STATE stn substates) :=
    flatten (map substates basicStates) |
basicStates (BASIC_STATE stn) := [stn];

/* -------------------------------------------------------------- */

(:stateName,:transName)
transInState
    (OR_STATE stn def substates
        (trans:((stateName,transName)trans)list)) :=
    append trans (flatten((map substates (transInState)))) |
transInState (AND_STATE stn substates) :=
    flatten(map substates transInState) |
transInState (BASIC_STATE stn) := [];

/* -------------------------------------------------------------- */

(:stateName,:transName)
exitBasicStates
    (OR_STATE stn def substates
         (trans:((stateName,transName)trans)list)) :=
    flatten (map substates exitBasicStates) |
exitBasicStates (AND_STATE stn substates) :=
    flatten (map substates exitBasicStates) |
exitBasicStates(BASIC_STATE stn) := [stn];

(:stateName,:transName)
basicStatesExited (s:(stateName,transName)sc_struct) stn :=
    exitBasicStates (state s stn) ;

/* ------------------------------------------------------------ */

(:stateName,:transName)
enterBasicStates
    (OR_STATE stn def substates
            (trans:((stateName,transName)trans)list)) :=
    enterBasicStates
        (hd(getStateFromStateList substates stateAux def)) |
enterBasicStates (AND_STATE stn substates) :=
    flatten (map substates enterBasicStates) |
enterBasicStates (BASIC_STATE stn) := [stn];

(:stateName,:transName)
```

```
basicStatesEntered (s:(stateName,transName)sc_struct) stn :=
    enterBasicStates (state s stn);

/* used by En, Ex events */

(:ty)
differingSuffixes (src:(ty)list,dest) :=
    COND (~(hd src EQ hd dest))
    (src,dest)
    (COND ((tl src EQ NIL) \/ (tl dest EQ NIL))
        (src,dest)
        (differingSuffixes(tl src,tl dest)));

(:stateName,:transName)
statesEnteredBelowAux
    (OR_STATE stn def substates
         (trans:((stateName,transName)trans)list)) :=
    CONS stn (statesEnteredBelowAux
         (hd(getStateFromStateList substates stateAux def))) |
statesEnteredBelowAux (AND_STATE stn substates) :=
    CONS stn (flatten (map substates statesEnteredBelowAux)) |
statesEnteredBelowAux(BASIC_STATE stn) := [stn];

(:stateName,:transName)
statesEnteredBelow(s:(stateName,transName)sc_struct) stn :=
    statesEnteredBelowAux (state s stn);

(:stateName,:transName)
statesExitedBelowAux (OR_STATE stn def substates
(trans:((stateName,transName)trans)list)) :=
    CONS stn (flatten (map substates statesExitedBelowAux)) |
statesExitedBelowAux (AND_STATE stn substates) :=
    CONS stn (flatten (map substates statesExitedBelowAux)) |
statesExitedBelowAux (BASIC_STATE stn) := [stn];

(:stateName,:transName)
statesExitedBelow (s:(stateName,transName)sc_struct) stn :=
    statesExitedBelowAux (state s stn);

(:stateName,:transName)
getFullExitPath h (s:(stateName,transName)sc_struct) :=
    COND (tl h EQ NIL)
      (statesExitedBelow s (hd h))
      (getFullExitPath (tl h) s);
```

```
(:stateName,:transName)
getFullEnterPath h (s:(stateName,transName)sc_struct) :=
    COND (tl h EQ NIL)
      (statesEnteredBelow s (hd h))
      (CONS (hd h) (getFullEnterPath (tl h) s));

/*
 given two paths find point at which they differ
 return those two parts suffixed with anything
 it takes to get to basic states
*/
(:stateName,:transName)
getEntEx (fullsrcP, fulldestP) (s:(stateName,transName)sc_struct) :=
    let (srcP,destP) := differingSuffixes (fullsrcP,fulldestP) in
    (srcP,getFullExitPath srcP s,getFullEnterPath destP s);

(:stateName,:transName)
statesEntered (s:(stateName,transName)sc_struct)
         (t:(stateName,transName)trans) :=
    let p2src := findPath s (transSrc2 t) in
    let p2dest := findPath s (transDest2 t) in
    let (ex1,ex2,ent) := getEntEx (p2src,p2dest) s in
    ent;

(:stateName,:transName)
statesExited (s:(stateName,transName)sc_struct)
        (t:(stateName,transName)trans) :=
    let p2src := findPath s (transSrc2 t) in
    let p2dest := findPath s (transDest2 t) in
    let (ex1,ex2,ent) := getEntEx (p2src,p2dest) s in
append ex1 ex2;
```

H.2 Semantics

```
/*
  CoreSc: semantics
  Keywords: Sc
*/

/* Primitives --------------------------------------- */

(:stateName)InBasicState : stateName -> (BOOL)exp;

/* -------------------------------------------------- */
/* optimisation */
(:stateName,:transName)
TransTakenInt (tr:(stateName,transName)transrec)
  (step:(transName)ext_step) :=
    transFlag tr;

(:stateName,:transName)
TransTaken (tr:(stateName,transName)trans) (step:(transName)ext_step) :=
    (SND step) (transLabel1 tr);


/* interface to event notation */
(:label)
EventUpdate (ev:(label)event) (lab:label) (p:path)
    := FST(SND (ev lab p)) T;

(:label)
EventInit (e:(label)event) (lab:label) (p:path)
    := FST(SND(SND(SND (e lab p) ))) T;

/* Transition Condition ------------------------------ */

(:stateName,:transName)
inAnyBasicState cf (s:(stateName,transName)sc_struct) :=
    COND (isBasicState s) (InBasicState (stateName s) cf )
      (COND (isAndState s)
        (every (inAnyBasicState cf) (stateSubstates s))
        (any (inAnyBasicState cf) (stateSubstates s)));

(:stateName,:transName)
Instate stName (s:(stateName,transName)sc_struct) (cf:config) :=
    inAnyBasicState cf (state s stName) ;

(:stateName,:transName)
Enabled (s:(stateName,transName)sc_struct)
  (tr:(stateName,transName)transrec) :=
    Instate (transSrc tr) s AND
    Occurred (transEvent tr) (transLabel tr) [] ;

(:stateName,:transName)
oneTransTaken (trlist:((stateName,transName)transrec)list)
      (step:(transName)ext_step) :=
    sizeIsOne (trlist) (\tr.TransTakenInt tr step);
```

```
(:stateName,:transName)
stateChange (s:(stateName,transName)sc_struct)
     (tr:(stateName,transName)transrec)
     (step:(transName)ext_step) :=
    let allBasicStatesInScope :=
        basicStatesExited s (stateName s) in
    let basicStatesEnt :=
        basicStatesEntered s (transDest tr) in
    every
      (\stn. COND (stn member basicStatesEnt)
                  (InBasicState stn (ExtNext step))
                  (~(InBasicState stn (ExtNext step))))
         allBasicStatesInScope;

(:stateName,:transName)
oneEnabledTransIsTaken (s:(stateName,transName)sc_struct)
  (trlist:(transrec)list)
       (step:(transName)ext_step) :=
    oneTransTaken (trlist) step  /\
    every
    (\tr. ~(TransTakenInt tr step)  \/
           (Enabled s tr (ExtPrev step) /\
           (stateChange s tr step)))
    trlist;

(:stateName,:transName)
noEnabledTrans (s:(stateName,transName)sc_struct)
  (trlist:(transrec)list) (step:(transName)ext_step) :=
   (every (\tr. ~(Enabled s tr (ExtPrev step))) trlist) /\
   (every (\tr. ~(TransTakenInt tr step)) trlist);

(:stateName,:transName)
thisScope NIL := (NIL,NIL) |
thisScope
  (CONS (x:(stateName)list,y:(stateName,transName)transrec) b) :=
    let (here,rest) := thisScope b in
    COND (tl x EQ NIL)
        (CONS y here,rest)
        (here,CONS (x,y) rest);

(:stateName,:transName)
scopeWithin NIL (stn:stateName) := NIL |
scopeWithin
  (CONS (x:(stateName)list,y:(stateName,transName)transrec) b) stn :=
    COND (hd (tl x) EQ stn)
```

```
       (CONS (tl x,y) (scopeWithin b stn))
       (scopeWithin b stn);

(:stateName,:transName)
noTransAreTaken
   (rest:(((stateName)list)#(stateName,transName)transrec)list)
   (step:(transName)ext_step) :=
    every (\tr.~(TransTakenInt (SND tr) step )) rest;

(:stateName,:transName)
TransStateCondAux (s:(stateName,transName)sc_struct) trs
  (step:(transName)ext_step) :=
    COND (isBasicState s)
/* scopetranslist must be NIL */
    (InBasicState (stateName s) (ExtNext step) EQ
        InBasicState (stateName s) (ExtPrev step))
    (COND (isAndState s)
        (every (\sub. TransStateCondAux sub
                         (scopeWithin trs (stateName sub)) step)
              (stateSubstates s))
        /* OrState */
       (let (trsThisLevel,rest) := thisScope trs  in
          (oneEnabledTransIsTaken s trsThisLevel step   /\
           (noTransAreTaken rest step))
            \/
          (noEnabledTrans s trsThisLevel step /\
          (every (\sub.TransStateCondAux sub
                         (scopeWithin rest (stateName sub)) step)
          (stateSubstates s)))));

(:stateName,:transName)
TransStateCond (s:(stateName,transName)sc_struct) trs step :=
    TransStateCondAux s (map trs (\t.(transScope t s,t))) step;

/* Event Condition ------------------------------------------ */

(:stateName,:transName)
EventCond (s:(stateName,transName)sc_struct)
       (trs:((stateName,transName)transrec)list) step :=
     every (\tr.EventUpdate (transEvent tr)
            (transLabel tr) [] step) trs;

/* Name Condition ----------------------------------------- */

/*
```

```
 returns a list of (transition,action) pairs
*/
(:stateName,:transName)
allLocalMods (NIL)  := NIL |
allLocalMods (CONS (tr:(stateName,transName)transrec) t)  :=
    append
       (map (transAction tr ) (\x.(tr,x)))
       (allLocalMods t);

(:stateName,:transName)
modTrans
   (a:(stateName,transName)transrec, b:STRING, c:model, d:model) := a;

(:stateName,:transName)
modName
   (a:(stateName,transName)transrec, b:STRING, c:model, d:model) := b;

(:stateName,:transName)
modNoChange
   (a:(stateName,transName)transrec, b:STRING, c:model, d:model) := d;

(:stateName,:transName)
modChange
   (a:(stateName,transName)transrec, b:STRING, c:model, d:model) := c;

(:stateName,:transName)
divideMatches NIL a := (NIL,NIL) |
divideMatches
    (CONS (b:(stateName,transName)transrec#mod) c)
    (a:STRING) :=
    let (matches,nonmatches) := divideMatches c a in
    COND (a EQ modName b)
/* put just the transition and the modification element on this list */
    (CONS (modTrans b, modChange b) matches, nonmatches)
      (matches, CONS b nonmatches);

(:stateName,:transName)
groupedModificationsAux NIL := NIL |
groupedModificationsAux
    (CONS (a:(stateName,transName)transrec # mod) b) :=
    let varname := modName a in
    let (matches,nonmatches) := divideMatches b varname in
    CONS
       (varname,modNoChange a,(CONS (modTrans a,modChange a) matches))
       (groupedModificationsAux nonmatches);
```

```
(:stateName,:transName)
groupedModifications (trs:((stateName,transName)transrec)list) :=
groupedModificationsAux (allLocalMods trs);

: (stateName,transName)namerec ==
   STRING # model # ((stateName,transName)transrec # model)list;

(:stateName,:transName)
nameTransList (vr:(stateName,transName)namerec) := map (SND(SND vr)) FST;

(:stateName,:transName)
nameNoChange (vr:(stateName,transName)namerec) := FST(SND vr);

(:stateName,:transName)
nameModList (vr:(stateName,transName)namerec) := SND(SND vr);

(:stateName,:transName)
NameCond
    (s:(stateName,transName)sc_struct)
    (trs:((stateName,transName)transrec)list)
    (step:(transName)ext_step) :=
  every
  (\namerec.
   (every
      (\tr. ~(TransTakenInt tr step))
      (nameTransList namerec)
    /\
    nameNoChange namerec (ExtPrev step, ExtNext step))
   \/
   (any
      (\(tr,md) . TransTakenInt tr step /\
                  md (ExtPrev step, ExtNext step))
      (nameModList namerec)))
  (groupedModifications trs);


/* Putting it all together --------------------------------- */

(:ty)
existsn_aux n (p:(BOOL)list->BOOL) list :=
    COND
    (n EQ 0)
    (p list)
    (EXISTS (\trn'. existsn_aux (n MINUS 1) p (append list [trn'])));
```

```
(:ty)
existsn n (p:(BOOL)list->BOOL)  :=
     existsn_aux n p NIL;

(:t1)
match (NIL) x a := T |
match (CONS (h:t1) t) (flags:(BOOL)list) a :=
    COND (h EQ a) (hd flags) (match t (tl flags) a);


(:stateName,:transName)
ScAux (s:(stateName,transName)sc_struct)
      (trs:((stateName,transName)transrec)list)
      (step:(transName)ext_step) :=
    NameCond s trs step  /\
    EventCond s trs step /\
    TransStateCond s trs step ;

/* have to get the associatively correct */

(:stateName,:transName)
trans2TransRec (a:(stateName,transName)trans) (b:BOOL) :=
    (FST a,
     FST(SND a),
     FST(SND(SND a)),
     FST(SND(SND(SND a))),
     SND(SND(SND(SND a))),
     b);

(:stateName,:transName)
pairs NIL a := NIL |
pairs (CONS (h:(stateName,transName)trans) t) j :=
    COND (j EQ NIL) NIL (CONS (trans2TransRec h (hd j))
                                  (pairs t (tl j)));

(:stateName,:transName)
Sc (s:(stateName,transName)sc_struct) (stp:step) :=
   let trs := transInState s in
   (existsn (length trs)
   (\flags.
   (let ext_step := (stp, match (map trs transLabel1) flags) in
   ScAux s (pairs trs flags) ext_step)));

/* initialisation ----------------------------------------- */
```

```
(:stateName,:transName)
InitStates (s:(stateName,transName)sc_struct) :=
    Every
    (\bsn.
    COND (bsn member (basicStatesEntered s (stateName s)))
       (InBasicState bsn)
       (NOT(InBasicState bsn)))
    (basicStates s);

(:stateName,:transName)
InitEvents (s:(stateName,transName)sc_struct) cf :=
    every
    (\t. (EventInit (transEvent1 t) (transLabel1 t) [] cf ))
    (transInState s);

(:stateName,:transName)
InitialCond (s:(stateName,transName)sc_struct) :=
    InitStates s AND InitEvents s;
```

Appendix I

ScExpr

```
/*
  ScExpr: expression notation specific for statecharts
  Depends on statechart notational style having been loaded.
  Keywords: InState
*/

(:stateName,:transName)
InState stName (s:(stateName,transName)sc_struct) := Instate stName s ;
```

Appendix J

ScEvent

```
/*
  ScEvents: event notation specific for statecharts
  Depends on statechart notational style having been loaded.
  Keywords: En, Ex
*/

/* Primitives */

(:stateName)
EnJustOccurred :stateName -> bool;

(:stateName)
ExJustOccurred :stateName -> bool;

/* En stn s - entered a state -------------------------------- */

(:stateName)
EnOccurred (stn:stateName) cf :=
    EnJustOccurred stn cf;

(:stateName,:transName)
EnOccurs stn (s:(stateName,transName)sc_struct)
   (step:(transName)ext_step) :=
    let alltrans := transInState s in
    any
    (\t. TransTaken t step /\ (stn member (statesEntered s t))) alltrans ;

(:stateName,:transName)
EnUpdate stn (s:(stateName,transName)sc_struct) (flag:BOOL)
```

```
             (step:(transName)ext_step) :=
    ~flag \/
    (EnJustOccurred stn (ExtNext step) EQ EnOccurs stn s step);

(:stateName,:transName)
EnOccursAtInit stn (s:(stateName,transName)sc_struct) :=
    Instate stn s;

(:stateName,:transName)
EnInit stn  (s:(stateName,transName)sc_struct) (flag:BOOL)
  (cf:config) :=
    ~flag \/ (EnJustOccurred stn cf EQ EnOccursAtInit stn s cf);

/* produces an event */
(:stateName,:transName)
En stn (s:(stateName,transName)sc_struct) (lab:transName) (p:path) :=
    (EnOccurred stn,
     EnUpdate stn s,
     (:stateName,:transName)EnOccurs stn s,
     EnInit stn s,
     EnOccursAtInit stn s);

/* Ex stn s ------------------------------------------------ */

(:stateName)
ExOccurred (stn:stateName) cf :=
    ExJustOccurred stn cf;

(:stateName,:transName)
ExOccurs stn (s:(stateName,transName)sc_struct)
  (step:(transName)ext_step) :=
    let alltrans := transInState s in
    any
    (\t. TransTaken t step /\
          (stn member statesExited s t)) alltrans;

(:stateName,:transName)
ExUpdate stn (s:(stateName,transName)sc_struct) (flag:BOOL)
    (step:(transName)ext_step) :=
    ~flag \/
    (ExJustOccurred stn (ExtNext step) EQ ExOccurs stn s step);

(:stateName,:transName)
ExInit (stn:stateName) (s:(stateName,transName)sc_struct)
    (flag:BOOL) (cf:config) :=
    ~flag \/ ~(ExJustOccurred stn cf);

/* produces an event */
(:stateName,:transName)
Ex stn (s:(stateName,transName)sc_struct) (lab:transName) (p:path) :=
    (ExOccurred stn,
     ExUpdate stn s,
     (:stateName,:transName)ExOccurs stn s,
     ExInit stn s,
     (C F));
```

Appendix K

Extended Communication for CoreSc

K.1 Primitives

```
/*
  Primitives for extended sc communication constructs
*/

(:stateName,:msg)
Msg : stateName -> stateName -> msg -> bool;

(:stateName,:msg,:ty)
Data: stateName -> stateName -> msg ->(ty)exp;
```

K.2 CommEvent

```
/*
  CommEvents: statechart communication events
  Depends on statechart notational style and
    statechart communication primitives having been loaded.
  Keywords: ReceiveData, Receive
*/

(:stateName,:transName,:msg)
ReceiveDataOccurred
```

```
        (src:stateName)
        (dest:stateName)
        (ms:msg)
        (cf:config) :=
   Msg src dest ms cf;

(:stateName,:transName,:msg)
ReceiveDataOccurs
        (src:stateName)
        (dest:stateName)
        (ms:msg)
        (step:(transName)ext_step) :=
   Msg src dest ms (ExtNext step) ;

/* nothing to do because Send sets Msg to be true or false */
(:stateName,:transName,:msg,:ty)
ReceiveDataUpdate
    (src:stateName)
    (dest:stateName)
    (ms:msg)
    (data:(ty)exp)
    (flag:BOOL)
    (step:(transName)ext_step) :=
  data (ExtNext step) EQ Data src dest ms (ExtNext step);

(:stateName,:transName,:msg)
ReceiveDataInit (src:stateName)
        (dest:stateName)
        (ms:msg)
        (flag:BOOL)
        (cf:config) := T;

(:stateName,:transName,:msg,:ty)
ReceiveData
    (s:(stateName,transName)sc_struct)
    (src:stateName)
    (ms:msg)
    (data:(ty)exp)
    (x:transName)
    (n:path) :=
 (ReceiveDataOccurred src (stateName s) ms,
  (:stateName,:transName,:msg,:ty)
       ReceiveDataUpdate src (stateName s) ms data,
  (:stateName,:transName,:msg)
       ReceiveDataOccurs src (stateName s) ms,
  ReceiveDataInit src (stateName s) ms,
  C F);

/* abstracted version without constraints on data */
(:stateName,:transName,:msg)
Receive
    (s:(stateName,transName)sc_struct)
    (src:stateName)
    (ms:msg)
    (x:transName)
    (n:path) :=
 (ReceiveDataOccurred src (stateName s) ms,
  (\(x:BOOL).(:transName)NoConstraintcfcf'),
  (:stateName,:transName,:msg)
         ReceiveDataOccurs src (stateName s) ms,
  ReceiveDataInit src (stateName s) ms,
  C F);
```

K.3 CommAction

```
/*
  CommActions: Statechart communication actions
  Depends on statechart notational style and
    statechart communication primitives having been loaded.
  Keywords: SendData, Send
*/

(:stateName,:msg)
MsgName (src:stateName) (dest:stateName) (ms:msg) :=
    NAME (Msg src dest ms);

(:stateName,:msg)
DataName (src:stateName) (dest:stateName) (ms:msg) :=
    NAME (Data src dest ms);

(:stateName,:msg)
SendMsgChange (src:stateName) (dest:stateName) (ms:msg)
    (step:step) :=
    (Msg src dest ms (Next step)) EQ T;

(:stateName,:msg)
SendMsgNoChange (src:stateName) (dest:stateName) (ms:msg)
    (step:step) :=
    (Msg src dest ms (Next step)) EQ  F;
```

```
(:stateName,:msg,:A)
SendSetData (src:stateName) (dest:stateName) (ms:msg)
  (data:config->A) (step:step) :=
    (Data src dest ms (Next step)) EQ (data (Prev step));

(:stateName,:transName,:msg,:ty)
SendData
    (s:(stateName,transName)sc_struct)
    (dest:stateName)
    (ms:msg)
    (data:(ty)exp)  :=
    [(
      MsgName (stateName s) dest ms,
      (:stateName,:msg)SendMsgChange (stateName s) dest ms,
      SendMsgNoChange (stateName s) dest ms);
     (
      DataName (stateName s) dest ms,
      SendSetData (stateName s) dest ms data,
      (\step.T))];

/* abstracted version without constraints on data */
(:stateName,:transName,:msg)
Send
    (s:(stateName,transName)sc_struct)
    (dest:stateName)
    (ms:msg) :=
    [(
      MsgName (stateName s) dest ms,
      (:stateName,:msg)SendMsgChange (stateName s) dest ms,
      (:stateName,:msg)SendMsgNoChange (stateName s) dest ms)];
```

Appendix L

Simulation Runs of the Heating System

```
Fusion+ - Version 1.0 Jun 13 1998 14:44:27
Copyright University of British Columbia, 1996, 1997

Type "%include <filename>" or type in S+ paragraphs directly
Type "%help" to see list of % commands.
search path: .


>/*
 Heating system specification
 simulation
 Last modified: 28 Apr 98
 Nancy A. Day
*/;
;
>%include rev_heating.s+
Including ./rev_heating.s+
search path: . ..
Including ../all.s+
Including ../basics.s+
Closing basics.s+
Including ../framework.s+
Closing framework.s+
Including ../bvfcns.s+
Closing bvfcns.s+
Including ../table.s+
Closing table.s+
Including ../events.s+
Closing events.s+
Including ../actions.s+
Closing actions.s+
Including ../sc.s+
Including ../scsyn.s+
Closing scsyn.s+
Including ../scsem.s+
Closing scsem.s+
Closing sc.s+
Including ../sc_ext_exp.s+
Closing sc_ext_exp.s+
Including ../sc_ext_ev.s+
Closing sc_ext_ev.s+
Including ../sc_comm_prim.s+
Closing sc_comm_prim.s+
Including ../sc_comm_action.s+
Closing sc_comm_action.s+
Including ../sc_comm_event.s+
Closing sc_comm_event.s+
Closing all.s+
Closing rev_heating.s+
>
/* analysis parameters:
 (as found in build.s+ which build the bdd loaded from the file
  below)

 - leave keepOldNodes ON
 - evaluate to rewrite level (for event label etc)
 - do condreduction for next vp table
 - no need to do sfe Boolean simplification
 - fusionMode not necessary
 - dT left uninterpreted

*/;
;
>%set sfeEvaluationLevel 3
sfeEvaluationLevel = 3
>%set condReduction ON
condReduction = 1
>%set doSfeBoolSimp OFF
doSfeBoolSimp = 0
```

```
>%set filterOutput 1
filterOutput = 1
>cf,cf':config;
cf, cf':config;
>%include heating_order.s+
Including ./heating_order.s+
Closing heating_order.s+
>%setorder heating_order
Setting up substitution order
.
Finished setting up substitution order
>%load_bdd nsr heating_nsr_bddfile
Bdd of nsr, size 2785 successfully loaded.
>/* ----------------------------------------------------- */

ic := InitialCond (heatingSystemScStruct) cf;
ic := ((InitialCond heatingSystemScStruct) cf);
>sim1 := [
  ic;
  heatSwitchOn cf;
  T
];
sim1 := [ic;(heatSwitchOn cf);T];
>/*
 observe that we need an environmental constraint here
 to connect "tooCold" and "tooHot" to differences in
 temperature
*/;
;
>%simulate nsr sim1

nsr BDD exists, size: 2785

Evaluating condition on first configuration
.....
Converting condition on first configuration to BDD
++

Configuration 0:
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
```

```
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T

calculating next config took: 0 sec bdd size: 48
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(heatSwitchOn cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T

calculating next config took: 0 sec bdd size: 48
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 2:
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
>/* ---------------------------------------------------- */

env :=
(forall i.
let delta := dT i - aT i in
(tooCold i =
    ( ((C 2 < delta) AND (delta <= C 5)) OR
      (C 5 < delta))) AND
(tooHot i =
    ( (delta < C -5) OR
      ((C -5 <= delta) AND (delta < C -2))))) cf;
env :=
  (
    (\cf.
      (FORALL
        (\i.
          (
            (LET
              (
                (\delta.
                  (
                    ((tooCold i) =
                      ((((C 2) < delta) AND (delta <= (C 5))) OR
                        ((C 5) < delta))) AND
                    ((tooHot i) =
                      ((delta < (C -5)) OR
                        (((C -5) <= delta) AND (delta < (C -2))))))
                  ) ((dT i) - (aT i)))) cf)))) cf);
>/* observe that now tooHot or tooCold appears in ouput */;
;
>%simulate nsr sim1 env

nsr BDD exists, size: 2785


Evaluating expression nsr
.
Boolean cond rewriting: 0; cond rewriting: 0
sfe nsr takes user: 0 sec ; system; 0 sec ; expr size: 304
```

```
Collapsing expression nsr
collapse nsr takes: 0 sec ; expr size: 143 ; shared: 100
Converting nsr to BDD
convert nsr takes: 0 sec; bdd size: 25 #bddvars: 16

Evaluating condition on first configuration
Converting condition on first configuration to BDD

Configuration 0:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(heatSwitchOn cf) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = (C 0)) cf) EQ T
```

```
((((TimeEventLastOccurred (T10 KITCHEN)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 2:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
>/* ----------------------------------------------------- */

/*
 typical simulation that takes the system from a room
 being too cold to the heater running
*/;
;
>/*
 with simulate_one_ahead can just go through states and
 desired output
*/;
;
>sim2 := [
  ic /\
    heatSwitchOn cf /\
    (valvePos KITCHEN = C CLOSED) cf;
  InBasicState (WAIT_FOR_HEAT KITCHEN) cf /\
    (valvePos KITCHEN = C HALF) cf;
  InBasicState (WAIT_FOR_HEAT KITCHEN) cf /\
    (valvePos KITCHEN = C OPEN) cf;
  requestHeat KITCHEN cf;
  T;                /* activate should be true */
  T;
  InBasicState (FURNACE_RUNNING) cf;
  T                /* in heater running */
  ];
sim2 :=
  [
    (ic /\
      ((heatSwitchOn cf) /\ (((valvePos KITCHEN) = (C CLOSED)) cf))
      );
    (((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) /\
      (((valvePos KITCHEN) = (C HALF)) cf));
    (((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) /\
      (((valvePos KITCHEN) = (C OPEN)) cf));
    ((requestHeat KITCHEN) cf);T;T;
    ((InBasicState FURNACE_RUNNING) cf);T];
>
%simulate_one_ahead nsr sim2 env

nsr BDD exists, size: 2785


env BDD exists, size: 25

Evaluating and converting all conditions on configurations

calculating "is next config satisfying" took: 0 sec; bdd size: 42

Configuration 0:
(
  (((C 2) < ((dT KITCHEN) - (aT KITCHEN))) AND
    (((dT KITCHEN) - (aT KITCHEN)) <= (C 5))) cf) EQ T
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
(heatSwitchOn cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T
```

```
calculating next config took: 0 sec bdd size: 46

calculating "is next config satisfying" took: 0 sec; bdd size: 49

Configuration 1:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ HALF) EQ T
((tooCold KITCHEN) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = warmUpTime) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
calculating "is next config satisfying" took: 0 sec; bdd size: 46

Configuration 2:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = warmUpTime) cf) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
```

```
calculating next config took: 0 sec bdd size: 45

calculating "is next config satisfying" took: 0 sec; bdd size: 45

Configuration 3:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45

calculating "is next config satisfying" took: 0 sec; bdd size: 45

Configuration 4:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
(activate cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
```

```
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45

calculating "is next config satisfying" took: 0 sec; bdd size: 47

Configuration 5:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_ACTIVATING) cf) EQ T
((((TimeEventLastOccurred T3) []) = furnaceStartupTime) cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
((((TimeEventLastOccurred T3) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
```

```
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45

calculating "is next config satisfying" took: 0 sec; bdd size: 45

Configuration 6:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_RUNNING) cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
(furnaceRunning cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
```

```
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45

Configuration 7:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_RUNNING) cf) EQ T
((InBasicState HEATER_RUNNING) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
>/* ---------------------------------------------------- */

/*
 to use just simulate we have to force triggers to be
 true in precedings states (but same result is accomplished in
 same number of steps)

 Also note that even though the trigger of t10 and t12 are the same
 they are different here based on transition label
*/;
;
>sim3 := [
  ic /\
    heatSwitchOn cf /\
    (valvePos KITCHEN = C CLOSED) cf /\
    ((C 2 < dT KITCHEN - aT KITCHEN) AND
                 (dT KITCHEN - aT KITCHEN <= C 5)) cf;
  InBasicState (WAIT_FOR_HEAT KITCHEN) cf /\
    (TimeEventLastOccurred (T10 KITCHEN) [] = warmUpTime) cf /\
    tooCold KITCHEN cf;
  InBasicState (WAIT_FOR_HEAT KITCHEN) cf /\
    (TimeEventLastOccurred (T12 KITCHEN) [S] = warmUpTime) cf;
  T;               /* requestHeat should be true */
  T                /* activate should be true */;
  (TimeEventLastOccurred T3 [] = furnaceStartupTime) cf;
  T;               /* in heater running */
  T                /* in furnace running */
  ];
sim3 :=
  [
    (ic /\
      ((heatSwitchOn cf) /\
        ((((valvePos KITCHEN) = (C CLOSED)) cf) /\
          (
            (((C 2) < ((dT KITCHEN) - (aT KITCHEN))) AND
              (((dT KITCHEN) - (aT KITCHEN)) <= (C 5))) cf))));
    (((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) /\
      (
        ((((TimeEventLastOccurred (T10 KITCHEN)) []) = warmUpTime) cf)
          /\ ((tooCold KITCHEN) cf)));
    (((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) /\
      ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = warmUpTime) cf
        ));T;T;
    ((((TimeEventLastOccurred T3) []) = furnaceStartupTime) cf);T;T
    ];
>%simulate nsr sim3 env

nsr BDD exists, size: 2785


nsr BDD exists, size: 25

Evaluating condition on first configuration
Converting condition on first configuration to BDD
.

Configuration 0:
(
  (((C 2) < ((dT KITCHEN) - (aT KITCHEN))) AND
    (((dT KITCHEN) - (aT KITCHEN)) <= (C 5))) cf) EQ T
(((valvePos KITCHEN) cf) EQ CLOSED) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_NO_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (IDLE_NO_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState OFF) cf) EQ T
```

```
(heatSwitchOn cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ CLOSED) EQ T
(((valvePos BEDROOM) cf) EQ CLOSED) EQ T

calculating next config took: 0 sec bdd size: 46
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ HALF) EQ T
((tooCold KITCHEN) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = warmUpTime) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
```

```
calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 2:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = warmUpTime) cf) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (WAIT_FOR_HEAT KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
```

```
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 3:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
((InBasicState IDLE) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
```

```
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 4:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_OFF) cf) EQ T
(activate cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 5:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_ACTIVATING) cf) EQ T
((((TimeEventLastOccurred T3) []) = furnaceStartupTime) cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
((((TimeEventLastOccurred T3) []) = (C 0)) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
```

```
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 6:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_RUNNING) cf) EQ T
((InBasicState ACTIVATING_HEATER) cf) EQ T
(furnaceRunning cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
```

```
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T

calculating next config took: 0 sec bdd size: 45
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 7:
(((C 5) < ((dT KITCHEN) - (aT KITCHEN))) cf) EQ T
((requestHeat KITCHEN) cf) EQ T
(((valvePos KITCHEN) cf) EQ OPEN) EQ T
((tooCold KITCHEN) cf) EQ T
((InBasicState (IDLE_HEATING KITCHEN)) cf) EQ T
(((C 5) < ((dT LIVING_ROOM) - (aT LIVING_ROOM))) cf) EQ T
(((valvePos LIVING_ROOM) cf) EQ OPEN) EQ T
((tooCold LIVING_ROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT LIVING_ROOM)) cf) EQ T
(((C 5) < ((dT BEDROOM) - (aT BEDROOM))) cf) EQ T
(((valvePos BEDROOM) cf) EQ OPEN) EQ T
((tooCold BEDROOM) cf) EQ T
((InBasicState (WAIT_FOR_HEAT BEDROOM)) cf) EQ T
((InBasicState FURNACE_RUNNING) cf) EQ T
((InBasicState HEATER_RUNNING) cf) EQ T
((((TimeEventLastOccurred (T11 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 KITCHEN)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 KITCHEN)) [S]) + (C 1)) p_cf)) EQ T
```

```
((((TimeEventLastOccurred (T10 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 KITCHEN)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 KITCHEN)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 LIVING_ROOM)) [S]) + (C 1)) p_cf)
  ) EQ T
((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 LIVING_ROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T11 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T11 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T12 BEDROOM)) [S]) cf) EQ
  ((((TimeEventLastOccurred (T12 BEDROOM)) [S]) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T10 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T10 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred (T13 BEDROOM)) []) cf) EQ
  ((((TimeEventLastOccurred (T13 BEDROOM)) []) + (C 1)) p_cf)) EQ T
((((TimeEventLastOccurred T3) []) cf) EQ
  ((((TimeEventLastOccurred T3) []) + (C 1)) p_cf)) EQ T
>
Cleaning up node use.
Cleaning up bdd use.
Fusion session over.
```

Appendix M

Specification of the Separation Minima for the North Atlantic Region

The complete formal specification of the separation minima for the North Atlantic Region written by the author, J. Joyce and G. Pelletier can be found in Day, Joyce and Pelletier [DJP97b]. The document is formatted in HTML and contains the formal specification as well as explanatory text. The fragments of the formal specification are automatically extracted from the HTML document and ordered based on their dependencies (so items are declared or defined before they are used) for input to Fusion. Here we present only the formal elements of the specification. Certain information included here, such as line numbers, is automatically generated for cross-referencing purposes with the HTML document. This example is written in S+ together with the TableExpr notational style.

```
%sourcefile SeparationMinimaSpec.hpp

%linenum 723
```

```
(:ty) (_ In 10 _) : (ty) -> ((ty)set) ->bool;

%include ../basics.s+
%include ../framework.s+
%include ../table.s+

%linenum 570
:FLIGHT;
:flight == config -> FLIGHT;

%linenum 572
:location := Azores | BDA | CAN | Caribbean | IberianPeninsula
            | Iceland | Scandinavia | UnitedKingdom | USA ;

%linenum 576
:SEGMENT;
:segment == config -> SEGMENT;
%linenum 577
:time == num;
:timeperiod == config -> (NUM # NUM);
%linenum 713
Min: ((num)set -> num);
%linenum 115

Routes1 := {(USA,BDA);(CAN,BDA);(IberianPeninsula, Azores);
(Iceland,Scandinavia);(Iceland, UnitedKingdom)};

%linenum 119

Routes2 := {(USA,Caribbean);(CAN,Caribbean);(BDA, Caribbean)};

%linenum 413
"WATRSSameDir LongSep" : (flight#flight) -> num;

%linenum 417
"WATRSOppDir NoLongSepPeriod": (flight#flight) -> timeperiod;

%linenum 467

Routes3 :=
{(USA,Caribbean);(CAN,Caribbean);
(BDA, Caribbean);(USA,BDA); (CAN,BDA)};

%linenum 587
IsSupersonic :(flight -> bool);
```

```
%linenum 588
IsTurbojet :(flight -> bool);
%linenum 589
HavePartOfRouteInMNPSAirspace : (flight -> bool) ;
%linenum 591
MeetMNPS :(flight -> bool);
%linenum 592
OnPublishedRoute :(flight -> bool);
%linenum 593
RouteDeparture :(flight -> location);
%linenum 594
RouteDestination :(flight -> location);
%linenum 596
MachTechniqueUsed : flight ->bool;
%linenum 603
FlightLevel:(flight -> num);
%linenum 604
InCruiseClimb:(flight -> bool);
%linenum 605
InWATRSAirspace :(flight -> bool);
%linenum 606
IsLevel:(flight -> bool);
%linenum 607
IsOutsideMNPSAirspace :(flight -> bool);
%linenum 609
IsWestOf60W :(flight -> bool);
%linenum 610
IsWestOf55W:(flight -> bool);
%linenum 611
"LatChange Per10DLong LessThanOrEq1" :flight->bool;
%linenum 613
"LatChange Per10DLong LessThanOrEq2" :flight->bool;
%linenum 615
"LatChange Per10DLong LessThanOrEq3" :flight->bool;
%linenum 617
LateralPositionInDegrees :(flight -> num);
%linenum 619
LateralPositionInMiles :(flight -> num);
%linenum 621
Mach : (flight->num);
%linenum 622
RouteSegment :(flight -> segment);
%linenum 623
"RouteSegment Degrees" :(flight -> num);
%linenum 624
```

```
TimeAtPosition :(flight->time);
%linenum 629
SameType :((flight # flight) -> bool);
%linenum 635
SameMachNumber :((flight # flight) -> bool);
%linenum 637
FirstAircraft :((flight # flight) -> flight);
%linenum 639
ept : ((flight # flight)->time);
%linenum 640
"SameOr Diverging Tracks"  :((flight # flight) -> bool);
%linenum 642
SecondAircraft :((flight # flight) -> flight);
%linenum 647
ReportedOverCommonPoint :((flight # flight) -> bool);
%linenum 649
"Appropriate TimeSep AtCommon Point":((flight # flight) -> bool);
%linenum 651
EnterWATRSAirspaceAtSomeTime : (flight -> bool);
%linenum 657
AngularDifferenceGreaterThan90Degrees:(segment # segment)->bool;
%linenum 668
StartTime : (timeperiod) -> time;
%linenum 672
EndTime : (timeperiod) -> time;
%linenum 677
MinEarliestTime : (timeperiod)set -> time;
%linenum 682
MaxLatestTime : (timeperiod)set -> time;
%linenum 97

VerticalSeparationRequired (A,B) := Table
[Row (FlightLevel (A)) [(\x.x <= (C 280)); Dc ;
                        (\x.x>(C 450));(\x.x>(C 450))];
 Row (FlightLevel (B)) [Dc;(\x.x <= (C 280));
                        (\x. x > (C 450));(\x.x>(C 450))];
 Row (IsSupersonic (A)) [Dc;Dc;True;Dc];
 Row (IsSupersonic (B)) [Dc;Dc;Dc;True] ]
[C 1000;C 1000;C 4000;C 4000;C 2000];


%linenum 128

IsOnRoute (R:(location#location)set) (X:flight) :=
   ((RouteDeparture (X), RouteDestination (X)) In R) OR
```

```
    ((RouteDestination (X), RouteDeparture (X)) In R);

%linenum 136

FlightLevelAbove275 (X:flight) := FlightLevel X > (C 275);

%linenum 142

"LateralSeparation RequiredInDegrees" (A,B) := Table
[Row (AllOf [A;B] IsOutsideMNPSAirspace) [True;True;Dc;Dc];
 Row (AllOf [A;B] (IsOnRoute (Routes1))) [True;Dc;Dc;Dc];
 Row (AllOf [A;B] (IsOnRoute (Routes2))) [Dc;True;Dc;Dc];
 Row (AllOf [A;B] IsWestOf55W) [Dc;True;Dc;Dc];
 Row (AllOf [A;B] IsSupersonic) [Dc;Dc;True;Dc];
 Row (AllOf [A;B] FlightLevelAbove275) [Dc;Dc;True;Dc];
 Row (AllOf [A;B] MeetMNPS) [Dc;Dc;Dc;True];
 Row (AllOf [A;B] HavePartOfRouteInMNPSAirspace) [Dc;Dc;Dc;True]]
[C 1.5;C 1.5;C 1;C 1;C 2];

%linenum 160

"LateralSeparation RequiredInMiles" (A,B) := Table
[Row (AllOf [A;B] IsOutsideMNPSAirspace) [True;True;Dc;Dc];
 Row (AllOf [A;B] (IsOnRoute (Routes1))) [True;Dc;Dc;Dc];
 Row (AllOf [A;B] (IsOnRoute (Routes2))) [Dc;True;Dc;Dc];
 Row (AllOf [A;B] IsWestOf55W) [Dc;True;Dc;Dc];
 Row (AllOf [A;B] IsSupersonic) [Dc;Dc;True;Dc];
 Row (AllOf [A;B] FlightLevelAbove275) [Dc;Dc;True;Dc];
 Row (AllOf [A;B] MeetMNPS ) [Dc;Dc;Dc;True];
 Row (AllOf [A;B] HavePartOfRouteInMNPSAirspace) [Dc;Dc;Dc;True]]
[C 90;C 90;C 60;C 60;C 120];

%linenum 190

LatitudeEquivalent (A,B) := PredicateTable
[Row ("RouteSegment Degrees" A)
  [(\x.x<=C 58);Dc;(\x.((C 58)<x) AND (x< C 70));Dc;
   (\x.(C 70<=x) AND (x<=C 80));Dc];
 Row ("RouteSegment Degrees" B)
  [Dc;(\x.x<=C 58);Dc;(\x.(x>C 58) AND (x< C 70));Dc;
   (\x.(C 70<=x) AND (x<=C 80))];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq3")
  [True;True;Dc;Dc;Dc;Dc];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq2")
  [Dc;Dc;True;True;Dc;Dc];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq1")
  [Dc;Dc;Dc;Dc;True;True]];

% added 24 Apr 98 NAD for demonstration purposes
LatitudeEquivalent2 (A,B) := PredicateTable
[Row ("RouteSegment Degrees" A)
  [(\x.x<=C 58);Dc;(\x.((C 58)<x) AND (x< C 70));Dc;
   (\x.(C 70<=x) AND (x<=C 80));Dc];
 Row ("RouteSegment Degrees" B)
  [Dc;(\x.x<=C 58);Dc;(\x.(C 58<x) AND (x< C 70));Dc;
   (\x.(C 70<=x) AND (x<=C 80))];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq3")
  [True;True;Dc;Dc;Dc;Dc];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq2")
  [Dc;Dc;True;True;Dc;Dc];
 Row (AllOf [A;B] "LatChange Per10DLong LessThanOrEq1")
  [Dc;Dc;Dc;Dc;True;True]];

%linenum 243

"ssOppDir NoLongSepPeriod" (A,B) := Table
[Row (ReportedOverCommonPoint(A,B)) [True;False]]
[P (ept(A,B),ept(A,B)+(C 10));P (ept(A,B)-(C 15),ept(A,B)+(C 15))];

%linenum 266

ssSubcondition(A,B) := PredicateTable
[Row (AllOf [A;B] IsLevel) [True;Dc];
 Row (SameMachNumber (A,B)) [True;Dc];
 Row (SameType(A,B)) [Dc;True];
 Row (AllOf [A;B] InCruiseClimb ) [Dc;True]];

%linenum 336

UnionOfRange (periods) :=
P (MinEarliestTime (periods), MaxLatestTime (periods));

%linenum 360

MNPSCondition(A,B) :=
(AllOf [A;B] MeetMNPS ) AND
(AllOf [A;B] HavePartOfRouteInMNPSAirspace );

%linenum 371
```

```
"MNPSOppDir NoLongSepPeriod"(A,B) := "ssOppDir NoLongSepPeriod"(A,B);

%linenum 380

"MNPSSameDir LongSep" (A,B) := Table
[Row ("Appropriate TimeSep AtCommon Point" (A,B))
[True;True;True;True;True];
 Row ("SameOr Diverging Tracks" (A,B)) [True;True;True;True;True];
 Row (Mach (FirstAircraft (A,B)) - Mach (SecondAircraft (A,B)))
[(\x. (x> C 0.06));
(\x. ((C 0.06>=x) AND (x>C 0.05)));
(\x. (((C 0.05)>=x) AND (x>C 0.04)));
(\x. (((C 0.04)>=x) AND (x>(C 0.03))));
(\x. (((C 0.03)>=x) AND (x>C 0.02)))]]
[C 5;C 6;C 7;C 8;C 9;C 10];

%linenum 395

WATRSCondition(A,B) := PredicateTable
[Row (AllOf [A;B] EnterWATRSAirspaceAtSomeTime ) [True;True];
 Row (AllOf [A;B] IsWestOf60W ) [True;Dc];
 Row (AllOf [A;B] InWATRSAirspace) [Dc;True];
 Row (AllOf [A;B] MachTechniqueUsed ) [True;True];
 Row (AllOf [A;B] OnPublishedRoute) [True;True];
 Row ("SameOr Diverging Tracks" (A,B)) [True;True]];

%linenum 432

"genOppDir NoLongSep Period"(A,B) := "MNPSOppDir NoLongSepPeriod"(A,B);

%linenum 440

"genSameDir LongSep" (A,B) := Table
[Row ("SameOr Diverging Tracks" (A,B)) [True;True;True];
 Row (AllOf [A;B] MachTechniqueUsed ) [False;True;True];
 Row (AtLeastOneOf [A;B] InCruiseClimb ) [False;False;False];
 Row (ReportedOverCommonPoint (A,B) ) [True;Dc;Dc];
 Row ("Appropriate TimeSep AtCommon Point" (A,B)) [Dc;True;True];
 Row (Mach (FirstAircraft(A,B)) - Mach (SecondAircraft(A,B)))
      [Dc;(\x. (x> C 0.6));(\x. (C 0.6>=x) AND (x>C 0.3))]]
[C 15;C 5;C 10;C 20];

%linenum 459

"otherOppDir NoLongSepPeriod" (A,B) :=
   "genOppDir NoLongSep Period"(A,B);

%linenum 474

otherSameDirLongSep (A,B) := Table
[Row (ReportedOverCommonPoint(A,B)) [True;Dc];
 Row ("SameOr Diverging Tracks"(A,B)) [True;Dc];
 Row (AllOf [A;B] (IsOnRoute Routes3)) [Dc;True]]
[C 15;C 20;C 30];

%linenum 492

env1 :=
(forall (A:flight).  NOT (IsLevel (A) AND InCruiseClimb (A)))
AND
(forall (A:flight).
   NOT (IsOnRoute (Routes1) (A) AND IsOnRoute (Routes2) (A)));

%linenum 501

env2 :=
(forall (A:flight) (B:flight).
ReportedOverCommonPoint(A,B) = ReportedOverCommonPoint(B,A))
AND
(forall (A:flight) (B:flight).
SameMachNumber(A,B) = SameMachNumber(B,A))
AND
(forall (A:flight) (B:flight).
SameType(A,B) = SameType(B,A))
AND
(forall (A:flight) (B:flight).
"SameOr Diverging Tracks"(A,B) = "SameOr Diverging Tracks"(B,A))
AND
(forall (A:flight) (B:flight).
"Appropriate TimeSep AtCommon Point"(A,B) =
"Appropriate TimeSep AtCommon Point"(B,A)) ;

%linenum 523

env3 :=
(forall A.
if "LatChange Per10DLong LessThanOrEq2" (A)
then "LatChange Per10DLong LessThanOrEq3" (A))
AND
(forall A.
if "LatChange Per10DLong LessThanOrEq1" (A)
then "LatChange Per10DLong LessThanOrEq2" (A))
AND
(forall A.
if "LatChange Per10DLong LessThanOrEq1" (A)
then "LatChange Per10DLong LessThanOrEq3" (A));

%linenum 540

env := env1 AND env2 AND env3;

%linenum 254

ssSameDirLongSep(A,B) := Table
[Row (ssSubcondition(A,B)) [True;True];
 Row ("SameOr Diverging Tracks"(A,B)) [True;True];
 Row (ReportedOverCommonPoint(A,B)) [True;Dc];
 Row ("Appropriate TimeSep AtCommon Point"(A,B)) [Dc;True]]
[C 10;C 10;C 15];

%linenum 306

MinAll(A,B) :=
    Min {
         "MNPSSameDir LongSep"(A,B);
         "WATRSSameDir LongSep"(A,B);
         "genSameDir LongSep"(A,B)};

%linenum 341

UnionAll (A,B) :=
    let periods :=
             {"MNPSOppDir NoLongSepPeriod"(A,B);
              "WATRSOppDir NoLongSepPeriod"(A,B);
              "genOppDir NoLongSep Period"(A,B)} in
    P (MinEarliestTime (periods), MaxLatestTime (periods));

%linenum 291

"turbojetSameDir LongSep" (A,B) := Table
[Row (MNPSCondition (A,B)) [True;False;True;False];
 Row (WATRSCondition (A,B)) [True;True;False;False]]
[MinAll (A,B);
 Min { "WATRSSameDir LongSep" (A,B);
     "genSameDir LongSep" (A,B)};
 Min { "MNPSSameDir LongSep" (A,B);
     "genSameDir LongSep" (A,B)};
 "genSameDir LongSep" (A,B)];

%linenum 318

"turbojetOppDir NoLongSepPeriod" (A,B) := Table
[Row (MNPSCondition (A,B)) [True;False;True;False];
 Row (WATRSCondition (A,B)) [True;True;False;False]]
[UnionAll (A,B);
 UnionOfRange { "WATRSOppDir NoLongSepPeriod" (A,B);
          "genOppDir NoLongSep Period" (A,B)};
 UnionOfRange { "MNPSOppDir NoLongSepPeriod" (A,B);
         "genOppDir NoLongSep Period" (A,B)};
 "genOppDir NoLongSep Period" (A,B)];

%linenum 219

LongSameDirSepRequired (A,B) := Table
[Row (AllOf [A;B] IsSupersonic ) [True;False];
 Row (AllOf [A;B] IsTurbojet ) [Dc;True]]
[ssSameDirLongSep (A,B);"turbojetSameDir LongSep" (A,B);
 otherSameDirLongSep (A,B)];

%linenum 226

"OppDir NoLongSepPeriod" (A,B) := Table
[Row (AllOf [A;B] IsSupersonic ) [True;False];
 Row (AllOf [A;B] IsTurbojet) [Dc;True]]
["ssOppDir NoLongSepPeriod" (A,B);
 "turbojetOppDir NoLongSepPeriod" (A,B);
 "otherOppDir NoLongSepPeriod" (A,B)];

%linenum 212

WithinOppDirNoLongSepPeriod(A:flight,B:flight,t:time) :=
  let timePeriod := "OppDir NoLongSepPeriod"(A,B) in
  (StartTime(timePeriod) <= t) AND (t <= EndTime(timePeriod));

%linenum 60

AreSeparated(A:flight,B:flight,t:time) :=
    /* A and B are vertically separated based on flight level */
    (ABS(FlightLevel A - FlightLevel B) >
                VerticalSeparationRequired(A,B))
```

```
OR

/* A and B are laterally separated based on either position
   in degrees of latitude or position in miles */
(if (LatitudeEquivalent(A,B))
then
  (ABS(LateralPositionInDegrees A - LateralPositionInDegrees B) >
       "LateralSeparation RequiredInDegrees" (A,B))
else
  (ABS(LateralPositionInMiles A - LateralPositionInMiles B) >
       "LateralSeparation RequiredInMiles" (A,B)))
OR

/* A and B are longitudinally separated based on time
   depending on whether the two flights are in the approximate
   same or opposite direction */
(if (AngularDifferenceGreaterThan90Degrees
        (RouteSegment A, RouteSegment B))
then     /* opposite direction */
    NOT (WithinOppDirNoLongSepPeriod(A,B,t))
else     /* same direction */
    ABS(TimeAtPosition A - TimeAtPosition B) >
           LongSameDirSepRequired(A,B));
```

Appendix N

Specification of the Association Control Service Element

This appendix includes the formal specification of the Association Control Service Element (ACSE) written by Jamie Andrews as part of a group effort to specify formally the Aeronautical Telecommunications Network (ATN). This specification is written in S+ together with the notational styles: CoreSc, CoreEvent, CoreAction, CommEvent, and CommAction. The first section is the file of common declarations and definitions used by all components of the ATN. The second section is the ACSE formal specification. The author of this specification used the form of comments in S+ consisting of a `%` at the beginning of a line following by a blank space.

N.1 Common declarations and definitions in the ATN

```
% File name:   atncommon.s
% Author:      Jamie Andrews
% Date:        1996 Dec. 2
% Description: Header file containing common declarations for all
%              ATN components being modelled by statecharts project

%addpath ../..
%include all.s+

% state names
%   - components are parameterised by numbers
%       e.g. (CM i) is the ASE for AE #i
%   - basic state names within components are parameterised by
%     numbers or state names as per author's choice
:stateName :=
  System |

  Ground_ADS :NUM |

  CM_Ground :NUM |
    CM_Ground_IDLE :NUM |
    CM_Ground_LOGON :NUM |
    CM_Ground_UPDATE :NUM |
    CM_Ground_CONTACT :NUM |
    CM_Ground_DIALOGUE :NUM |
    CM_Ground_CONTACT_DIALOGUE :NUM |
    CM_Ground_END :NUM |
    CM_Ground_FORWARD :NUM |
  CM_Air :NUM |
    CM_Air_IDLE :NUM |
    CM_Air_LOGON :NUM |
    CM_Air_CONTACT :NUM |
    CM_Air_DIALOGUE :NUM |
    CM_Air_CONTACT_DIALOGUE :NUM |
  CPDLC :NUM |

% Components of every AE; e.g. (CF i) is the CF for AE #1
  CF :NUM |
    CF_STA0 :NUM |
    CF_STA1 :NUM |
    CF_STA2 :NUM |
    CF_STA3 :NUM |
    CF_STA4 :NUM |

  ACSE :NUM |
    ACSE_Idle :stateName |
    ACSE_Awaiting_AARE :stateName |
    ACSE_Awaiting_AASCrsp :stateName |
    ACSE_Awaiting_RLRE :stateName |
    ACSE_Awaiting_ARLSrsp :stateName |
    ACSE_Associated :stateName |
    ACSE_Collision_initiator :stateName |
    ACSE_Collision_responder :stateName |

% (SuppSvc i j) is the supporting service connecting AE #i with AE #j
  SuppSvc :NUM :NUM |
    SuppSvc_Translate :stateName |


% The error state, to which errors should be reported
  Error  |

% applications - interact with CF
  AppUser :NUM;

% Possible kinds of error messages
:msg :=
  NotPermitted |
  CannotOccur |
  BlankCell |

% protocol messages

  /* Going between the ASE and the CF */

  D_START_req |
  D_START_rsp_pos |
  D_START_rsp_neg |
  D_DATA_req |
  D_END_req |
  D_END_rsp_pos |
  D_END_rsp_neg |
  D_ABORT_req |

  D_START_ind |
  D_START_cnf_pos |
  D_START_cnf_neg |
  D_DATA_ind |
  D_END_ind |
  D_END_cnf_pos |
  D_END_cnf_neg |
  D_ABORT_ind |
  D_P_ABORT_ind |
/* Going between the CF and the ACSE */

A_ASSOCIATE_req |
A_ASSOCIATE_rsp_pos |
A_ASSOCIATE_rsp_neg |
A_RELEASE_req |
A_RELEASE_rsp_pos |
A_RELEASE_rsp_neg |
A_ABORT_req |

A_ASSOCIATE_ind |
A_ASSOCIATE_cnf_pos |
A_ASSOCIATE_cnf_neg |
A_RELEASE_ind |
A_RELEASE_cnf_pos |
A_RELEASE_cnf_neg |
A_ABORT_ind |
A_P_ABORT_ind |

/* Going between the ASE or ACSE and the supporting service
   via the CF */

Prev_D_END_cnf :NUM |

P_CONNECT_req |
P_CONNECT_rsp_pos |
P_CONNECT_rsp_neg |
P_RELEASE_req |
P_RELEASE_rsp_pos |
P_RELEASE_rsp_neg |
P_U_ABORT_req |

P_DATA_req |
P_DATA_ind |

P_CONNECT_ind |
P_CONNECT_cnf_pos |
P_CONNECT_cnf_neg |
P_RELEASE_ind |
P_RELEASE_cnf_pos |
P_RELEASE_cnf_neg |
P_U_ABORT_ind |
P_P_ABORT_ind |

/* Going between CM_Air, CM_Ground and AppUser */
```

```
  CM_LOGON_req |
  CM_LOGON_ind |
  CM_LOGON_rsp_pos |
  CM_LOGON_rsp_neg |
  CM_LOGON_cnf_pos |
  CM_LOGON_cnf_neg |

  CM_UPDATE_req |
  CM_UPDATE_ind |
  CM_UPDATE_cnf_neg |

  CM_CONTACT_req |
  CM_CONTACT_ind |
  CM_CONTACT_rsp_pos |
  CM_CONTACT_cnf_pos |
  CM_CONTACT_cnf_neg |

  CM_END_req |
  CM_END_ind |
  CM_END_cnf_neg |

  CM_FORWARD_req |
  CM_FORWARD_ind |
  CM_FORWARD_cnf_pos |
  CM_FORWARD_cnf_neg |

  CM_PROVIDER_ABORT_req |
  CM_PROVIDER_ABORT_ind |

  CM_USER_ABORT_req |
  CM_USER_ABORT_ind |
  CM_USER_ABORT_cnf_pos |
  CM_USER_ABORT_cnf_neg ;

:transName :=
  ACSE_tr :stateName :msg |
  ACSE_tr_A_ASSOCIATE_req_ACSE_Idle :NUM :NUM |
  ACSE_tr_P_CONNECT_ind_ACSE_Idle :NUM :NUM |
  ACSE_tr_P_RELEASE_ind_ACSE_Awaiting_RLRE :NUM :NUM |

  CF_tr :NUM |

  SuppSvc_tr :NUM :NUM :msg |
  CM_Ground_tr :NUM |
  CM_Air_tr :NUM ;

(:ty)
NumberTransAux NIL (tr_constr:NUM->transName) n := NIL |
NumberTransAux (CONS (a:ty) b) (tr_constr) n :=
  COND (~(n EQ 0))
    (CONS (tr_constr n,a) (NumberTransAux b tr_constr (n PLUS 1))) NIL;

(:ty)
NumberTrans (a:(ty)list) (tr_constr:NUM->transName) :=
  NumberTransAux a tr_constr 1;

% added to ensure all uses of Send send the same message type
% use conservative Send, which only places constraints on
% messages (not data)

(:A)ATNSend s dst ms (data:(A)exp):=
  (:stateName,:transName,:msg)Send s dst ms;

(:A)ATNReceive s src ms (data:(A)exp) :=
  (:stateName,:transName,:msg)Receive s src ms;
```

N.2 Association Control Service Element

N.2.1 ACSE data declarations

```
% File name: acse_data.s+
% Author: Jamie Andrews

% A variable for recording whether this ACSE was the one
% initiating the association.
Initiating_ACSE:
   NUM -> bool;

% A variable for recording whether the version of the protocol
% requested on a P_CONNECT_ind is supported by this ACSE.
ACSEVersionSupported:
   NUM -> bool;

% The type of data in messages flowing through the ACSE.
```

```
: DataInMessageToACSE;
: dataInMessageToACSE == config -> DataInMessageToACSE;

% A global for storing the data in a given message.
ACSEData:   NUM -> dataInMessageToACSE;
```

N.2.2 ACSE specification

```
% ACSE: Association Control Service Element

%include acse_data.s+

% Most transition names are given as
% (PTrans <source state> <message name>).
PTrans (s:stateName) (se:msg) :=
    ACSE_tr s se;

% Special transition names for two places where we have
% to do something different depending on the above variables.
ACSE_A_ASSOCIATE_req_ACSE_Idle_transition (n1:NUM) (n2:NUM) :=
    ACSE_tr_A_ASSOCIATE_req_ACSE_Idle n1 n2;
ACSE_P_CONNECT_ind_ACSE_Idle_transition (n1:NUM) (n2:NUM) :=
    ACSE_tr_P_CONNECT_ind_ACSE_Idle n1 n2;
ACSE_P_RELEASE_ind_ACSE_Awaiting_RLRE_transition (n1:NUM) (n2:NUM) :=
    ACSE_tr_P_RELEASE_ind_ACSE_Awaiting_RLRE n1 n2;


% A normal transition for the ACSE (no conditions).
% Normally called as "inMessage.(ACSE_TRANS ...)" in order to
% emphasize message.
ACSE_TRANS (s:(stateName,transName)sc_struct) i sourceState
     (outMessage:msg) (destState: stateName -> stateName)
     (inMessage:msg) :=
  ( (PTrans ((ACSE i).sourceState) inMessage),
    ((ACSE i).sourceState),
    (ATNReceive s (CF i) inMessage (ACSEData i)),
    (ATNSend s (CF i) outMessage (ACSEData i)),
    ((ACSE i).destState)
  );

% The "error" transition for the idle state (before an
% association has been initiated).  Normally called as
% "inMessage.(ACSE_idle_error i)" in order to emphasize message.
ACSE_idle_error (s:(stateName,transName)sc_struct) i inMessage :=
  ( (PTrans ((ACSE i).ACSE_Idle) inMessage),
    (ACSE i).ACSE_Idle,
    ATNReceive s (CF i) inMessage (ACSEData i),
    ATNSend s Error BlankCell (C T),
    (ACSE i).ACSE_Idle
  );

% The usual "error" transition; takes action recommended in ISO
% 8650 section 7.3.3.4. Normally called as "inMessage.(ACSE_error
% i sourceState)" in order to emphasize message.
ACSE_error (s:(stateName,transName)sc_struct) i sourceState inMessage :=
  ( (PTrans ((ACSE i).sourceState) inMessage),
    (ACSE i).sourceState,
    ATNReceive s (CF i) inMessage (ACSEData i),
    Both
      (ATNSend s Error BlankCell (C T))
      (Both (ATNSend s (CF i) A_ABORT_ind (C T))
            (ATNSend s (CF i) P_U_ABORT_req (C T))),
    (ACSE i).ACSE_Idle
  );



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transitions: from ISO 8650, Table 14
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  Transitions_From_ACSE_Idle s i :=
    /* From ACSE_Idle state (STA0) */
    let Error_Cell := (ACSE_idle_error s i) in

    [ /* Making connection */

      /* For A_ASSOCIATE_req: must set P2 variable to T            */
      /*   ("this ACSE initiated the association").                */
      /* Also, must test whether version is supported.             */
      /* Written as message . function to make it the same         */
      /*   as the others.  */
      A_ASSOCIATE_req     .
        (function inMessage .
          ( (ACSE_A_ASSOCIATE_req_ACSE_Idle_transition i 1),
            (ACSE i).ACSE_Idle,
        EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
               ((ACSEVersionSupported i) = (C T)),
        Both (ATNSend s (CF i) P_CONNECT_req (ACSEData i))
             (Asn (Initiating_ACSE i) (C T)),
        (ACSE i).ACSE_Awaiting_AARE
      )
    )   ;
  A_ASSOCIATE_req      .
   (function inMessage .
     ( (ACSE_A_ASSOCIATE_req_ACSE_Idle_transition i 2),
        (ACSE i).ACSE_Idle,
        EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
               ((ACSEVersionSupported i) = (C F)),
        ATNSend s Error BlankCell (C T),
        (ACSE i).ACSE_Idle
      )
    )   ;

  A_ASSOCIATE_rsp_pos . Error_Cell;
  A_ASSOCIATE_rsp_neg . Error_Cell ;

  /* For P_CONNECT_ind: must set P2 variable to F       */
  /*   ("this ACSE did not initiate the association").  */
  /* If version supported, pass indication on normally. */
  P_CONNECT_ind        .
    (function inMessage .
      ( (ACSE_P_CONNECT_ind_ACSE_Idle_transition i 1),
        (ACSE i).ACSE_Idle,
        EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
               ((ACSEVersionSupported i) = (C T)),
        Both (ATNSend s (CF i) A_ASSOCIATE_ind (ACSEData i))
             (Asn (Initiating_ACSE i) (C F)),
        (ACSE i).ACSE_Awaiting_AASCrsp
      )
    )  ;
  /* If version not supported, send negative response.  */
  P_CONNECT_ind        .
    (function inMessage .
      ( (ACSE_P_CONNECT_ind_ACSE_Idle_transition i 2),
        (ACSE i).ACSE_Idle,
        EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
              ((ACSEVersionSupported i) = (C F)),
        ATNSend s (CF i) P_CONNECT_rsp_neg (C T),
        (ACSE i).ACSE_Awaiting_AASCrsp
      )
      ) ;

    P_CONNECT_cnf_pos   . Error_Cell;
    P_CONNECT_cnf_neg   . Error_Cell;
    /* Releasing connection normally */
    A_RELEASE_req       . Error_Cell;
    A_RELEASE_rsp_pos   . Error_Cell  ;
    A_RELEASE_rsp_neg   . Error_Cell;
    P_RELEASE_ind       . Error_Cell;
    P_RELEASE_cnf_pos   . Error_Cell;
    P_RELEASE_cnf_neg   . Error_Cell  ;
    /* Releasing connection abnormally */
    A_ABORT_req         . Error_Cell;
    P_U_ABORT_ind       . Error_Cell;
    P_P_ABORT_ind       . Error_Cell
  ];



Transitions_From_ACSE_Awaiting_AARE s i :=
  /* From ACSE_Awaiting_AARE state (STA1) */
  let Error_Cell := (ACSE_error s i ACSE_Awaiting_AARE) in
  let TRANS_CELL := (ACSE_TRANS s i ACSE_Awaiting_AARE) in

  [ /* Making connection */
    A_ASSOCIATE_req     . Error_Cell;
    A_ASSOCIATE_rsp_pos . Error_Cell;
    A_ASSOCIATE_rsp_neg . Error_Cell;
    P_CONNECT_ind       . Error_Cell;
    P_CONNECT_cnf_pos   . (TRANS_CELL A_ASSOCIATE_cnf_pos
                                          ACSE_Associated);
    P_CONNECT_cnf_neg   . (TRANS_CELL A_ASSOCIATE_cnf_neg ACSE_Idle);
    /* Releasing connection normally */
    A_RELEASE_req       . Error_Cell;
    A_RELEASE_rsp_pos   . Error_Cell;
    A_RELEASE_rsp_neg   . Error_Cell;
    P_RELEASE_ind       . Error_Cell;
    P_RELEASE_cnf_pos   . Error_Cell;
    P_RELEASE_cnf_neg   . Error_Cell;
    /* Releasing connection abnormally */
    A_ABORT_req         . (TRANS_CELL P_U_ABORT_req       ACSE_Idle);
    P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind         ACSE_Idle);
    P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind       ACSE_Idle)
  ];
```

```
Transitions_From_ACSE_Awaiting_AASCrsp s i :=
  /* From ACSE_Awaiting_AASCrsp state (STA2) */
  let Error_Cell := (ACSE_error s i ACSE_Awaiting_AASCrsp) in
  let TRANS_CELL := (ACSE_TRANS s i ACSE_Awaiting_AASCrsp) in

  [ /* Making connection */
    A_ASSOCIATE_req     . Error_Cell ;
    A_ASSOCIATE_rsp_pos . (TRANS_CELL P_CONNECT_rsp_pos
                                              ACSE_Associated)  ;
    A_ASSOCIATE_rsp_neg . (TRANS_CELL P_CONNECT_rsp_neg ACSE_Idle)  ;
    P_CONNECT_ind       . Error_Cell   ;
    P_CONNECT_cnf_pos   . Error_Cell ;
    P_CONNECT_cnf_neg   . Error_Cell;
    /* Releasing connection normally */
    A_RELEASE_req       . Error_Cell;
    A_RELEASE_rsp_pos   . Error_Cell;
    A_RELEASE_rsp_neg   . Error_Cell;

    P_RELEASE_ind       . Error_Cell;
    P_RELEASE_cnf_pos   . Error_Cell;
    P_RELEASE_cnf_neg   . Error_Cell;
    /* Releasing connection abnormally */
    A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
    P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
    P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
  ];



Transitions_From_ACSE_Awaiting_RLRE s i :=
  /* From ACSE_Awaiting_RLRE state (STA3) */
  let Error_Cell := (ACSE_error s i ACSE_Awaiting_RLRE) in
  let TRANS_CELL := (ACSE_TRANS s i ACSE_Awaiting_RLRE) in

  [ /* Making connection */
    A_ASSOCIATE_req     . Error_Cell;
    A_ASSOCIATE_rsp_pos . Error_Cell;
    A_ASSOCIATE_rsp_neg . Error_Cell;
    P_CONNECT_ind       . Error_Cell;
    P_CONNECT_cnf_pos   . Error_Cell;
    P_CONNECT_cnf_neg   . Error_Cell;
    /* Releasing connection normally */
    A_RELEASE_req       . Error_Cell;
    A_RELEASE_rsp_pos   . Error_Cell;
    A_RELEASE_rsp_neg   . Error_Cell;

  /* For P_RELEASE_ind: 2 possibilities depending on whether this is */
  /* the ACSE which initiated the connection (the P2 variable).      */
    P_RELEASE_ind       .
      (function inMessage .
        ( (ACSE_P_RELEASE_ind_ACSE_Awaiting_RLRE_transition i 1),
          (ACSE i).ACSE_Awaiting_RLRE,
          EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
                 ((Initiating_ACSE i) = (C T)),
          ATNSend s (CF i) A_RELEASE_ind (ACSEData i),
          (ACSE i).ACSE_Collision_initiator
        )
      );
    P_RELEASE_ind       .
      (function inMessage .
        ( (ACSE_P_RELEASE_ind_ACSE_Awaiting_RLRE_transition i 2),
          (ACSE i).ACSE_Awaiting_RLRE,
          EvCond (ATNReceive s (CF i) inMessage (ACSEData i))
                 ((Initiating_ACSE i) = (C F)),
          ATNSend s (CF i) A_RELEASE_ind (ACSEData i),
          (ACSE i).ACSE_Collision_responder
        )
      );

    P_RELEASE_cnf_pos   . (TRANS_CELL A_RELEASE_cnf_pos ACSE_Idle);
    P_RELEASE_cnf_neg   . (TRANS_CELL A_RELEASE_cnf_neg
                                                  ACSE_Associated);
    /* Releasing connection abnormally */
    A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
    P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
    P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
  ];



Transitions_From_ACSE_Awaiting_ARLSrsp s i :=
  /* From ACSE_Awaiting_ARLSrsp state (STA4) */
  let Error_Cell := (ACSE_error s i ACSE_Awaiting_ARLSrsp) in
  let TRANS_CELL := (ACSE_TRANS s i ACSE_Awaiting_ARLSrsp) in

  [ /* Making connection */
    A_ASSOCIATE_req     . Error_Cell;
    A_ASSOCIATE_rsp_pos . Error_Cell;
```

```
      A_ASSOCIATE_rsp_neg . Error_Cell;
      P_CONNECT_ind       . Error_Cell;
      P_CONNECT_cnf_pos   . Error_Cell;
      P_CONNECT_cnf_neg   . Error_Cell;
      /* Releasing connection normally */
      A_RELEASE_req       . Error_Cell;
      A_RELEASE_rsp_pos   . (TRANS_CELL P_RELEASE_rsp_pos ACSE_Idle);
      A_RELEASE_rsp_neg   . (TRANS_CELL P_RELEASE_rsp_neg
                                                ACSE_Associated);
      P_RELEASE_ind       . Error_Cell;
      P_RELEASE_cnf_pos   . Error_Cell;
      P_RELEASE_cnf_neg   . Error_Cell;
      /* Releasing connection abnormally */
      A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
      P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
      P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
    ];


  Transitions_From_ACSE_Associated s i :=
    /* From ACSE_Associated state (STA5) */
    let Error_Cell := (ACSE_error s i ACSE_Associated) in
    let TRANS_CELL := (ACSE_TRANS s i ACSE_Associated) in

    [ /* Making connection */
      A_ASSOCIATE_req     . Error_Cell;
      A_ASSOCIATE_rsp_pos . Error_Cell;
      A_ASSOCIATE_rsp_neg . Error_Cell;
      P_CONNECT_ind       . Error_Cell;
      P_CONNECT_cnf_pos   . Error_Cell;
      P_CONNECT_cnf_neg   . Error_Cell;
      /* Releasing connection normally */
      A_RELEASE_req       . (TRANS_CELL P_RELEASE_req
                                            ACSE_Awaiting_RLRE);
      A_RELEASE_rsp_pos   . Error_Cell;
      A_RELEASE_rsp_neg   . Error_Cell;
      P_RELEASE_ind       . (TRANS_CELL A_RELEASE_ind
                                         ACSE_Awaiting_ARLSrsp);
      P_RELEASE_cnf_pos   . Error_Cell;
      P_RELEASE_cnf_neg   . Error_Cell;
      /* Releasing connection abnormally */
      A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
      P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
      P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
  ];



 Transitions_From_ACSE_Collision_initiator s i :=
   /* From ACSE_Collision_initiator state (STA6) */
   let Error_Cell := (ACSE_error s i ACSE_Collision_initiator) in
   let TRANS_CELL := (ACSE_TRANS s i ACSE_Collision_initiator) in

   [ /* Making connection */
     A_ASSOCIATE_req     . Error_Cell;
     A_ASSOCIATE_rsp_pos . Error_Cell;
     A_ASSOCIATE_rsp_neg . Error_Cell;
     P_CONNECT_ind       . Error_Cell;
     P_CONNECT_cnf_pos   . Error_Cell;
     P_CONNECT_cnf_neg   . Error_Cell;
     /* Releasing connection normally */
     A_RELEASE_req       . Error_Cell;
     A_RELEASE_rsp_pos   . (TRANS_CELL P_RELEASE_rsp_pos
                                          ACSE_Awaiting_RLRE);
     A_RELEASE_rsp_neg   . Error_Cell;
     P_RELEASE_ind       . Error_Cell;
     P_RELEASE_cnf_pos   . Error_Cell;
     P_RELEASE_cnf_neg   . Error_Cell;
     /* Releasing connection abnormally */
     A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
     P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
     P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
   ];



 Transitions_From_ACSE_Collision_responder s i :=
   /* From ACSE_Collision_responder state (STA7) */
   let Error_Cell := (ACSE_error s i ACSE_Collision_responder) in
   let TRANS_CELL := (ACSE_TRANS s i ACSE_Collision_responder) in

   [ /* Making connection */
     A_ASSOCIATE_req     . Error_Cell;
     A_ASSOCIATE_rsp_pos . Error_Cell;
     A_ASSOCIATE_rsp_neg . Error_Cell;
     P_CONNECT_ind       . Error_Cell;
     P_CONNECT_cnf_pos   . Error_Cell;
     P_CONNECT_cnf_neg   . Error_Cell;
     /* Releasing connection normally */
```

```
      A_RELEASE_req       . Error_Cell;
      A_RELEASE_rsp_pos   . Error_Cell;
      A_RELEASE_rsp_neg   . Error_Cell;
      P_RELEASE_ind       . Error_Cell;
      P_RELEASE_cnf_pos   . (TRANS_CELL A_RELEASE_cnf_pos
                                              ACSE_Awaiting_ARLSrsp);
      P_RELEASE_cnf_neg   . Error_Cell;
      /* Releasing connection abnormally */
      A_ABORT_req         . (TRANS_CELL P_U_ABORT_req     ACSE_Idle);
      P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind       ACSE_Idle);
      P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind     ACSE_Idle)
    ];




ACSE_sc i :=
  OrState (ACSE i) ((ACSE i).ACSE_Idle) [
    /* State names */
    BasicState ((ACSE i).ACSE_Idle);
    BasicState ((ACSE i).ACSE_Awaiting_AARE);
    BasicState ((ACSE i).ACSE_Awaiting_AASCrsp);
    BasicState ((ACSE i).ACSE_Awaiting_RLRE);
    BasicState ((ACSE i).ACSE_Awaiting_ARLSrsp);
    BasicState ((ACSE i).ACSE_Associated);
    BasicState ((ACSE i).ACSE_Collision_initiator);
    BasicState ((ACSE i).ACSE_Collision_responder)
  ]
    /* Transitions: from ISO 8650, Table 14 */
    (append (Transitions_From_ACSE_Idle (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Awaiting_AARE (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Awaiting_AASCrsp (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Awaiting_RLRE (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Awaiting_ARLSrsp (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Associated (ACSE_sc i) i)
    (append (Transitions_From_ACSE_Collision_initiator (ACSE_sc i) i)
     (Transitions_From_ACSE_Collision_responder (ACSE_sc i) i)))))));
```

Appendix O

First Example Simulation Run of the ATN

```
Fusion+ - Version 1.0 Jun 13 1998 14:44:27
Copyright University of British Columbia, 1996, 1997

Type "%include <filename>" or type in S+ paragraphs directly
Type "%help" to see list of % commands.
search path: .


>/*
 Simulating message sequence chart on page 2-4.
 Last modified: 1 May 98
 Nancy A. Day

 This MSC shows that a dialogue initiator issuing
 a D_START_req causes a D_START_ind to be sent to
 the dialogue responder.  This can then be followed by
 the dialogue responder issuing a D_START_rsp which
 causes a D_START_conf to be sent to the dialogue
 initiator.

 Here the dialogue initiator is CM_Ground and the
 responder is CM_Air.  A D_START_req is caused by
 the AppUser sending a CM_UPDATE_req to CM_Ground.
```

```
*/;
;
>%set keepOldNodes OFF
keepOldNodes = 0
>%set sfeEvaluationLevel 3
sfeEvaluationLevel = 3
>%include load_system.s+
Including ./load_system.s+
Including ./atncommon.s+
search path: . ../..
Including ../../all.s+
Including ../../basics.s+
Closing basics.s+
Including ../../framework.s+
Closing framework.s+
Including ../../bvfcns.s+
Closing bvfcns.s+
Including ../../table.s+
Closing table.s+
Including ../../events.s+
Closing events.s+
Including ../../actions.s+
Closing actions.s+
Including ../../sc.s+
Including ../../scsyn.s+
Closing scsyn.s+
Including ../../scsem.s+
Closing scsem.s+
Closing sc.s+
Including ../../sc_ext_exp.s+
Closing sc_ext_exp.s+
Including ../../sc_ext_ev.s+
Closing sc_ext_ev.s+
Including ../../sc_comm_prim.s+
Closing sc_comm_prim.s+
Including ../../sc_comm_action.s+
Closing sc_comm_action.s+
Including ../../sc_comm_event.s+
Closing sc_comm_event.s+
Closing all.s+
Closing atncommon.s+
Including ./cf_data.s+
Closing cf_data.s+
Including ./acse_data.s+
Closing acse_data.s+
Including ./suppsvc_data.s+
Closing suppsvc_data.s+
Including ./cm_data.s+
Closing cm_data.s+
Including ./system_subst.s+
Closing system_subst.s+
Setting up substitution order
.
.
.
Finished setting up substitution order
Bdd of atn_nc, size 15553 successfully loaded.
Bdd of atn_ic, size 165 successfully loaded.
Closing load_system.s+
>
%set filterOutput 1
filterOutput = 1
>page2_4 :=
[
/* cf0 */  atn_ic /\ Msg (AppUser 1) (CM_Ground 1) (CM_UPDATE_req) cf;
/* cf1 */  Msg (CM_Ground 1) (CF 1) D_START_req cf;
/* cf2 */  ((ACSEVersionSupported 1) = (C T)) cf;
/* cf3 */  T;
/* cf4 */  T;
/* cf5 */  T;
/* cf6 */  ((ACSEVersionSupported 2) = (C T)) cf;
/* cf7 */  T;
/* cf8 */  (dataCM_USER 2 = C CM_UPDATE_ind) cf;
               /* Msg (CF 2) (CM_Air 2) D_START_ind cf */
/* cf9 */  T;
/* cf10 */ T;
/* cf11 */ T;
/* cf12 */ T;
/* cf13 */ T;
/* cf14 */ T;
/* cf15 */ T;
/* cf16 */ T
];
page2_4 :=
  [
    (atn_ic /\
      ((((Msg (AppUser 1)) (CM_Ground 1)) CM_UPDATE_req) cf));
    ((((Msg (CM_Ground 1)) (CF 1)) D_START_req) cf);
    (((ACSEVersionSupported 1) = (C T)) cf);T;T;T;
    (((ACSEVersionSupported 2) = (C T)) cf);T;
    (((dataCM_USER 2) = (C CM_UPDATE_ind)) cf);T;T;T;T;T;T;
    T;T];
>

/*
 don't need one_ahead because there is little non-determinism.
 A few extra constraints are needed along the way
*/

%simulate atn_nc page2_4

nsr BDD exists, size: 15553

Evaluating condition on first configuration
Converting condition on first configuration to BDD

Configuration 0:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA0 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((((Msg (AppUser 1)) (CM_Ground 1)) CM_UPDATE_req) cf) EQ T
((InBasicState (CM_Ground_IDLE 1)) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg (CM_Ground 1)) (CF 1)) D_START_req) cf) EQ T
((InBasicState (CF_STA0 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD
```

```
Configuration 2:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((ACSEVersionSupported 1) cf) EQ T
((((Msg (CF 1)) (ACSE 1)) A_ASSOCIATE_req) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 3:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg (ACSE 1)) (CF 1)) P_CONNECT_req) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 4:
((((Msg (CF 1)) ((SuppSvc 1) 2)) P_CONNECT_req) cf) EQ T
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
```

```
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 5:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg ((SuppSvc 1) 2)) (CF 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 6:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((ACSEVersionSupported 2) cf) EQ T
((((Msg (CF 2)) (ACSE 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 7:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg (ACSE 2)) (CF 2)) A_ASSOCIATE_ind) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
```

```
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 8:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((((Msg (CF 2)) (CM_Air 2)) D_START_ind) cf) EQ T
(((dataCM_USER 2) cf) EQ CM_UPDATE_ind) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 9:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg (CM_Air 2)) (CF 2)) D_START_rsp_pos) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
((((Msg (CM_Air 2)) (AppUser 2)) CM_UPDATE_ind) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 10:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
```

```
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((((Msg (CF 2)) (ACSE 2)) A_ASSOCIATE_rsp_pos) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 11:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg (ACSE 2)) (CF 2)) P_CONNECT_rsp_pos) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 12:
((((Msg (CF 2)) ((SuppSvc 1) 2)) P_CONNECT_rsp_pos) cf) EQ T
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA2 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 13:
```

```
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg ((SuppSvc 1) 2)) (CF 1)) P_CONNECT_cnf_pos) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA2 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 14:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA2 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((((Msg (CF 1)) (ACSE 1)) P_CONNECT_cnf_pos) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 15:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg (ACSE 1)) (CF 1)) A_ASSOCIATE_cnf_pos) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA2 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
```

```
Converting condition on next configuration to BDD

Configuration 16:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((AmInitiator 1) cf) EQ T
((InBasicState (CF_STA2 1)) cf) EQ T
((InBasicState (CF_STA2 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Associated (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_UPDATE 1)) cf) EQ T
(CM_TIMEOUT_update_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
((((Msg (CF 1)) (CM_Ground 1)) D_START_cnf_pos) cf) EQ T
>
Cleaning up node use.
Cleaning up bdd use.
Fusion session over.
```

Appendix P

Second Example Simulation Run of the ATN

```
Fusion+ - Version 1.0 Jun 13 1998 14:44:27
Copyright University of British Columbia, 1996, 1997

Type "%include <filename>" or type in S+ paragraphs directly
Type "%help" to see list of % commands.
search path: .


>/*
 Illustrate simulation of a full ATN that results in
 behaviour found in model checking half of the ATN.
 Last modified: 4 May 98
 Nancy A. Day
*/

%set keepOldNodes OFF
keepOldNodes = 0
>%set sfeEvaluationLevel 3
sfeEvaluationLevel = 3
>%include load_system.s+
Including ./load_system.s+
Including ./atncommon.s+
search path: . ../..
Including ../../all.s+
Including ../../basics.s+
Closing basics.s+
Including ../../framework.s+
Closing framework.s+
Including ../../bvfcns.s+
Closing bvfcns.s+
Including ../../table.s+
Closing table.s+
Including ../../events.s+
Closing events.s+
Including ../../actions.s+
Closing actions.s+
Including ../../sc.s+
Including ../../scsyn.s+
Closing scsyn.s+
Including ../../scsem.s+
Closing scsem.s+
Closing sc.s+
Including ../../sc_ext_exp.s+
Closing sc_ext_exp.s+
Including ../../sc_ext_ev.s+
Closing sc_ext_ev.s+
Including ../../sc_comm_prim.s+
Closing sc_comm_prim.s+
Including ../../sc_comm_action.s+
Closing sc_comm_action.s+
Including ../../sc_comm_event.s+
Closing sc_comm_event.s+
Closing all.s+
Closing atncommon.s+
Including ./cf_data.s+
Closing cf_data.s+
Including ./acse_data.s+
Closing acse_data.s+
Including ./suppsvc_data.s+
Closing suppsvc_data.s+
Including ./cm_data.s+
Closing cm_data.s+
Including ./system_subst.s+
Closing system_subst.s+
Setting up substitution order
.
.
.
Finished setting up substitution order
```

```
Bdd of atn_nc, size 15553 successfully loaded.
Bdd of atn_ic, size 165 successfully loaded.
Closing load_system.s+
>
%set filterOutput 1
filterOutput = 1
>CannotOccurSim :=
[
atn_ic /\
  Msg (AppUser 1) (CM_Ground 1) CM_CONTACT_req cf;
Msg (CM_Ground 1) (CF 1) D_START_req cf;
Msg (CF 1) (ACSE 1) A_ASSOCIATE_req cf /\
  ((ACSEVersionSupported 1) = (C T)) cf;
Msg (ACSE 1) (CF 1) P_CONNECT_req cf;
Msg (CF 1) (SuppSvc 1 2) P_CONNECT_req cf;
Msg (SuppSvc 1 2) (CF 2) P_CONNECT_ind cf;
Msg (CF 2) (ACSE 2) P_CONNECT_ind cf /\
  ((ACSEVersionSupported 2) = (C T)) cf;
Msg (ACSE 2) (CF 2) A_ASSOCIATE_ind cf /\
  Msg (AppUser 2) (CM_Air 2) CM_LOGON_req cf;
Msg (CF 2) (CM_Air 2) D_START_ind cf /\
  Msg (CM_Air 2) (CF 2) D_START_req cf /\
  (dataCM_USER 2 = C CM_CONTACT_req) cf;
Msg (CM_Air 2) Error CannotOccur cf
];
CannotOccurSim :=
  [
    (atn_ic /\
      ((((Msg (AppUser 1)) (CM_Ground 1)) CM_CONTACT_req) cf));
    ((((Msg (CM_Ground 1)) (CF 1)) D_START_req) cf);
    (((((Msg (CF 1)) (ACSE 1)) A_ASSOCIATE_req) cf) /\
      (((ACSEVersionSupported 1) = (C T)) cf));
    ((((Msg (ACSE 1)) (CF 1)) P_CONNECT_req) cf);
    ((((Msg (CF 1)) ((SuppSvc 1) 2)) P_CONNECT_req) cf);
    ((((Msg ((SuppSvc 1) 2)) (CF 2)) P_CONNECT_ind) cf);
    (((((Msg (CF 2)) (ACSE 2)) P_CONNECT_ind) cf) /\
      (((ACSEVersionSupported 2) = (C T)) cf));
    (((((Msg (ACSE 2)) (CF 2)) A_ASSOCIATE_ind) cf) /\
      ((((Msg (AppUser 2)) (CM_Air 2)) CM_LOGON_req) cf));
    (((((Msg (CF 2)) (CM_Air 2)) D_START_ind) cf) /\
      (((((Msg (CM_Air 2)) (CF 2)) D_START_req) cf) /\
        (((dataCM_USER 2) = (C CM_CONTACT_req)) cf)));
    ((((Msg (CM_Air 2)) Error) CannotOccur) cf)];
>%simulate atn_nc CannotOccurSim
```

```
nsr BDD exists, size: 15553

Evaluating condition on first configuration
Converting condition on first configuration to BDD

Configuration 0:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA0 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((((Msg (AppUser 1)) (CM_Ground 1)) CM_CONTACT_req) cf) EQ T
((InBasicState (CM_Ground_IDLE 1)) cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 1:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg (CM_Ground 1)) (CF 1)) D_START_req) cf) EQ T
((InBasicState (CF_STA0 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 2:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((ACSEVersionSupported 1) cf) EQ T
((((Msg (CF 1)) (ACSE 1)) A_ASSOCIATE_req) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
```

```
calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 3:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((((Msg (ACSE 1)) (CF 1)) P_CONNECT_req) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 4:
((((Msg (CF 1)) ((SuppSvc 1) 2)) P_CONNECT_req) cf) EQ T
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 5:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg ((SuppSvc 1) 2)) (CF 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (CF_STA0 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
```

```
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 6:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((ACSEVersionSupported 2) cf) EQ T
((((Msg (CF 2)) (ACSE 2)) P_CONNECT_ind) cf) EQ T
((InBasicState (ACSE_Idle (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T

calculating next config took: 1 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 7:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((((Msg (ACSE 2)) (CF 2)) A_ASSOCIATE_ind) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_IDLE 2)) cf) EQ T
((((Msg (AppUser 2)) (CM_Air 2)) CM_LOGON_req) cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 8:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
```

```
((((Msg (CM_Air 2)) (CF 2)) D_START_req) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
(((dataCM_USER 2) cf) EQ CM_CONTACT_req) EQ T
((((Msg (CF 2)) (CM_Air 2)) D_START_ind) cf) EQ T
((InBasicState (CM_Air_LOGON 2)) cf) EQ T
(CM_TIMEOUT_logon_active cf) EQ T

calculating next config took: 0 sec bdd size: 193
Evaluating condition on next configuration
Converting condition on next configuration to BDD

Configuration 9:
((InBasicState (SuppSvc_Translate ((SuppSvc 1) 2))) cf) EQ T
((InBasicState (CF_STA1 1)) cf) EQ T
((InBasicState (CF_STA1 2)) cf) EQ T
((Initiating_ACSE 1) cf) EQ T
((InBasicState (ACSE_Awaiting_AARE (ACSE 1))) cf) EQ T
((InBasicState (ACSE_Awaiting_AASCrsp (ACSE 2))) cf) EQ T
((InBasicState (CM_Ground_CONTACT 1)) cf) EQ T
(CM_TIMEOUT_contact_active cf) EQ T
((InBasicState (CM_Air_LOGON 2)) cf) EQ T
(CM_TIMEOUT_logon_active cf) EQ T
((((Msg (CM_Air 2)) Error) CannotOccur) cf) EQ T
>

Cleaning up node use.
Cleaning up bdd use.
Fusion session over.
```