A Model Checker for Statecharts

(Linking CASE tools with Formal Methods)

by
Nancy Day

Technical Report 93-35
October 1993

Department of Computer Science
University of British Columbia
Rm 201 - 2366 Main Mall
Vancouver, B.C.
CANADA V6T 1Z4

Telephone: (604) 822-3061
Fax: (604) 822-5485

A Model Checker for Statecharts
(Linking CASE tools with Formal Methods)

Nancy Day
Integrated Systems Design Laboratory
Department of Computer Science
University of British Columbia
e-mail: daycs.ubc.ca

18 October 1993

Abstract. Computer-Aided Software Engineering (CASE) tools encourage users to codify the specification for the design of a system early in the development process. They often use graphical formalisms, simulation, and prototyping to help express ideas concisely and unambiguously. Some tools provide little more than syntax checking of the specification but others can test the model for reachability of conditions, nondeterminism, or deadlock.

Formal methods include powerful tools like automatic model checking to exhaustively check a model against certain requirements. Integrating formal techniques into the system development process is an effective method of providing more thorough analysis of specifications than conventional approaches employed by Computer-Aided Software Engineering (CASE) tools. In order to create this link, the formalism used by the CASE tool must have a precise formal semantics that can be understood by the verification tool.

The CASE tool STATEMATE makes use of an extended state transition notation called statecharts. We have formalized an operational semantics for statecharts by embedding them in the logical framework of an interactive proof-assistant system called HOL. A software interface is provided to extract a statechart directly from the STATEMATE database.

Using HOL in combination with Voss, a binary decision diagram-based verification tool, we have developed a model checker for statecharts which tests whether an operational specification, given by a statechart, satisfies a descriptive specification of the system requirements. The model checking procedure is a simple higher-order logic function which executes the semantics of statecharts.

In this thesis, we describe the formal semantics of statecharts and the model checking algorithm. Various examples, including an intersection with a traffic light and an arbiter, are presented to illustrate the method.

This work was submitted in partial fulfillment of requirements for a Master of Science degree at the University of British Columbia, September, 1993.

Contents

List of Tables

List of Figures

Acknowledgement

The credit for the idea for this work and many of the key insights along the way belongs to my supervisor Jeff Joyce. He has been a constant source of encouragement and enthusiasm and as he is just starting out, I know that many students in the future will both benefit and enjoy working with him. I would also like to thank Carl Seger, Mark Greenstreet and the other members of the Integrated Systems Design Group for creating an interesting and friendly atmosphere in which to do research. As much as a person can look forward to another thesis at this time, I am happy to be staying here to study for my doctorate. The graduate students at UBC are a terrific group of people and in particular I would like to acknowledge the help of Andy Martin and Michael McAllister who have read many of my papers and taken part in discussions about semantics, model checkers and even LaTeX, always offering constructive comments. Through the last two years I have been funded by the Natural Science and Engineering Research Council of Canada and the Department of Computer Science at UBC.

Finally, as always, my family and friends have been my support the last two years. In particular, I want to thank my parents, whose constant belief that I can do anything I set my mind to, has made me always reach a little beyond what I achieve each day.

Chapter 1

Introduction

Integrating formal techniques into the system development process is an effective method of providing more thorough analysis of specifications than achieved by conventional approaches employed by Computer-Aided Software Engineering (CASE) tools. We begin by describing the existing capabilities of the CASE tool STATEMATE for specification analysis and how it can benefit from formal techniques. The approach taken in this work is to create a model checker for statecharts in the hybrid verification tool HOL-Voss.

1.1 Introduction

Previous work has stated that errors introduced in the specification stage of the system development process are often the most costly to correct [21]. Computer-Aided Software Engineering (CASE) tools are mechanical aids to the system specifier. The ability to analyze these specifications can help eliminate errors at this early stage and ensure that the specification has its intended meaning. Formal methods, such as model checking, have been developed to analyze specifications. This work describes a particular example of linking the CASE tool STATEMATE with a model checker. The main conclusion is that formal techniques are an effective method for providing more thorough analysis of specifications than achieved by conventional approaches employed by CASE tools.

1.2 CASE Tools

CASE tools are intended to help the system developer by providing ways of codifying requirements early in the process. The specification is developed in a graphical notation which is intended to be an improvement over natural language but may still be open to interpretation. In this work, we focus on CASE tools used commercially by software engineers who are not familiar with formal methods.

The specification that the user creates with the CASE tool is usually an *operational* model. It can be considered a very abstract view of the system implementation. This model can often be simulated or executed although it may include non-determinism. Examples of operational specification notations supported by CASE tools include data flow diagrams, petri nets and finite state machines [5].

1.3 Specification Analysis

Once a specification has been created, it is useful to analyze it before proceeding with system development. Some CASE tools provide little more than syntax and type checking of the notation, but others exploit the possibilities for doing further analysis of the requirements. Simulation and prototyping help ensure that the requirements are complete and that they capture the intended behaviour of the system. Tests for deadlock, non-determinism, and race conditions are all useful for checking general properties of the system.

Given an operational model of the system, we can also ask whether it has particular properties. Safety or liveness conditions can be checked at this initial stage of specification. For example, a model of a traffic light at a two way intersection should have the property that at least one of the lights is red at all times. These properties are called *descriptive* specifications. They are often global conditions which should be satisfied throughout the system's execution.

1.4 STATEMATE

The CASE tool STATEMATE uses a graphical extended state transition notation called statecharts as the operational specification notation for real-time systems. STATEMATE integrates tools to analyze and execute the model [14].

The STATEMATE Simulator provides interactive or batch mode executions of the model. It relies on the user to play the role of the environment by changing the values of external data items. In cases where the model is non-deterministic, the user can choose or the system will randomly select one execution path to follow.

STATEMATE's Dynamic Analysis tests provide more comprehensive examination of the model for particular properties. The "reachability of conditions" test checks whether the system ever reaches a point in execution where certain conditions, given in the syntax of statechart Boolean expressions, hold true. This test is not completely comprehensive because initial or default values for internal data-items and events must be given. A range of values can be assigned to external data-items. A test is performed for each different value within this range, but it is unclear from the manual whether the value is constant throughout the test, or whether all different possible values are considered at each decision point. The second interpretation is the more conservative and the more appropriate since the system has no control over when the value of an external data-item is changed. The user must also give a limit on the number of execution steps that the model will take while performing these checks. In cases where the condition is reached, the execution path followed to arrive at that point is documented.

1.5 Formal Methods

While type-checking or testing a model for general properties like non-determinism could be considered formal methods, we will use the term in a more specialized way. In this work, the term "formal methods" encompasses a range of techniques where principles of reasoning and mathematics are used to examine models more thoroughly than can be achieved by traditional testing and simulation. These techniques include both interactive and automatic theorem proving, and model checking. The test for reachability of conditions in STATEMATE is a restricted form of model checking.

1.6 Linking CASE tools with Formal Methods

The intent of this work is to determine if by using formal techniques it is possible to do more thorough analysis of specifications beyond the ability of conventional methods employed by commercial CASE tools. To carry out any type of formal analysis, precise semantics are required for both the descriptive and the operational specifications. Statecharts were chosen as the language for the operational specifications because they are supported by a CASE tool and because they already have a reasonably well-developed semantics.

In general, the automatic formal techniques are more appealing to non-experts than the interactive tools. Harel [11] and others have suggested that automatic verification techniques could be integrated effectively into system analysis tools. Model checking is an automatic way of verifying properties of an operational model. The link to CASE tools is provided by a precise semantics for the operational specification notation. However, much of the work to formalize these semantics and create the infrastructure to connect a CASE tool with a model checker can be carried out by an expert. The result is a tool that can be used by non-experts to verify properties of their model automatically.

This thesis presents a formalization of the semantics of statecharts used in the CASE tool STATEMATE and the implementation of a model checker for statecharts in an existing verification tool. We potentially improve upon the existing test for reachability of conditions within STATEMATE in the following ways:

1. allowing symbolic values in the expression of the properties,

2. making the semantics adaptable to suit variations of the statechart notation, and

3. providing a framework for using more expressive descriptive specification languages.

It is not entirely clear from the STATEMATE manual to what extent symbolic functionality properties can be proven. For example, we would like to set an initial state where a variable has the value a and, given an increment operation, the model checker could prove that the resulting value is $a + 1$. We also want to be certain that the model checker recognizes that external data-items can change their values at any time and therefore examines branching execution paths. Both of these are accomplished using symbolic values for variables which means many values for a variable can be checked with one run of the model checker.

Chapter 2 presents situations in the semantics that can be interpreted in various ways, pointing out areas where others might wish to make different choices. By making the semantics used in the model checker explicit, it should not be difficult to adapt them to suit variations of the formalism. Leveson et al.'s Requirements State Machine Language (RSML) [21] falls into this category.

Properties that we wish to verify can often only be expressed in more complex descriptive specification languages. Computational Tree Logic (CTL) is an example of such a language which includes temporal operators in its expressions. Given a decision procedure our model checker can be adapted to a language that includes these features, using the same semantic definitions that we supply. Section 9.5 describes one option for a more descriptive specification language called State Transition Assertions.

1.7 The Overall Method

Given an operational specification of system created in STATEMATE, can we create the links necessary to use a model checker to answer the question of whether a given operational specification satisfies certain descriptive requirements?

Formal methods rely on having a precise semantics for the language used to describe the model. Statecharts were developed with an accompanying semantics that has since been refined and given in different forms by various authors [8][26][21][13]. These descriptions often differ from each other or do not always discuss some of the more subtle aspects of the semantics. Therefore, we also had to use our intuition to determine the meaning of statecharts. We have embedded an operational semantics for statecharts as a next configuration relation in a target language.

The target language is a subset of higher-order logic that can be informally regarded as a functional programming language. Details of this language and any functions used in the semantics but not defined there can be found in Appendix A (MAP, MEMBER, etc.).

The descriptive specification gives an initial set of configurations and a condition that must hold along all (or some) execution paths starting at those configurations, within a certain number of steps. A software interface extracts the statechart directly from the STATEMATE database and the model checker tests whether the statechart model satisfies the descriptive requirements.

Many improvements in the speed of model checkers have been made in recent years, most notably giving symbolic values for variables and using binary decision diagrams for efficient representation of configurations. HOL-Voss [28] is a hybrid verification tool that combines an interactive proof-assistant, HOL [7], based on higher-order logic, with an efficient, automatic symbolic simulator, Voss [27], that uses ordered binary decision diagrams (BDDs) [3]. By implementing our model checker in this tool we can take advantage of the expressiveness of higher-order logic to give the semantics of statecharts and then execute the model checking algorithm using BDDs. The model checker is written as a function in higher-order logic that takes a next configuration relation describing the semantics of the model as a parameter. It returns either true or false depending on whether or not the descriptive specification is satisfied. The complete method used to do this is summarized in Figure 1.1.

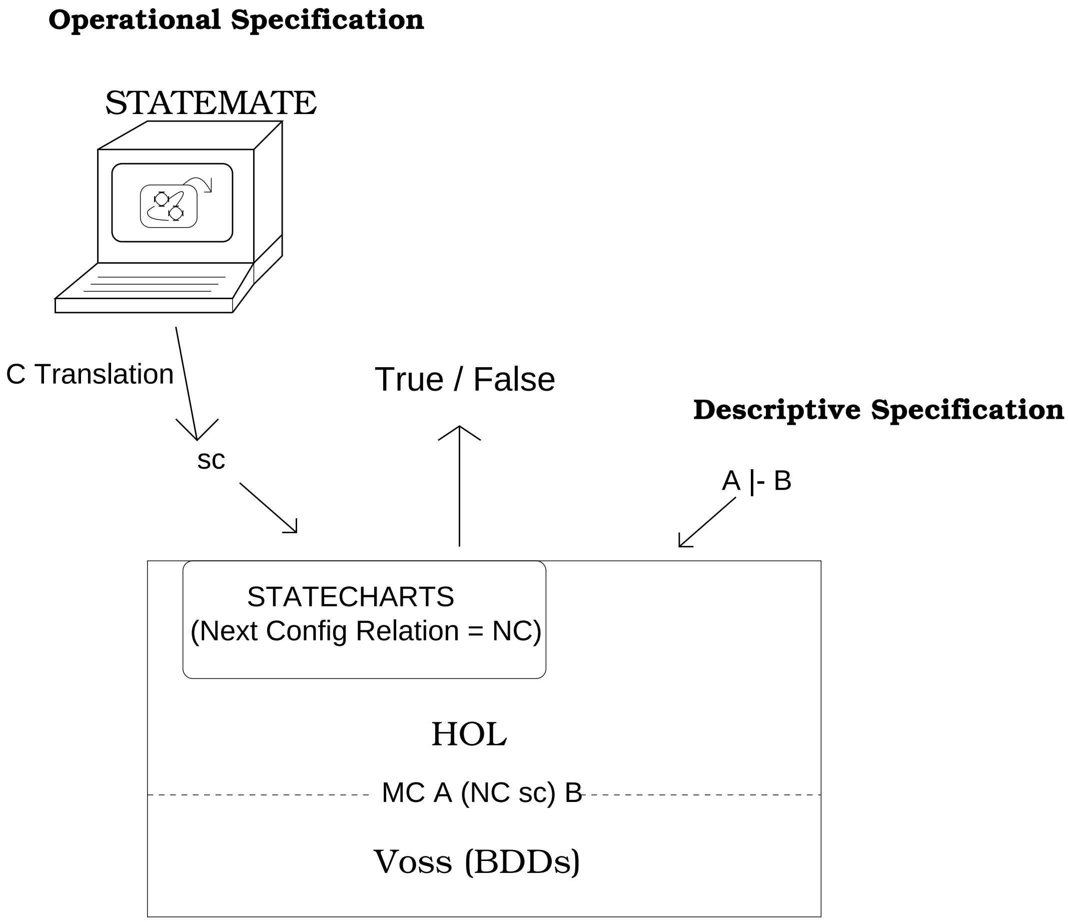

Figure 1.1: The overall method

Given that only the target language is used to express the semantic definitions, the question of why we chose to use HOL should be answered. The first reason for this is that there have already been interesting results from hybrid tools used for hardware verification. Combining a model checker with a theorem prover allows the use of mathematical reasoning techniques like induction and abstraction to prove results beyond the capacity of a model checker [28]. Our model checker is created in HOL-Voss so the theorem-prover is available for this type of use. Sections 6.7 and 9.4 describe ways induction can make use of results returned by the model checker for particular statecharts.

The second reason is that HOL is a theorem-prover in which properties of the semantics of statecharts themselves could be verified. The correctness of our definitions can only be evaluated relative to our interpretation of the meaning of statecharts. Demonstrating overall properties of the semantics would provide a formal basis to our claim that these definitions match our interpretation. Examples of the types of properties we would like to demonstrate about the semantics can be found at the end of Chapter 2.

1.8 Results

By linking CASE tools and formal methods both will benefit. CASE tools provide a graphical interface to create models to be analyzed using formal methods. Formal methods provide exhaustive techniques to verify that a specification created in a CASE tool has certain properties. Cross checking descriptive and operational specifications will increase the specifier's confidence in the result.

The purpose of this work can be summarized in three main goals :

1. to demonstrate that formal methods can be effectively integrated into commercial CASE tools - in particular, tools intended for use by non-experts
2. to formalize an operational semantics for statecharts
3. to create a model checker for a hierarchical graphical specification language where realistic assertions can be verified automatically

The main conclusion of this work is that formal techniques can be integrated into the system development process to provide more thorough analysis of specifications than achieved by conventional methods employed by most commercial CASE tools.

A reader interested in experimenting with the operational semantic definitions given in Chapters 4 and 5 or the model checking functions of Chapter 6 should be able to implement them in a functional programming language such as ML.

1.9 Thesis Outline

The next chapter gives an introduction to statecharts and examines some of the difficulties in giving their semantics by discussing previous work in this area. Chapters 3, 4 and 5 present an abstract syntax for statecharts and the operational semantics that express our interpretation of their behaviour.

Chapter 6 describes the descriptive specification language and the model checking algorithm. Examples of using the model checker on small systems are described in Chapter 7.

Finally, Chapters 9 and 10 present possible extensions to the semantics and the model checker and overall conclusions for this work.

Chapter 2

An Introduction to Statecharts

In this chapter, the graphical statechart notation, used for the operational specification of a system, will be informally introduced. The ideas of concurrency in a hierarchy of states and transitions that move the system between states based on certain triggers and that modify variables are explained briefly through an example of a traffic light at a two way intersection. While statecharts are designed to be a concise and intuitive notation, ambiguous situations can still arise. These are discussed in the second section of this chapter by looking at how our approach differs from existing versions of the semantics. We conclude with a list of properties to characterize the semantics of statecharts. This chapter is followed by three chapters on the syntax and semantics of statecharts that will formalize these ideas.

2.1 Statecharts

There is a great deal of interest from both academia and industry in the statecharts formalism. It is an extended state transition notation for expressing the concurrent operation of real-time systems. It is often described as:

state-diagrams + depth + orthogonality + broadcast-communication [8]

In statecharts, the diagrammatic layout of the notation has meaning beyond just the labels on states and transitions. A hierarchy of states is portrayed in a style similar to set inclusion in Venn diagrams to reduce the complexity of the model and therefore make it more readable. The reader is referred to Harel [10] for an explanation of the origins of statecharts as a type of higraph that combines the elements of graphs and Venn diagrams.

The STATEMATE manual describes a traffic light system controlling a two-way intersection which is a simple but effective example of the expressiveness of statecharts [14]. The statechart for the traffic light controller is given in Figure 2.1 (from Figure 6-22 in [14]) and will be used to illustrate the elements of statecharts.

A statechart models the system as being in a number of *states* which describe its operation. These states are depicted by rounded boxes. A state can be considered a point in the computation. For example, the state labeled **NORMAL**, at the top of the figure represents the normal operation of the lights in both directions. The dashed line through its middle splits it into two substates, north-south(**N_S**) and east-west(**E_W**), which operate concurrently, representing the two directions of the traffic light. **NORMAL** is called an AND-state because it has these orthogonal components. **N_S** and **E_W** are decomposed into substates labeled red, yellow, and green to indicate that when the model is in one of those states, the light is showing that colour, which can be considered the output from this controller. The model can be in only one of them (i.e. red, green or yellow) at any time making **N_S** an OR-state (exclusive-OR).

The representation of these substates within the larger rounded box creates a hierarchy of states (depth). In this hierarchy, the state **NORMAL** is an *ancestor* of **N_S** and **E_W**. Similarly, **N_S** and **E_W** are both

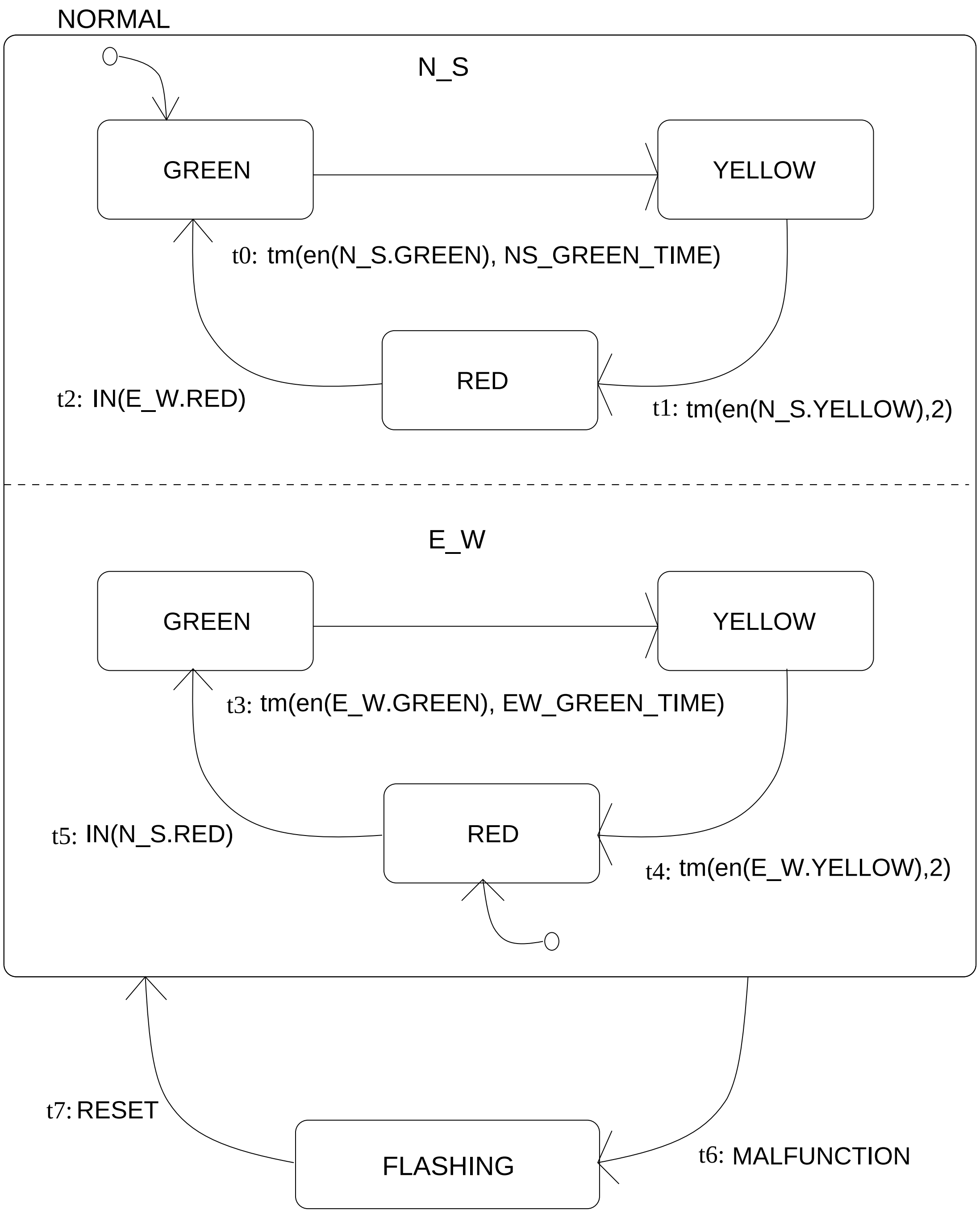

Figure 2.1: Traffic light statechart

descendants of **NORMAL**. When a state is not decomposed into AND or OR-states, it is called a basic state. There are seven basic states in Figure 2.1.

States are connected by *transitions* with labels of the form:

$$\xrightarrow{\hspace{4em} event\ [condition]\ /\ action \hspace{4em}}$$

For reference purposes, we have given each transition a unique name like **t0** or **t1**. If the system is currently in the source state of a certain transition labeled *e[c]/a*, and the event e occurs when the condition c is satisfied then the transition is *enabled. Broadcast communication* is used; this means that all events and the values of any data-items can be referenced anywhere in the system. The event and condition are together referred to as the trigger of the transition.

A condition is a Boolean expression that can include statements like $\mathsf{IN}(x)$ to check whether the system is currently in state x. These are often used to synchronize components as in transition **t5** in the **E_W** state, labeled IN(**N_S.RED**).

An *event* is generated when there is a change in a condition. This is a discrete version of “the instantaneous occurrence of a stimulus” [15]. Entering a state x is a change that causes the event $\mathbf{en}(x)$ to occur. A timeout, $\mathbf{tm}(ev, x)$, is an event that occurs x time units after the event ev. We will call ev the *timeout event* and x the *timeout step number*. Transition **t1** is triggered by the timeout **tm(en(N_S.YELLOW),2)** where **en(N_S.YELLOW)** is the timeout event and **2** is the timeout step number.

Enabled transitions move the system between states. *Following*, or *taking* a transition means exiting its source state, carrying out the actions on its label, and entering its destination state. Informally, following a set of these transitions generally corresponds to a *step* or one time unit. Events occurring in one step can trigger transitions in the next step.

Transitions can be taken in the substates of an AND-state simultaneously. A transition can be enabled if it originates in any ancestor of the current set of basic states. Transitions can also terminate at the outer boundary of a state with substates. *Default arrows*, given diagrammatically as open circles pointing at a state, lead the system into a set of basic states. For example, when transition **t7** is followed, it terminates at the state **NORMAL**, which is made up of two orthogonal components. The default arrows for each of its substates point at **E_W.RED** and **N_S.GREEN**.

If a transition is followed, the action part of the label is carried out and the system moves into the destination state. Actions include generating events or modifying values of variables in the data store through assignment statements. This example does not have any actions on its transitions.

We use the term *configuration* to include the set of states the system is currently in, the values for all data-items, and the events that just occurred.[1] The current set of states alone is called the *state configuration*. A set of basic states is a *legal state configuration* if it satisfies the constraints of the hierarchy. A discrete notion of time is used where the system moves between configurations as a result of stimuli generated both from within the system and externally.

Statecharts often include elements like history states, conditional connectives for transitions, static reactions, and transitions with multiple source and destination states. For simplicity, these are not considered here since they are not included in the subset of statecharts we give semantics for but they will be discussed Chapter 9.

2.2 A Discussion of the Existing Semantic Approaches

The notation described above may seem very straightforward, however statecharts can be created where their intended meaning is not so obvious. These are the situations that make it difficult to give a semantics for statecharts.

The first effort towards a formal semantics for statecharts was by Harel et al. [8]. Pnueli and Shalev [26] pointed out difficulties with the first approach and described revisions. They also show that declarative and operational versions of their semantics are equivalent given a restricted form of the syntax of events.

[1] The STATEMATE manual calls this concept a *status* [14].

The version of statecharts used in STATEMATE has a semantics given by the simulation and analysis tools which is not entirely consistent with Harel et al. [13][14]. We shall also consider the semantics presented by Leveson et al. [21] for the notation called Requirements State Machine Language (RSML) which is a variation of statecharts.

All of this previous work, including less formal discussions of the operation of statecharts [9][10][11][13], has been used to help determine the less obvious features of statecharts and formalize our interpretation of the semantics of statecharts. In the following sections, we will highlight situations where the meaning of a statechart is not graphically apparent and discuss our understanding of the approaches taken to these situations in previous semantics. If one of the four versions given above is not mentioned in a section then either this point was not discussed in their work or they agree with one of the other approaches. Each section concludes with an informal explanation of the interpretation formalized in the next three chapters under the title "Resolution".

2.2.1 What is a Step?

Transitions move the system between states in a statechart. Following a set of transitions and carrying out their actions is considered one time unit or *step* and moves the system between configurations. There are different interpretations of what constitutes this set of transitions.

Intuitively, this set is limited by the following conditions:

1. Any transitions that are followed must be enabled. A transition is enabled if the system is in its source state and its trigger is true.

2. Within an OR-state, only one transition can be followed.

3. Transitions may be followed within each component (substate) of an AND-state.

The set of transitions that are enabled depend on the events generated in the previous step. Each transition may generate events and carry out other actions. Most previous versions of the semantics of statecharts make a distinction between internal and external events and when these are recognized to determine enabled transitions. Internal events are those that are generated as actions of transitions within this statechart. They are often used to sequentially order transitions taken in an operation [8], and therefore transitions enabled by internal events generated in this step should also be taken. Harel et al. calls sets of transitions a *micro-step*. The transitions taken in each micro-step may generate internal events which can trigger other transitions. A step is a maximal sequence of these micro-steps which means at the end there are no more enabled transitions that could be taken still satisfying the three requirements given above. This is called a *stable configuration* [15]. No external events are admitted for consideration during the execution of the micro-steps. This satisfies the synchrony hypothesis which says the system can always compute its complete response to an event before the next external event is ready [26].

When trying to formulate the definition of a step, the factors to consider are:

- Can events generated by actions of transitions followed in this step trigger transitions that are also followed in this step? In Figure 2.2, if the system starts in the states **A** and **C**, and follows **t0**, then the event **f** is generated. Is **t1** then enabled and followed in the same step?

- Are transitions only from the current set of source states considered or can we move through multiple states in a path in one step? From Figure 2.3, we can see that this could lead to infinite loops within a step [14].

- Do events generated in some micro-step persist throughout all the remain micro-steps?

- Which events are generated for the next step?

- When do the actions on the labels of the transitions chosen take effect? And what are the values of variables in the middle of a step?

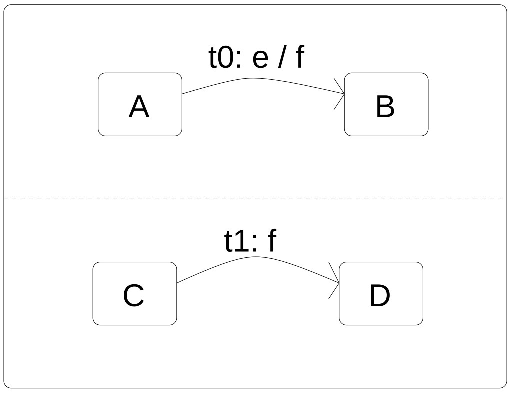

Figure 2.2: What is a step? Example 1

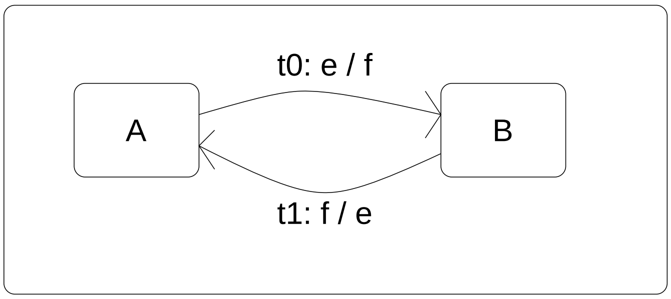

Figure 2.3: What is a step? Example 2

In Harel et al.'s original formulation of the semantics, the set of events used to determine whether transitions are enabled or not consists of the external events given at the beginning of the step and any events generated in micro-steps before the current one. This means events persist for the length of the step but are not used in the next step. A new set of external events must be provided to start the next step. Only transitions from the current set of source states are considered so multiple states in a path can not be taken. The process is guaranteed to terminate since there are a finite number of transitions that can be added to the set taken in a step [26]. At any time during the step, variables still evaluate to their values at the beginning of the step unless special operators called *cr* (current) and *ny* (not yet) are used. At the end of the step the variables take on the cumulative effects of all the modifications made by the transitions.

Pnueli and Shalev [26] points out that Harel et al.'s semantics can result in *global inconsistency* among micro-steps. For example, if a transition is triggered by the event a and it generates the event $\neg a$, another transition may be enabled by the event $\neg a$. Having opposite events both trigger transitions in one step is not consistent. They resolve this problem by using the following definitions for enabled transitions (En) in a step. Given state configuration C, set of transitions T, and external events I:[2]

$$En(T, C, I) = relevant(C) \cap consistent(T) \cap triggered(I \cup generated(T))$$

where:

- $relevant(C)$ is the set of transitions whose source is in the set C
- $consistent(T)$ is the set of transitions that do not conflict with anything in T (for example, if two transitions both leave one state, they can not both be taken and therefore they conflict)
- $triggered(E)$ is the set of transitions whose triggers are satisfied by the complete set E. This is where global inconsistency is eliminated.

[2]Here we present Leveson et al.'s description of this formula; Pnueli and Shalev left out the parameters C and I on the left-hand side of the equation.

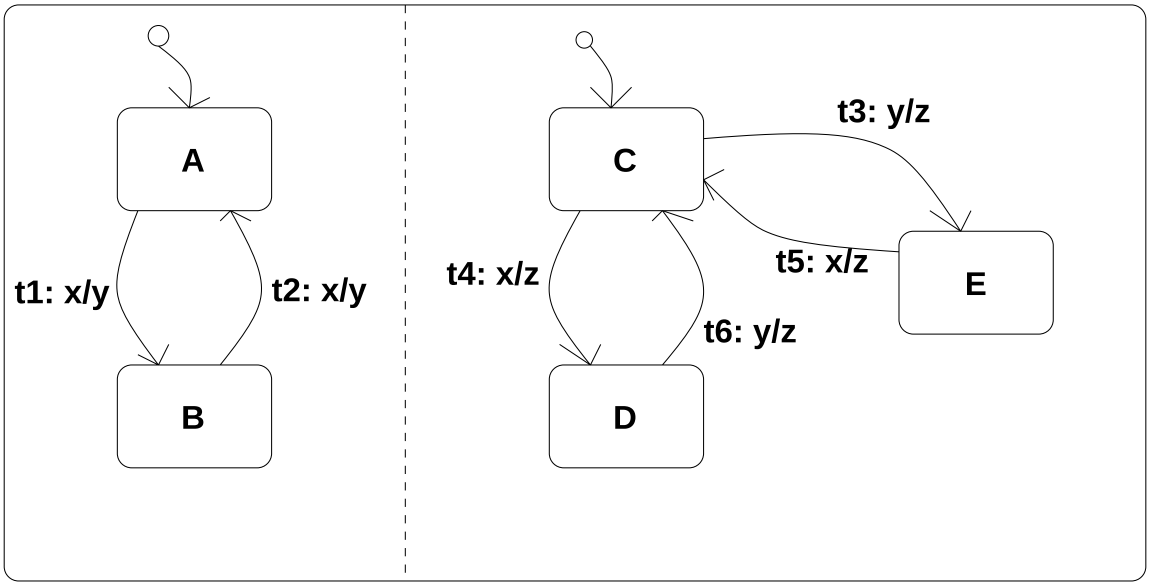

Figure 2.4: Step construction

- $generated(T)$ is the set of events generated by the transitions of T

A random transition is chosen from this set and added to T repeatedly until $En(T, C, I) = T$, i.e. a fixed point is reached. The complete set of transitions T is considered one step. In this description, events last for the duration of the step and only transitions from the state configuration at the beginning of the step are considered.

In Pnueli and Shalev's algorithm, the transitions' actions do not take effect until the next step unless the action involves generating an event. In our opinion, dealing with assignments and events differently is an inconsistent treatment of actions. An assignment action could affect the set of transitions that are taken if transition triggers are conditional upon the values of variables. For example, if a and b true at the beginning of the step and then b is modified, a transition triggered by $a \land \neg b$ may become enabled [2].

Leveson et al. show that Pnueli and Shalev's simulation algorithm which chooses one transition at a time can lead to non-intuitive sets of transitions chosen as a step when considered together. For example in Figure 2.4 (Figure 14 of [21]), if the event x occurs when the system is in states **A** and **C**, Pnueli and Shalev's algorithm could result in the step $\{\mathbf{t1}, \mathbf{t3}\}$ depending on the order in which transitions are chosen from the set $En(T, C, I)$ even though **t4** is enabled by x. (See Table 2.1 from Table 1 in [21])

RSML uses an alternative to this algorithm by first considering and executing in random order all the transitions triggered by the set of external events and then considering the set of transitions triggered by only the internal events generated in this first micro-step, etc. It can also pass through multiple states in a path. This process stops when there are no more enabled transitions. In Figure 2.4, this algorithm chooses $\{\mathbf{t1}, \mathbf{t4}\}$ first. This process may not terminate if the first part of the algorithm generates internal events that return the system to its original configuration (Figure 2.3).

STATEMATE offers two models of timing. In the first, called *step-dependent*, time is incremented after each simulation step. Assignment actions take effect at the end of the step. Internal events generated by these transitions as well as external events can be used to trigger transitions in the next step. If no transitions are enabled, time is still incremented.

The second is called step-independent and time is incremented after a *super-step* in which steps are taken until the system reaches a stable configuration. This is defined as a point where either an external event must occur or time must be incremented, perhaps to generate a timeout event, for the system configuration to change [13]. Unlike the idea of a micro-step, each step in this super-step is executed independently, with actions that take effect at the end of the step, and generated events that may be used in the next step but last only for the duration of one step. Each super-step is considered to take one time unit.

Construction 1			
loop #	T	En(T)	generated(T)
0	$\emptyset$	$\{\mathbf{t_1}, \mathbf{t_4}\}$	$\emptyset$
1	$\{\mathbf{t_1}\}$	$\{\mathbf{t_1}, \mathbf{t_3}, \mathbf{t_4}\}$	$\{\mathbf{y}\}$
2	$\{\mathbf{t_1}, \mathbf{t_3}\}$	$\{\mathbf{t_1}, \mathbf{t_3}\}$	$\{\mathbf{y}, \mathbf{z}\}$

Construction 2			
loop #	T	En(T)	generated(T)
0	$\emptyset$	$\{\mathbf{t_1}, \mathbf{t_4}\}$	$\emptyset$
1	$\{\mathbf{t_4}\}$	$\{\mathbf{t_1}, \mathbf{t_4}\}$	$\{\mathbf{z}\}$
2	$\{\mathbf{t_1}, \mathbf{t_4}\}$	$\{\mathbf{t_1}, \mathbf{t_4}\}$	$\{\mathbf{y}, \mathbf{z}\}$

Table 2.1: Two possible step constructions

The first model is synchronous and the second is asynchronous since it jumps ahead to a point where some change will occur. In essence, a super-step is Leveson et al.'s model.

Resolution: Our semantics use STATEMATE's step-dependent model, where internal events are not distinguished from external ones, and assignments and generating events are treated consistently as actions. This eliminates many of the questions outlined above. For most cases, the simplicity of this interpretation and its formal expression compensate for the loss of the idea of internal events sequencing an operation and the synchrony hypothesis. Chapter 9 describes how our semantic definitions could be used to generate a super-step model of time.

2.2.2 Multiple Actions on a Transition and Race Conditions

A transition may have multiple actions separated by a ';'. If more than one of these actions modify the same variable, what will the value of the variable be at the end of the transition? For example,

$$/x := 1; x := x + 2$$

with 0 as the value of x in the current configuration, may have the following possible interpretations:

1. The actions are taken sequentially so that after following the transition, x has the value 3.

2. The actions are taken relative to the beginning of the step but their effects are evaluated sequentially, therefore the second action takes precedence and the result is that x becomes 2.

3. The actions are evaluated relative to the beginning of the step, and they are not assumed to happen in any particular order, however, the actions are atomic and do not conflict. With this interpretation, the result is that x could be 1 or 2 after the transition is taken.

A race condition occurs if transitions followed simultaneously in orthogonal components modify the same variable in a step. Figure 2.5 provides an example of this, when the configuration includes states **A** and **C** and the event e occurs so the system follows both **t0** and **t1**. This is similar to having multiple assignments on the same transition, but has the added possible interpretation that the actions could conflict with each other (i.e. they are not atomic), and the value for x would then be indeterminate.

Harel et al. do not allow multiple assignments to the same variable within a micro-step. The operators cr and ny can be used to examine modifications made in the sequence of micro-steps.

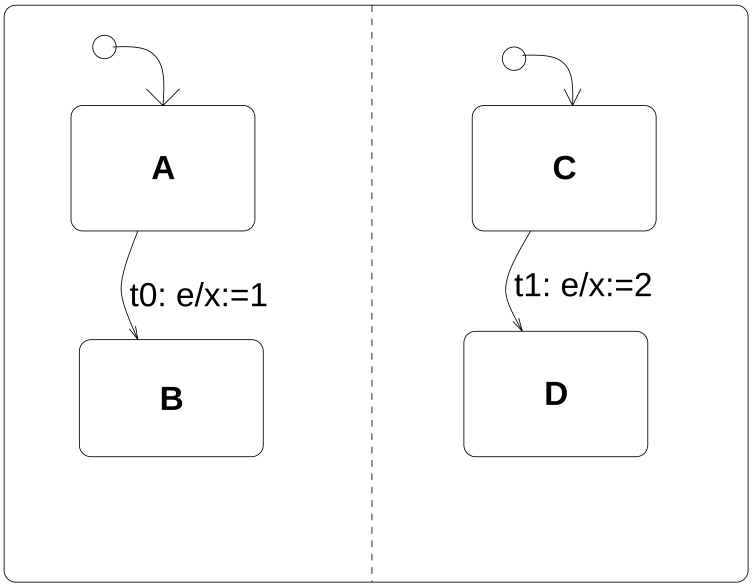

Figure 2.5: Race conditions

Pnueli and Shalev's *consistent*(T) function limits its result to having at most one transition that modifies a given variable.

STATEMATE assumes the assignments are carried out in some random order within a step and so effects can be cumulative in both a step and super-step.

Resolution: By eliminating the idea of micro-steps, our semantics do not encounter the problem of what the value of a modified variable is in the midst of a step. Of the options presented above we have chosen the third one which states that the variable takes on one of its possible values. All actions, including those on the same transition, are considered together when determining the next configuration.

2.2.3 Non-determinism

Enabled transitions may originate at exactly the same state. All previous semantics agree that it is equally likely that any one of them will be taken creating a non-deterministic situation.

Statecharts also have a hierarchy of states and transitions can originate from states at any level in the hierarchy. If multiple transition are enabled from states that are descendants or ancestors of each other in the hierarchy as in Figure 2.6 (**B** and **A**), which transition should be taken? Transitions from parent states are often used to model interrupts or preemption [26].

In Pnueli and Shalev's approach and in STATEMATE, priority is given by the scope (or arena) of the transition. The scope is the lowest OR-state in the hierarchy that is an ancestor state of both the source and destination of the transition. For example, in Figure 2.7 (from Figure 2-10 in [14]), transitions **t0** and **t1** have the same priority since the scope of each is **E**.

Pnueli and Shalev discuss a way of expressing priority using the negation of events. Transitions at lower levels of priority would include in their triggers the negation of the enabling event of higher priority transitions to indicate that these could not be taken. Section 9.3 points out that the negation of events is not a sufficient expression of priority when the destination states of transitions conflict.

Resolution: We base the priority of transitions on source state since making **t0** in Figure 2.7 equally likely to be taken as **t1** does not agree with our intuition. If only state **C** and its transitions are given we should be able to determine its behaviour without knowing the destination of its transitions. These restrictions are used to determine the set of transitions that can be followed in a step.

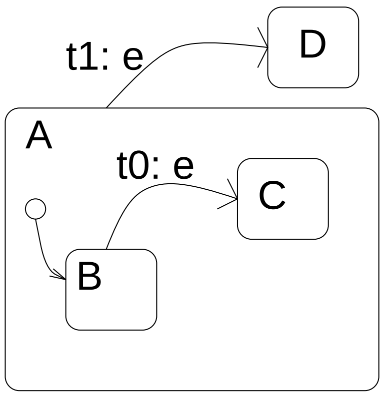

Figure 2.6: Structural non-determinism

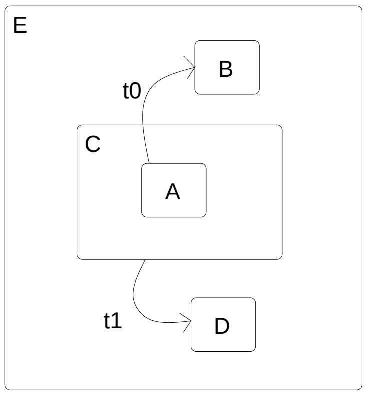

Figure 2.7: Priority

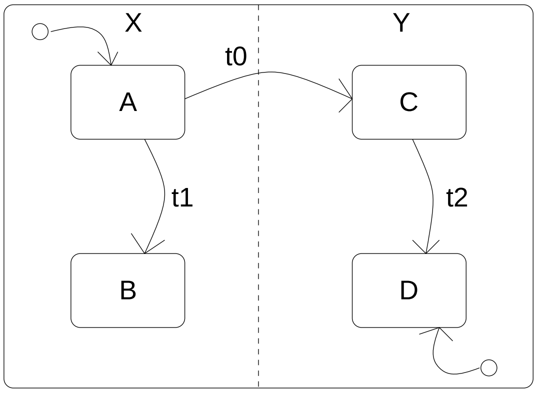

Figure 2.8: Crossing AND-state boundaries - Example 1

2.2.4 Timeouts

When do timeouts occur?

The statechart for the traffic light in Figure 2.1 uses several timeouts to trigger different transitions, such as **t0** or **t3** . When should the system begin to consider the event upon which the timeout is based? Is it the last time the timeout event occurred throughout the system? Another possible interpretation is that the timeout event must occur after we have entered the source state and the system waits the appropriate number of steps before following the transition.

Resolution: All the explanations of the operation of statecharts agree that if x is the timeout step number then a timeout occurs exactly x steps after its timeout event occurs relative to the system as a whole. This is the interpretation used in our semantics.

When is the timeout step number evaluated?

When the timeout step number is a variable, there is the further question of when to evaluate it. Is it evaluated only when the system arrives in the transition's source state (i.e. the first time the transition could be enabled)? Or can the value change between steps? An example of a situation where this might occur in the traffic light is if the **NS_GREEN_TIME** is affected by a pedestrian button which indicates someone wants to cross the street. For example, if the timeout step number currently evaluates to 3 and two steps have passed since the timeout event occurred, then the timeout will not occur in this step. But an action on a transition taken in this step could change the timeout step number to 2. In the next step, three steps will have passed since the timeout event occurred which will not equal the timeout step number that currently holds 2, so the timeout may never occur.

Resolution: This issue is not discussed in any previous versions of the semantics. We have chosen to evaluate the timeout step number in the current configuration of the system. That is, the number may change value while waiting in the source state.

2.2.5 Transitions Among the Components of AND-states

The orthogonal components of AND-states operate concurrently, so it is difficult to see the need for a transition that goes between them. However, these transitions are possible, depending on the definition of a legal statechart. In Figure 2.8, we can see that following transition **t0**, leads into the state **C**, but the system must remain in some state of **X** at all times. At this point, should it follow the default transition into **A** to reach a legal state configuration?

The situation could occur where two transition cross AND-state boundaries at the same time. For example, in Figure 2.9, if **t0** and **t3** are followed at the same time, the system will arrive in states **B** and **C**,

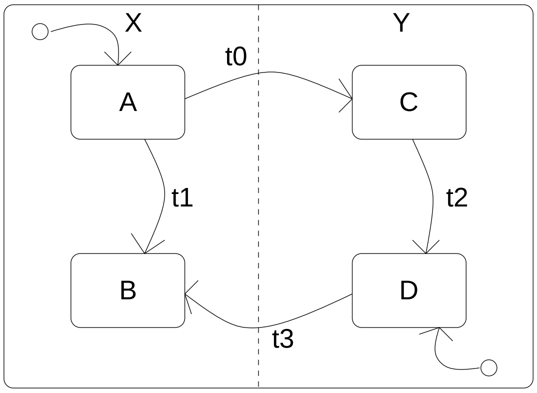

Figure 2.9: Crossing AND-state boundaries - Example 2

which is a legal state configuration.

For the semantics presented in Chapters 4 and 5 we assume that a syntactically correct statechart does not include:

1. transitions that connect components of an AND-state.
2. transitions that cross from within an AND-state to outside the state. To leave an AND-state the transitions must originate at its border.
3. transitions that enter a substate of an AND-state directly. To enter an AND-state the transition must arrive at its border and the defaults followed to enter its components.

Transitions that violate these assumptions could conflict in their destination states even though they exit orthogonal components. Section 9.3 discusses a possible way of modifying the semantics to accommodate these situations.

2.2.6 Configuration Representation

In RSML and STATEMATE the statechart model is augmented with ranges for the values of variables. For a model checker, we need a precise description of each data item to limit the set of possible values. The semantics presented here use an abstract representation which is implemented by bit vectors.

2.3 Properties of Statecharts

We can begin to give a rigorous characterization of the semantics of statecharts, by stating the following properties which include the decisions described above:

2.3.1 Conditions on the State Configuration

1. If the system is in an OR-state, it must be in exactly one of the OR-state's substates.
2. If the system is in an AND-state, it must be in all of its substates.

2.3.2 Conditions on the Transitions

A *step* means taking a set of transitions that satisfy the following conditions:

1. Any transitions that are followed must be enabled. A transition is enabled if the system is in its source state and its trigger is true.

2. Within an OR-state, only one transition can be followed.
3. Transitions may be followed within each substate of an AND-state.
4. If two or more transitions are enabled and have the same source, only one will be taken but it is indeterminate as to which will be chosen. (Section 2.2.3)

The following conditions are a result of decisions made in the previous sections:

1. Triggers are evaluated relative to the configuration at the beginning of the step. (There are no micro-steps so this is the only possibility.)
2. Taking no transitions is a legitimate step if no transitions are enabled. (Section 2.2.1
3. If a transition from a parent state is enabled, it has precedence over one from a descendant, where the priority is based on the source state of the transition. (Section 2.2.3)
4. Timeouts are determined relative to the last time the timeout event occurred throughout the system. (Section 2.2.4)

2.3.3 Conditions on the Results of Transitions

Once a set of transitions has been chosen to form a step in the system, the actions of these transitions are all considered together.

1. If a given transition is taken, at the end of the step the system will be in a configuration which includes the destination state of the transition and all its actions will be carried out except where conflicts occur among the actions of all transitions.
2. If a variable is not modified by any transition in a step, then it retains its previous value.
3. If more than one modification is made to the same variable (i.e. a conflict occurs) then exactly one of these modifications will be true in the next configuration. (Section 2.2.2)

Transitions should not cross AND-state boundaries (Section 2.2.5). If this is true, then the above conditions on which transitions can be taken in a step ensure that more than one chosen transition will not modify the same basic state.

2.4 Conclusions

Statecharts are a state transition notation that alleviates some of the problems encountered with other notations. In particular, the state explosion problem is lessened by using a hierarchy of states. This also reduces the complexity of the transitions in the model since some can be grouped together to form one transition exiting a higher-level state. Through the use of AND-states, statecharts can model concurrent operations.

Here we have focused on the difficulties in giving the semantics for what appears at first glance to be a clear, and concise graphical notation. This should not be viewed as an argument against the usefulness of statecharts or graphical notations in general. Rather it is intended to point out some interesting questions in modeling real-time systems and to stress the importance of having a formal semantics for these notations before using them.

State transition models are amenable to automatic analysis such as model checking. It is necessary to have a rigorous semantics for the notation to make this possible. The list of properties given in the preceding section describing the behaviour of statecharts will be formalized in the next three chapters.

Chapter 3

An Abstract Syntax for Statecharts

This chapter presents a syntax for statecharts. The statechart is represented as an abstract data type (ADT) whose attributes can be accessed by selector functions. Chapter 8 will discuss the concrete representation used in this work. The ADT is intended to hide these details in our presentation of the semantics.

3.1 Introduction

This chapter describes a representation of statecharts as an abstract data type (ADT) whose attributes can be accessed by selector functions. The next chapter will use this ADT to define a semantics for statecharts. The graphical representation used in STATEMATE may be viewed as an instantiation of this ADT. For our purposes, we use a textual representation which is described in Chapter 8. A tool for extracting the information about the statechart from STATEMATE and translating it into this textual representation is also described in Chapter 8.

This chapter gives a "bottom-up" presentation of our abstract syntax for statecharts, beginning with expressions, events, and actions. The meaning of these was given informally in the previous chapter and will be supplemented here. These parts of a transition can be accessed through selector functions. Each state has attributes associated with it which include a list of transitions. These attributes of a state, as well as information about the overall statechart, are also accessed through functions. The following sections will describes this ADT and its associated functions.

3.2 Variables and Values

The configuration of the system has been defined informally as the set of states the system is in, the values of the data-items and the status of events at a given moment. Every element of the configuration, including basic states[1], data-items (Boolean or arithmetic) and events, is given a variable name. These names are used to reference the values of variables.

Presently only Boolean and natural number values are considered, but this is not a limitation of the semantics. Given an underlying representation for values of other types and basic operations on expressions to deal with them, the semantic functions would not change. We have chosen only to implement these two types for simplicity.

3.3 Expressions

Expressions are used both in conditions of transition triggers and assignment actions. The syntax for expressions is given by the following recursive data type:

[1] We can determine the states the system is in completely from the basic states only.

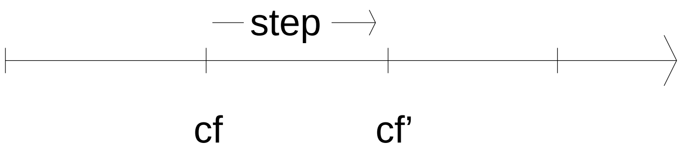

Figure 3.1: Events occurring in a step

```
Exp =
  VAR Variable |
  CONST Constant |
  IN Variable |
  PLUS Exp Exp |
  MULT Exp Exp |
  EQUAL Exp Exp |
  GREATER Exp Exp |
  OR Exp Exp |
  AND Exp Exp |
  NOT Exp |
  TRUE |
  FALSE
```

The expression VAR returns the value of a variable in the current configuration. CONST returns the value of its natural number argument. IN *statename* is the condition that the system is in the state called *statename*.

The remaining operators perform arithmetic and logical functions on their arguments. Some of these expressions operate on natural numbers and some on Boolean values. They have all been grouped together because it is expected that STATEMATE will have done the type-checking as the user builds the model, not allowing them to mix types in expressions. As far as the abstract syntax is concerned an expression such as NOT(CONST 21) is a legal expression which will be assigned a meaning by the semantics in Chapter 4.

The reader will also note the absence of the subtraction and division operations in expressions. These were left out because they were not needed in any of the examples, but they could easily be added.

3.4 Events

The status of an event is determined by changes between the previous configuration and the current one. It can be considered as a Boolean condition that has the value true if the event occurred in the previous step. The system must be able to determine its truth value relative to the current configuration only. For example in Figure 3.1, if event f occurs in the step moving the system from configuration cf to cf', then within cf' a flag representing the occurrence of event f will be true.

We can generalize the idea of events to include timeouts, which may depend on events that happened several steps earlier. A non-timeout event can be interpreted as a timeout with zero as the timeout step number. For example, $\mathbf{en}(x)$ is equivalent to $\mathbf{tm}(\mathbf{en}(x),\ \mathbf{0})$. In this way also, the idea of a flag can be generalized to be a counter that is maintained in the configuration as a variable and indicates how many steps have passed since an event occurred. When the counter has the value 0, the event occurred in the previous step. These counters are implicit in the graphical version of statecharts presented to a STATEMATE

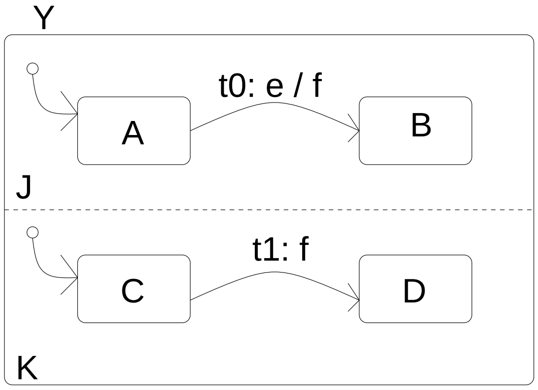

Figure 3.2: Example of a primitive event

user, however, we have made them explicit in our abstract syntax.

Timeouts can be expressed relative to *single events* only. Single events include:

- entering a state (EN *stname*),
- exiting a state (EX *stname*), and
- *primitive* events (EV *evn*) such as e in Figure 3.2. A Boolean variable is associated with each of these (evn) and the event occurs when the value of this variable changes.

The traffic light example of Figure 2.1 has many examples of using the event generated when a state is entered to trigger a transition.

Primitive events may be external or generated by an action. For example, in Figure 3.2, the event e is a primitive event that is external to the state **Y**. The primitive event f is generated by transition **t0**. If this transition is taken in the first step, then **t1** is enabled in the next step, providing the system is in state **C**. In this abstract syntax, event e as a trigger on a transition is written EV e. Generating f as an action is given by the statement GEN f.

A counter variable is created for each single event that is referenced in the statechart. This counter is given as an argument in the expression of each single event:

```
SingleEvent =
  EN Variable Counter |
  EX Variable Counter |
  EV Variable Counter
```

Events, in general, include single events (SE), and the combination of zero or more events:

```
Event =
  NONE |
  SE SingleEvent |
  EVEXPR Event Exp |
  AND_E Event Event |
  OR_E Event Event |
  NOT_E Event |
  TM SingleEvent Exp
```

The event NONE is always true. EVEXPR $ev\ c$ is true if the event ev occurs when the condition c is true. STATEMATE uses the syntax $ev[c]$ to represent this event. AND_E, OR_E, are logical connectors used to combine events. NOT_E ev is true if the event ev did not occur. TM $ev\ exp$ means the single event ev occurred exactly exp steps earlier.

3.5 Actions

Actions modify the configuration by changing the values of variables. Statecharts provide two special operations for changing the values of conditions, MAKE_TRUE and MAKE_FALSE, which make the value of their argument true or false respectively. These operate only on variables, not on compound condition expressions. The action GEN causes a primitive event to occur in this step. This means the variable for the primitive event changes its value. NILL means the transition's action does not make any changes to the system's configuration.

```
Action =
  MAKE_TRUE Variable |
  MAKE_FALSE Variable |
  ASN Variable Exp |
  BOTH Action Action |
  GEN Variable
  NILL
```

Note that the operator BOTH is not sequential and multiple assignments in an action are all evaluated relative to the configuration at the beginning of the step.

3.6 Transitions

Transitions have unique names and are made up of a source state, a destination state, a triggering event, and an action.[2] To return the transition associated with a particular name ($name$) in a statechart (sc), we use:

$$\mathsf{TRAN}\ sc\ name$$

[2] Section 9.1.5 discusses transitions with multiple source and/or destination states.

If the statechart has no transition with this name, it returns a transition that has null values for all its elements.

Given a particular transition (tr), the following selector functions are provided:

NAME (tr) – returns the name of tr
SRC (tr) – returns the name of the source state of tr
EVENT (tr) – returns the triggering event of tr
ACTION (tr) – returns the action on the label of tr
DEST (tr) – returns the name of the destination state of tr

The null values for each of these are NONAME, NOSTATE (used for both the source and destination state), NONE, NILL, respectively. Note that the null values for events and actions are legal values.

3.7 States

Every state has the type basic, AND, or OR.

$$\text{Typ} \equiv \text{B} \mid \text{A} \mid \text{O}$$

Given a particular state ($stname$), its type is returned by:

$$\mathsf{TYP}\, sc\ \ stname$$

The remaining elements associated with a state are its default state name, its immediate substates, and a set of transitions that originate at its immediate substates. Given a particular state ($stname$), these parts are accessed by the following functions:

SUBSTATES sc $stname$ – returns the names of the immediate substates of $stname$

DEFAULT sc $stname$ – returns the default state name of the state $stname$

TRANSOFSTATE sc $stname$ – returns the transitions whose source states are in the set of immediate substates of $stname$

The transitions are grouped according to their source's parent state since the priority of choosing transitions for execution is based on the position of their source states in the hierarchy.

Another useful function is one that returns the set of all transitions whose source states are descendants of a particular state. The function GET_TRANS_STATE sc $stname$ does this. It can be defined in terms the selector functions already described here.

It is assumed that STATEMATE produces a complete statechart with unique state names and therefore cases where these functions are given arguments that are not contained in the statechart should not happen. However, a defined null value (NOTYP, NOSUBSTATES, NODEFAULT, NOTRANS), is returned to make the functions total. Checks can be done from within STATEMATE to ensure the completeness of the model so the semantic functions do not usually test for the null values when using these functions. In some of these cases, no next configurations may exist for a given statechart configuration.

3.8 The Complete Statechart

A statechart is a set of states. Sometimes, we need information about the overall statechart.

It must have one state, called the root, that is an ancestor of all the other states. We can access it using:

$$\mathsf{ROOT}\, sc$$

It is also useful to return the set of names for all the transitions within the statechart:

$$\mathsf{GET\_TRANS\_NAMES}\ sc$$

In the next chapter, we will make use of these functions to give an operational semantics for statecharts. As much as possible these semantics are given compositionally, decomposing the parts of transition triggers and actions based on their syntactical components, however, because transitions can conflict, higher-level functions must resolve differences to determine the next configuration of the system.

Chapter 4

Compositional Aspects of the Semantics of Statecharts

This chapter begins the presentation of the formalization of our semantics for statecharts. We first describe useful functions that create quantifier abstractions and then give an abstract model of the configuration of the system. The meaning of events and actions are compositional and can be given in terms of functions defined recursively on their syntactical structure.

4.1 Introduction

Meaning is given to a statechart represented by the ADT described in the previous chapter through semantic functions that translate it into a relation over the current configuration and next configuration. Some aspects of these semantics are compositional and their meaning can be given in terms of functions defined recursively on their syntactical structure. This chapter examines the compositional aspects of the semantics as a preliminary step in our formalization. The next chapter will look at using these definitions to give the complete next configuration relation.

The first section presents definitions for quantifier abstractions which are used throughout the semantic definitions to deal with finite sets of elements like states or variables. The next section describes how the configuration is modeled, including data-items, events, event counters, and a representation for the states the system is currently in. In Sections 4.5, 4.7, and 4.9, the semantic functions for expressions, events, and actions are defined over the recursive data types given in the previous chapter. These definitions are used in two main functions:

ENABLED $sc\ tname\ cf$ **:** which is true if the transition $tname$ of statechart sc is enabled in cf (Section 4.8), and

RESULT $sc\ tname$ **:** which returns the list of modifications to the configuration given by the transition $tname$ of statechart sc (Sections 4.11).

These will be used in the next chapter to define the next configuration relation.

4.2 Quantifier Abstractions

As a preliminary step, we define a set of quantifier abstraction functions that express conditions over finite sets.

EVERY : returns the conjunction of applying a predicate p to all elements of a list x:

$$\mathsf{EVERY}\,p\,x =_{def} \quad (x = [\,]) \rightarrow \mathsf{T} \mid p\,(\mathsf{HD}\ x) \wedge \mathsf{EVERY}\,p\,(\mathsf{TL}\ \ x)$$

EXISTS : returns the disjunction of applying a predicate p to the elements of a list x:

$$\mathsf{EXISTS}\, p\, x =_{def} \quad (x = [\,]) \rightarrow \mathsf{F} \mid p\,(\mathsf{HD}\ x) \vee \mathsf{EXISTS}\, p\,(\mathsf{TL}\ \ x)$$

ALL_FALSE : returns the conjunction of the negation of the predicate p applied to each element in the list x:

$$\mathsf{ALL\_FALSE}\ p\ \ x =_{def} \quad (x = [\,]) \rightarrow \mathsf{T} \mid \neg(p\,(\mathsf{HD}\ x)) \wedge \mathsf{ALL\_FALSE}\ p\ \ (\mathsf{TL}\ \ x)$$

X_EXISTS : exactly one element of the list x returns true when p is applied to all elements in the list x:

$$\begin{array}{l} \mathsf{X\_EXISTS}\ p\ \ x =_{def} \\ \quad (x = [\,]) \rightarrow \mathsf{F} \mid \\ \quad (p\,(\mathsf{HD}\ x) \wedge \mathsf{ALL\_FALSE}\, p\,(\mathsf{TL}\ \ x)) \vee (\neg p\,(\mathsf{HD}\ x) \wedge \mathsf{X\_EXISTS}\, p\,(\mathsf{TL}\ \ x)) \end{array}$$

PAIR_EVERY : given two lists, returns the conjunction of applying the predicate p to pairs made from corresponding element of the lists:

$$\begin{array}{l} \mathsf{PAIR\_EVERY}\, p\ \ x1\ \ x2 =_{def} \\ \quad ((x1 = [\,]) \vee (x2 = [\,])) \rightarrow \mathsf{T} \mid p\,(\mathsf{HD}\ x1, \mathsf{HD}\ x2) \wedge \mathsf{EVERY}\, p\,(\mathsf{TL}\ \ x1)\ \ (\mathsf{TL}\ x2) \end{array}$$

ALT_X_EXISTS : given two predicates $p1$ and $p2$, returns the disjunction of the condition on each element n x that $p1$ is true for that element and $p2$ is true for all other elements in the list:

$$\begin{array}{l} \mathsf{ALT\_X\_EX}\ p1\ \ p2\ \ x =_{def} \\ \quad (x = [\,]) \rightarrow \mathsf{F} \mid \\ \quad (p1\,(\mathsf{HD}\ x) \wedge \mathsf{EVERY}\, p2\,(\mathsf{TL}\ \ x))\ \ \vee\ \ (p2\,(\mathsf{HD}\ x) \wedge \mathsf{ALT\_X\_EX}\ \ p1\ \ p2\ \ (\mathsf{TL}\ \ x)) \end{array}$$

It is also useful to existentially quantify over all the variables in a bit vector. The predicate EXISTSN $n\ p$ is used to state the condition that there exists some set of Booleans of size n that satisfy the predicate p. It existentially quantifies all the Boolean variables making up the bit vector:[1]

$$\begin{array}{l} \mathsf{EXISTSN\_AUX}\ (\mathsf{SUC}\ \ n)\ \ p\ \ list =_{def} \\ \quad (n = 0) \rightarrow p\ \ list \mid (\exists a.\, \mathsf{EXISTSN\_AUX}\ \ n\ \ p\ \ (\mathsf{CONS}\ \ a\ \ list)) \end{array}$$

$$\mathsf{EXISTSN}\ \ n\ \ p =_{def} \quad \mathsf{EXISTSN\_AUX}\ \ n\ \ p\ \ [\,]$$

4.3 An Abstract Model of the Configuration

The configuration of the system is represented by the values of a set of variables which include elements for the basic states, data-items, and events. As discussed in the last chapter these values can be Booleans or natural numbers. We can describe a configuration as a function mapping variables to values:

$$Config \equiv Variable \rightarrow Value$$

The variables cf and cf' will be used in functions to represent configurations. The function that accesses the value of any variable is called SemVAR and has the following definition:

$$\mathsf{SemVAR}\ \ v =_{def} \quad \lambda cf.\, cf\ \ v$$

We also need functions that operate on values. These semantics are parameterized by the underlying representation used for values through the following functions. Chapter 6 discusses the representation of values chosen for this work and the definitions of these functions.

[1] Over finite domains, existential quantification is equivalent to taking the disjunction of the predicate p applied to all the possible combinations of values for these variables.

NVAL : $Num \rightarrow Value$ - returns the value representation for the number

BVAL : $Bool \rightarrow Value$ - returns the value representation for the Boolean

BOOL : $Value \rightarrow Bool$ - returns the Boolean for the value representation

NPLUS : $Value \times Value \rightarrow Value$ - returns the representation for the result of adding the two values

NMULT : $Value \times Value \rightarrow Value$ - returns the representation for the result of multiplying the two values

NGREATER : $Value \times Value \rightarrow Value$ - returns the representation for the Boolean indicating if the first argument has a greater numeric value than the second one

BOR : $Value \times Value \rightarrow Value$ - returns the representation for taking the logical OR of the two values representing Booleans

BAND : $Value \times Value \rightarrow Value$ - returns the representation for taking the logical AND of the two values representing Booleans

BNOT : $Value \rightarrow Value$ - returns the representation for taking the logical NOT of the value representing a Boolean

BFALSE : $Value$ - returns the representation for False

BTRUE : $Value$ - returns the representation for True

EQVAL : $Value \times Value \rightarrow Value$ - returns the representation for the Boolean indicating if the first argument is equal to the second one

MAXVALUE : $Value \rightarrow Bool$ - test whether a value is equal to the maximum value that can be represented

4.4 Hierarchy of States

The set of variables includes not only the traditional idea of variables used in computation but also a record of the states the system is currently in since this is explicit in statecharts.

As previously described, statecharts use an AND/OR hierarchy of states for clarity in the graphical representation of the system. One Boolean variable represents each basic state to indicate whether or not the system is currently in that state. Higher-level states are given meaning through the values of the basic states:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{INSTATE}\ sc\ cf\ stname =_{def}} \\
(\mathsf{TYP}\ sc\ stname = \mathsf{B}) & \wedge & \mathsf{BOOL}(\mathsf{SemVAR}\ stname\ cf)\ \vee \\
(\mathsf{TYP}\ sc\ stname = \mathsf{A}) & \wedge & \mathsf{EVERY}\,(\mathsf{INSTATE}\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stname)\ \vee \\
(\mathsf{TYP}\ sc\ stname = \mathsf{O}) & \wedge & \mathsf{EXISTS}\,(\mathsf{INSTATE}\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stname)
\end{array}
$$

where *sc* is the statechart, and *stname* is the name of the state. Providing the system is currently in a legal state configuration, EXISTS is equivalent to X_EXISTS for the condition on substates of an OR state.

For example, if *sc* is the traffic light statechart in Figure 2.1, then INSTATE *sc cf* **N_S_G**[2] would partially evaluate to

$$\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf)$$

since it is a basic state. Because **NORMAL** is an AND-state with two substates and both of these substates are OR-states each with three basic substates, INSTATE *sc cf* **NORMAL** partially evaluates to:

$$
\begin{array}{l}
(\mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf) \vee \mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{N\_S\_Y}\ cf)\ \vee \\
\qquad \mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf))\ \wedge \\
(\mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{E\_W\_R}\ cf) \vee \mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{E\_W\_Y}\ cf)\ \vee \\
\qquad \mathsf{BOOL}\,(\mathsf{SemVAR}\ \mathbf{E\_W\_G}\ cf))
\end{array}
$$

[2] From now on we will use abbreviated names for the states in the traffic light. For example, **N_S_G** refers to **N_S.GREEN**.

Since all basic states have a truth value, when following a transition, it becomes necessary to explicitly say that the source basic states are exited (i.e. take on the value false) and the destination basic states are entered (i.e. take on the value true). Section 4.10 describes how to determine which basic states are affected for a transition.

4.5 Expressions

Expressions are used in the conditions and actions of transition labels. Their semantic functions take arguments of the type:

$$Expsem \equiv Config \rightarrow Value$$

and return an element of type $Expsem$. They make use of the operations on values in the following ways:
$\mathsf{SemCONST}\ n \ =_{def} \ \lambda cf.\, \mathsf{NVAL}\ n$
$\mathsf{SemPLUS}\,(a_1, a_2) \ =_{def} \ \lambda cf.\, \mathsf{NPLUS}\,(a_1\ cf,\ a_2\ cf)$
$\mathsf{SemMULT}\,(a_1, a_2) \ =_{def} \ \lambda cf.\ \mathsf{NMULT}\,(a_1\ cf, a_2\ cf)$
$\mathsf{SemGREATER}\,(a_1, a_2) \ =_{def} \ \lambda cf.\, \mathsf{NGREATER}\,(a_1\ cf, a_2\ cf)$
$\mathsf{SemEQUAL}\,(v_1, v_2) \ =_{def} \ \lambda cf.\, \mathsf{EQVAL}\,(v_1\ cf, v_2\ cf)$
$\mathsf{SemAND}\,(b_1, b_2) \ =_{def} \ \lambda cf.\, \mathsf{BAND}\,(b_1\ cf, b_2\ cf)$
$\mathsf{SemOR}\,(b_1, b_2) \ =_{def} \ \lambda cf.\, \mathsf{BOR}\,(b_1\ cf, b_2\ cf)$
$\mathsf{SemNOT}\ b \ =_{def} \ \lambda cf.\, \mathsf{BNOT}\,(b\ cf)$
$\mathsf{SemTRUE} =_{def} \ \lambda cf.\, \mathsf{BTRUE}$
$\mathsf{SemFALSE} =_{def} \ \lambda cf.\, \mathsf{BFALSE}$

Putting these all together, we can define the following function of the recursive data type Exp:

$$\begin{aligned}
\mathsf{SemExp}\,(\mathsf{VAR}\ v)\ sc &=_{def} \mathsf{SemVAR}\ v \\
\mathsf{SemExp}\,(\mathsf{CONST}\ n)\ sc &=_{def} \mathsf{SemCONST}\ n \\
\mathsf{SemExp}\,(\mathsf{IN}\ stname)\ sc &=_{def} \lambda cf.\, \mathsf{BVAL}\,(\mathsf{INSTATE}\ sc\ cf\ stname) \\
\mathsf{SemExp}\,(\mathsf{PLUS}\ a_1\ a_2)\ sc &=_{def} \mathsf{SemPLUS}\,(\mathsf{SemExp}\ a_1\ sc, \mathsf{SemExp}\ a_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{MULT}\ a_1\ a_2)\ sc &=_{def} \mathsf{SemMULT}\,(\mathsf{SemExp}\ a_1\ sc, \mathsf{SemExp}\ a_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{EQUAL}\ a_1\ a_2)\ sc &=_{def} \mathsf{SemEQUAL}\,(\mathsf{SemExp}\ a_1\ sc, \mathsf{SemExp}\ a_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{GREATER}\ a_1\ a_2)\ sc &=_{def} \mathsf{SemGREATER}\,(\mathsf{SemExp}\ a_1\ sc, \mathsf{SemExp}\ a_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{OR}\ b_1\ b_2)\ sc &=_{def} \mathsf{SemOR}\ (\mathsf{SemExp}\ b_1\ sc,\ \mathsf{SemExp}\ b_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{AND}\ b_1\ b_2)\ sc &=_{def} \mathsf{SemAND}\,(\mathsf{SemExp}\ b_1\ sc,\ \mathsf{SemExp}\ b_2\ sc) \\
\mathsf{SemExp}\,(\mathsf{NOT}\ b)\ sc &=_{def} \mathsf{SemNOT}\,(\mathsf{SemExp}\ b\ sc) \\
\mathsf{SemExp}\,(\mathsf{TRUE})\ sc &=_{def} \mathsf{SemTRUE} \\
\mathsf{SemExp}\,(\mathsf{FALSE})\ sc &=_{def} \mathsf{SemFALSE}
\end{aligned}$$

SemExp is higher-order because it returns a function giving the denotation of the expression, which can be evaluated relative to a particular configuration. This style of function has been used previously to give the semantics of a small imperative language where all of its elements are compositional [18].

4.6 Conditions

To evaluate a condition in a transition label, we assume the expression represents a Boolean value:
$\mathsf{SemCondition}\ cond\ sc \ =_{def} \ \lambda\, cf.\, \mathsf{BOOL}\ (\mathsf{SemExp}\ cond\ sc\ cf)$
Given a configuration, this predicate is true when the expression evaluates to the Boolean value true.

4.7 Events

The meaning of an event expression is true if the event occurred in the previous step. The event may involve examining the values of variables in configurations earlier than the current one to determine if values have changed in a particular step. Except for timeouts, this is relative to the previous configuration only; for timeouts, this can involve checking several steps earlier. To minimize the number of values used in the overall expression, we must determine if a given event occurred by examining the current configuration only.

A counter is maintained for each event which is reset to zero when the event occurs and otherwise is incremented in each time step [23]. When the counter is evaluated in the current configuration, it gives a measure of how many steps occurred since the event last happened. In the translation process, the statechart is examined to determine which events trigger transitions and a counter is created for each of these. To determine if an event occurred in the previous step, we check to see if the counter is zero in the current step; for a timeout, we check if the counter is equal to the timeout step number. This allows us to interpret events as Boolean expressions evaluated relative to the current configuration only.

Since each counter has a maximum value, we must ensure that it does not falsely indicate that the event occurred when it overflows. This is done by incrementing it only up to its maximum value. This maximum value, say max, can never be used to indicate an event occurring, since it really means the event happened max or more steps ago. This means that if the timeout step number of a particular timeout is n then n must be less than max.

A transition is enabled if its trigger, consisting of events, is true in the current configuration. The semantic functions for events have the type:

$$Evsem \equiv Config \rightarrow Bool$$

so they can be evaluated in the current configuration to determine if the event occurred in the previous step.

Single events and timeouts are evaluated by examining their counters. The semantic function for timeouts checks whether the denotation of the counter expression ($counter$) equals the denotation of the expression (exp) for the timeout step number when both are evaluated in the current configuration:

$$\begin{array}{l} \mathsf{SemTM}\ (counter, exp)\ =_{def} \\ \quad \lambda cf. \neg \mathsf{MAXVALUE}\,(counter\ \ cf)\ \wedge\ \mathsf{BOOL}\,(\mathsf{EQVAL}\,(counter\ \ cf, exp\ \ cf)) \end{array}$$

The timeout step number is always evaluated in the current configuration resolving the questions raised in Sect.2.2.4.

As mentioned in the previous chapter, we distinguish between single and compound events so that timeouts can only be based on single events. Single events include primitive events, and entering and exiting states, while compound events include timeouts, and events joined by logical connectives. This way a counter for each single event is sufficient to evaluate any timeout. Keeping a separate counter for each timeout (even if two timeouts are based on the same event) is a more general method but was judged to be unnecessary for this research.

The meaning of single events is the value of their counters. This is defined over the recursive data type SingleEvent as:

$$\begin{array}{rcl} \mathsf{SemSingleEvent}\ (\mathsf{EX}\ stname\ \ counter) & =_{def} & \mathsf{SemVAR}\ counter \\ \mathsf{SemSingleEvent}\ (\mathsf{EV}\ evn\ \ counter) & =_{def} & \mathsf{SemVAR}\ counter \\ \mathsf{SemSingleEvent}\ (\mathsf{EN}\ stname\ \ counter) & =_{def} & \mathsf{SemVAR}\ counter \end{array}$$

The function SemSingleEvent gives the denotation of a single event as a function mapping a configuration onto the value of the event's counter in that configuration. Note that the name of the primitive event, or the name of the state entered or exited is not used at all to determine the truth value of the event.

Compound events can be created using the operators AND_E, OR_E, and NOT_E. These take on their intuitive meanings:

$$\begin{array}{rcl} \mathsf{SemAND\_E}\ (ev_1, ev_2) & =_{def} & \lambda cf.\ (ev_1\ cf)\ \wedge\ (ev_2\ cf) \\ \mathsf{SemOR\_E}\ (ev_1, ev_2) & =_{def} & \lambda cf.\ (ev_1\ cf)\ \vee\ (ev_2\ cf) \\ \mathsf{SemNOT\_E}\ ev & =_{def} & \lambda cf. \neg\,(ev\ cf) \end{array}$$

Note that these functions are not equivalent to the functions that evaluate the meaning of expressions since SemAND_E, SemOR_E, and SemNOT_E operate on the logical values of events while SemAND, SemOR, and SemNOT operate on configuration values.

The complete expression for events can be given as:

$$
\begin{aligned}
&\mathsf{SemEvent}\ (\mathsf{NONE})\ \ sc =_{def} \lambda cf.\,\mathsf{T}\\
&\mathsf{SemEvent}\ (\mathsf{SE}\ se)\ \ sc =_{def}\ \mathsf{SemTM}\ (\mathsf{SemSingleEvent}\ se,\ \ \mathsf{SemCONST}\ 0)\\
&\mathsf{SemEvent}\ (\mathsf{EVEXPR}\ ev\ cond)\ \ sc =_{def}\\
&\quad \lambda cf.\ \mathsf{SemEvent}\ ev\ sc\ cf\ \wedge\ \ \mathsf{SemCondition}\ cond\ sc\ cf\\
&\mathsf{SemEvent}\ (\mathsf{TM}\ se\ exp)\ \ sc =_{def}\ \mathsf{SemTM}\ (\mathsf{SemSingleEvent}\ se,\ \mathsf{SemExp}\ exp\ sc)\\
&\mathsf{SemEvent}\ (\mathsf{AND\_E}\ ev_1\ ev_2)\ \ sc =_{def}\\
&\quad \mathsf{SemAND\_E}\ (\mathsf{SemEvent}\ ev_1\ sc,\ \ \mathsf{SemEvent}\ ev_2\ sc)\\
&\mathsf{SemEvent}\ (\mathsf{OR\_E}\ \ ev_1\ ev_2)\ \ sc =_{def}\\
&\quad \mathsf{SemOR\_E}\ (\mathsf{SemEvent}\ ev_1\ sc,\ \ \mathsf{SemEvent}\ ev_2\ sc)\\
&\mathsf{SemEvent}\ (\mathsf{NOT\_E}\ ev)\ \ sc =_{def}\ \mathsf{SemNOT\_E}\ (\mathsf{SemEvent}\ ev\ sc)
\end{aligned}
$$

4.8 Enabling a Transition (ENABLED)

Using the semantics for events and what it means to be in a state, we can define when a transition is enabled:

$$
\begin{aligned}
&\mathsf{ENABLED}\ sc\ tname\ cf\ \ =_{def}\\
&\quad \mathbf{let}\ \ tr = \mathsf{TRAN}\ \ sc\ tname\ \mathbf{in}\\
&\quad \mathsf{INSTATE}\ sc\ cf\ (\mathsf{SRC}\ tr)\ \wedge\ \mathsf{SemEvent}\ (\mathsf{EVENT}\ tr)\ sc\ cf
\end{aligned}
$$

where *tname* is the name of a transition. The trigger is evaluated relative to the configuration at the beginning of the step.

4.9 Actions

The transitions are labeled with actions which modify the data items in the system. These actions can all be defined in terms of an assignment statement. The semantic function for an assignment statement returns a pair, (v, exp), which indicates that the expression exp, evaluated in the current configuration, should be assigned to the variable v in the next configuration.

$$\mathsf{SemASN}\ v\ exp\ \ =_{def}\ \ [(v, exp)]$$

A SKIP statement makes no changes to the system:

$$\mathsf{SemSKIP} =_{def}\ \ [\,]$$

The BOTH action returns the set made up of the changes from its constituent actions:

$$\mathsf{SemBOTH}\ a_1\ a_2 =_{def}\ \ \mathsf{APPEND}\ a_1\ a_2$$

MAKE_FALSE and MAKE_TRUE assign the value false or true to their variable argument. The statement GEN generates an internal primitive event which means its variable takes on the opposite value than it had previously to create the occurrence of the event in this time step.

The meaning of the actions can be grouped together and recursively defined over the data type Action as:

$$
\begin{aligned}
&\mathsf{SemAction}\ (\mathsf{MAKE\_TRUE}\ v)\ \ sc\ \ =_{def}\ \mathsf{SemASN}\ v\ (\mathsf{SemTRUE})\\
&\mathsf{SemAction}\ (\mathsf{MAKE\_FALSE}\ v)\ \ sc\ \ =_{def}\ \mathsf{SemASN}\ v\ (\mathsf{SemFALSE})
\end{aligned}
$$

$$\mathsf{SemAction}\ (\mathsf{ASN}\ v\ e)\ \ sc\ =_{def}\ \mathsf{SemASN}\ v\ (\mathsf{SemExp}\ e\ sc)$$
$$\mathsf{SemAction}\ (\mathsf{BOTH}\ a_1\ a_2)\ \ sc\ =_{def}$$
$$\quad \mathsf{SemBOTH}\ (\mathsf{SemAction}\ a_1\ sc)\ (\mathsf{SemAction}\ a_2\ sc)$$
$$\mathsf{SemAction}\ (\mathsf{GEN}\ ev)\ \ sc\ =_{def}\ \mathsf{SemASN}\ ev\ (\mathsf{SemNOT}(\mathsf{SemVAR}\ ev))$$
$$\mathsf{SemAction}\ (\mathsf{NILL})\ \ sc\ =_{def}\ \mathsf{SemSKIP}$$

SemAction returns a list of pairs of variables and denotations of expressions.

4.10 Source and Destination State

The result of executing a transition modifies the configuration not only by the actions but also by leaving the source state and entering the destination state.

The variables for the basic states of the source state must be set to false and the ones for the destination should be set to true as the defaults allow. If the destination modifications overlap with the ones for the source (for example if a transition loops), then the changes for the destination take precedence.

To enter the destination state, it may be necessary to follow default entrances. Given a destination state name (*stname*), the set of modifications to the basic states are given by:

$$\begin{array}{l}
\mathsf{ENTERDEST}\ sc\ stname =_{def} \\
\quad (\mathsf{TYP}\ sc\ stname = \mathsf{B}) \rightarrow [(stname, \lambda cf.\,\mathsf{BTRUE})]\ | \\
\quad ((\mathsf{TYP}\ sc\ stname = \mathsf{A}) \rightarrow \\
\qquad\qquad \mathsf{FLAT}\ (\mathsf{MAP}\ (\mathsf{ENTERDEST}\ sc)\ (\mathsf{SUBSTATES}\ sc\ stname))\ | \\
\quad \mathsf{ENTERDEST}\ sc\ (\mathsf{DEFAULT}\ sc\ stname))
\end{array}$$

For example, in the traffic light of Figure 2.1 (*tls*), the transition $t7$ goes from the state **FL** (flashing) to the state **NORMAL**. This destination state is an AND-state whose components are further decomposed into OR-states. Following the defaults, ENTERDEST *tls* **NORMAL** returns the list:

$$[(\mathbf{E\_W\_R}, \lambda cf.\,\mathsf{BTRUE}); (\mathbf{N\_S\_G}, \lambda cf.\,\mathsf{BTRUE})]$$

The modifications to exit the source state (*stname*) are also determined by traversing through the hierarchy, but in this case, before returning a (variable, expression) pair, the list of modifications for entering the destination are checked so that conflicting assignments are not returned. This list is given by the parameter *mods*:

$$\begin{array}{l}
\mathsf{EXITSRC}\ mods\ sc\ stname =_{def} \\
\quad ((\mathsf{TYP}\ sc\ stname = \mathsf{B}) \wedge \neg \mathsf{MEMBER}\ stname\ mods) \rightarrow [(stname, \lambda cf.\,\mathsf{BFALSE})]\ | \\
\quad \mathsf{FLAT}\ (\mathsf{MAP}\ (\mathsf{EXITSRC}\ mods\ sc)\ (\mathsf{SUBSTATES}\ sc\ stname))
\end{array}$$

Some of these basic states will already have the value false so in effect they keep their previous values.

For transition **t6** which leaves the state **NORMAL**, EXITSRC returns the list of pairs to set all the basic states in **NORMAL** to false:

$$\begin{array}{l}
[(\mathbf{E_W_R}, \lambda cf.\,\mathsf{BFALSE}); (\mathbf{E_W_Y}, \lambda cf.\,\mathsf{BFALSE}); (\mathbf{E_W_G}, \lambda cf.\,\mathsf{BFALSE}); \\
(\mathbf{N_S_R}, \lambda cf.\,\mathsf{BFALSE}); (\mathbf{N_S_Y}, \lambda cf.\,\mathsf{BFALSE}); (\mathbf{N_S_G}, \lambda cf.\,\mathsf{BFALSE})]
\end{array}$$

4.11 Executing a Transition (RESULT)

The complete semantics for the result of one transition combines the functions for entering and exiting the source and destination and the actions:

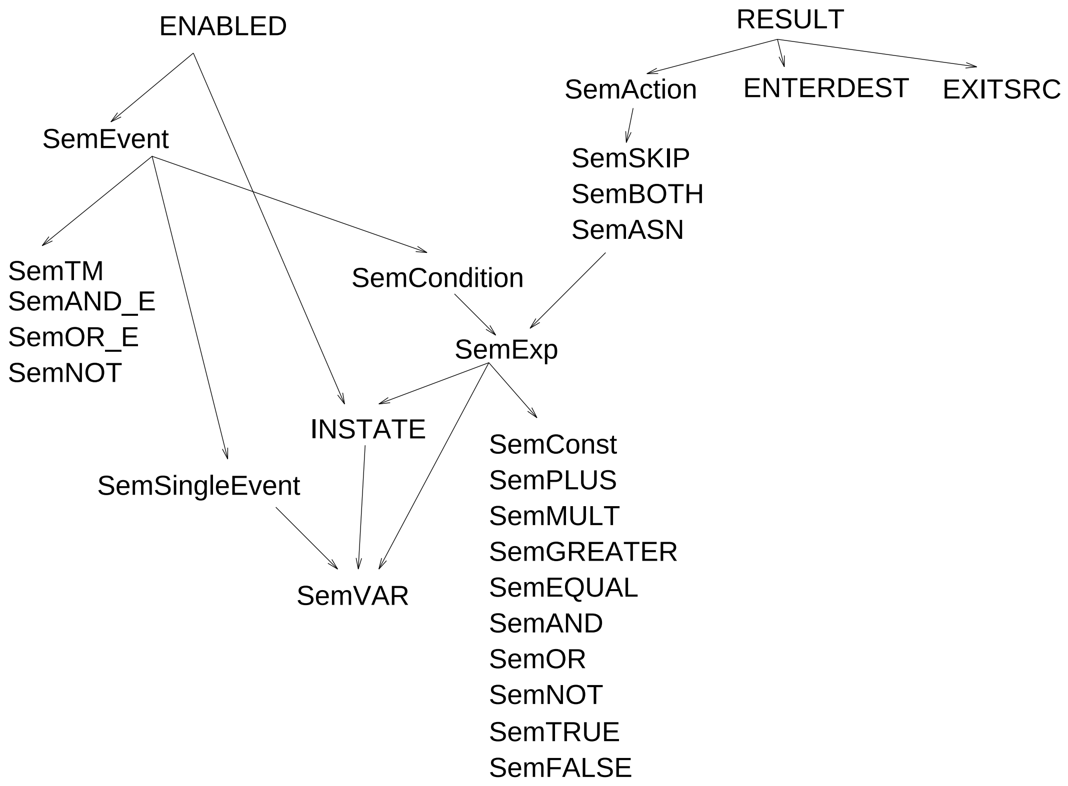

Figure 4.1: Relationship among compositional definitions

$\mathsf{RESULT}\ sc\ tname\ =_{def}$
$\quad$ **let** $tr = \mathsf{TRAN}\ sc\ tname$ **in**
$\quad$ **let** $mods = \mathsf{ENTERDEST}\ sc\ (\mathsf{DEST}\ tr)$ **in**
$\quad$ $\mathsf{APPEND}\ mods\ (\mathsf{APPEND}\ (\mathsf{EXITSRC}\ (\mathsf{MAP\ FST}\ mods)\ sc\ (\mathsf{SRC}\ tr))$
$\qquad (\mathsf{SemAction}\ (\mathsf{ACTION}\ tr)\ sc))$

4.12 Summary

This chapter has presented the compositional aspects of the semantics of statecharts. Figure 4.1 shows the "uses" diagram for the primary functions defined in this chapter; each arrow may be read as "is defined in term of".

The definitions ENABLED, RESULT, INSTATE and SemVAR will be used in the next chapter.

Chapter 5

A Semantics for Statecharts

This chapter presents the formalization of an operational semantics for statecharts as a next configuration predicate that holds true if one system configuration is a successor to another for a given statechart. The compositional aspects of the semantics were given in the previous chapter and will be used here. The difficulty comes in determining both which transitions can be taken and what the combined result is of following a set of transitions. Particular attention is given to issues such as what is a step, race conditions, and multiple actions associated with one transition. The parts of the semantics are illustrated using the traffic light example.

5.1 Introduction

The previous chapter described the compositional aspects of the meaning of a statechart. These functions are used by other definitions to create a relation over the current configuration and next configuration of a given statechart. This relation, called NC, has the form:

$$\mathsf{NC}\ \ sc\ \ varlist\ \ cf\ \ cf'$$

where,
sc is the statechart,
$varlist$ is the set of internal variables,
cf is the current configuration, and
cf' is the next configuration.

NC is true if cf' is a configuration that the system, described by the statechart sc, could be in one step after being in cf. The statechart controls the variables in the list $varlist$. All other variables could possibly change their values in a step. Because of non-determinism, there may be multiple next configurations that satisfy the relation for a given current configuration.

The difficulty in giving the semantics for statecharts comes in expressing both which transitions can be taken and what the combined result is of following a set of transitions. The set of transitions that can be taken is limited by which ones are enabled, the hierarchy of the statechart and the priority within that hierarchy. The result of following a transition may depend on whether other transitions modify the same variables. It is also necessary to express the condition that if an internal variable is not modified in a given step then it retains its previous value.

NC is defined using four semantic functions that formalize the properties of statecharts stated in Chapter 2:

$$
\begin{array}{ll}
\mathsf{NC}\ \ sc\ \ varlist\ \ cf\ \ cf' =_{def} & \\
\quad \mathbf{let}\ \ tnames = \mathsf{GET\_TRANS\_NAMES}\ sc\ \ \ \mathbf{in} & \\
\quad \mathsf{EXISTSN}\ \ (\mathsf{LENGTH}\ tnames)\ (\lambda tflags. & \\
\qquad \mathsf{TRANS\_COND}\ cf\ sc\ tflags\ \ \wedge & (5.1)\\
\qquad \mathsf{STATE\_COND}\ sc\ cf'\ \ \wedge & (5.2)\\
\qquad \mathsf{VAR\_COND}\ sc\ cf\ cf'\ tflags\ varlist\ \ \wedge & (5.3)\\
\qquad \mathsf{EVENT\_COND}\ sc\ cf\ cf') & (5.4)
\end{array}
$$

Informally, this relation checks whether there is any way (EXISTSN) of choosing a set of transitions to follow that will move the system from cf to cf' in one step. The sections in this chapter define the following parts of the next configuration relation:

TRANS_COND: conditions on the set of transitions that can be taken, including hierarchy, priority and that the chosen transitions are enabled (Section 5.2)

STATE_COND: ensuring that the next state configuration is legal (Section 5.3)

VAR_COND: conditions on all the variables in the next configuration (i.e. whether they are modified by a transition that is taken or keep their previous values) (Section 5.4)

EVENT_COND: determining if events occur in a step to update the event counters mentioned in the previous chapter (Section 5.5)

The functions given in the previous chapter return denotations for transition triggers (ENABLED) and expressions used in assignment statements (RESULT). These are evaluated relative to the current configuration. Assignments are made to variables in the next configuration.

The validity of these semantics depends on our interpretation of the operation of statecharts, and in the correctness of expressing this interpretation in the target language. They have been informally checked using a mechanical proof-assistant to reduce the semantic functions to Boolean expressions over the variables for particular problems. They have also been executed in the model checker described in the next chapter. Through this process, errors were discovered and fixed, and we have increased confidence in the result.

5.2 Transition Condition (TRANS_COND)

A *step* was defined previously as following zero or more transitions that satisfy a number of conditions. In this section, we will formalize the meaning of those conditions.

Each transition is represented by a Boolean flag indicating if the transition is taken in this step. If a vector of transition flags satisfies the transition condition then following these transitions is a step. Because statecharts can describe non-deterministic operation, there may be several possible sets. The purpose of the transition condition is to ensure that the set chosen satisfies the various relationships that must hold among transitions:

1. Any transitions that are followed must be enabled. A transition is enabled if the system is in its source state and its trigger is true.

2. Within an OR-state, only one transition can be followed.

3. Transitions may be followed within each substate of an AND-state.

4. If two or more transitions are enabled and have the same source, only one will be taken but it is indeterminate as to which will be chosen.

5. Triggers are evaluated relative to the configuration at the beginning of the step.

6. Taking no transitions is a legitimate step if no transitions are enabled, however, there may still be changes in the configuration such as updating event counters.

7. If a transition from a parent state is enabled, it has precedence over one from a descendant, where the priority is based on the source state of the transition.

8. Timeouts are determined relative to the last time the timeout event occurred throughout the system.

Providing the system is currently in a legal state configuration, the next configuration will be legal if the set of transitions taken satisfies these conditions. A step does not include transitions triggered by events occurring in this step and therefore only transitions out of the set of states at the beginning of the step are considered. These issued were discussed in Section 2.2.1.

To formulate these conditions on transitions, we can first write a predicate that determines if the set of transitions among the descendants of a given state (*stname*) satisfies these conditions. This predicate depends on transitions chosen throughout its descendant states. Using *tnames* as the set of transition names of statechart *sc*, and *tflags* as the bit vector of flags for the transitions, the complete definition for this predicate is:

$$
\begin{array}{l}
\mathsf{TRANS\_COND\_AUX}\ cf\ sc\ tnames\ tflags\ stname =_{def} \\
(\mathsf{TYP}\ sc\ stname = \mathsf{B}) \rightarrow \mathsf{T}\ | \\
(\mathsf{TYP}\ sc\ stname = \mathsf{A}) \rightarrow \\
\quad \mathsf{EVERY}\ (\mathsf{TRANS\_COND\_AUX}\ cf\ sc\ tnames\ tflags)\ (\mathsf{SUBSTATES}\ sc\ stname)\ \ | \\
\mathbf{let}\ here\ =\ \mathsf{MAP\ NAME}\ (\mathsf{TRANSOFSTATE}\ sc\ stname)\ \mathbf{and} \\
\quad below\ = \mathsf{FLAT}\ (\mathsf{MAP}\ (\mathsf{GET\_TRANS\_NAMES}\ sc)\ (\mathsf{SUBSTATES}\ sc\ stname))\ \mathbf{in} \\
\mathbf{let}\ prioritytest = \mathsf{EXISTSN}\ (\mathsf{LENGTH}\ here)\ (\mathsf{ONE\_LEVEL}\ cf\ sc\ here)\ \mathbf{in} \\
\quad (prioritytest \wedge \\
\qquad \mathsf{ONE\_LEVEL}\ cf\ sc\ here\ (\mathsf{MAP}\ (\mathsf{GET\_TR\_FLAG}\ tnames\ tflags)\ here) \wedge \\
\qquad \mathsf{ALL\_FALSE}(\mathsf{GET\_TR\_FLAG}\ tnames\ tflags)\ below) \quad \vee \\
\quad (\neg prioritytest\ \ \wedge \\
\qquad \mathsf{ALL\_FALSE}(\mathsf{GET\_TR\_FLAG}\ tnames\ tflags)\ here \wedge \\
\qquad \mathsf{EVERY}\ (\mathsf{TRANS\_COND\_AUX}\ cf\ sc\ tnames\ tflags)\ (\mathsf{SUBSTATES}\ sc)\ stname)
\end{array}
$$

The above definition can be explained by considering the three main parts corresponding to the three types of states. If *stname* is a basic state ($\mathsf{TYP}\ sc\ stname = \mathsf{B}$), then there are no transitions below it and therefore no limits are specified on any transitions in the statechart for this predicate to hold true for this state.

For AND-states ($\mathsf{TYP}\ sc\ stname = \mathsf{A}$), there are no transitions to consider at this level but it must ensure that $\mathsf{TRANS\_COND\_AUX}$ holds for all substates (orthogonal components) of the AND-state.

The last case is if the state is an OR-state which may have transitions among its substates. These names of these transitions can be determined by:

$$\mathbf{let}\ here\ =\ \mathsf{MAP\ NAME}\ (\mathsf{TRANSOFSTATE}\ sc\ stname)$$

The system can take exactly one of these transitions providing it is enabled. Using the parameter *flags* to represent a possible set of flags for the transitions in *here*, this exclusive-OR condition can be stated as:

$$
\begin{array}{l}
\mathsf{ONE\_LEVEL}\ cf\ sc\ here\ flags =_{def} \\
\quad \mathsf{X\_EXISTS}\ (\lambda y.\, y)\ flags\quad \wedge \\
\quad \mathsf{PAIR\_EVERY}\ (\lambda(flag, tname).\, flag \Longrightarrow \mathsf{ENABLED}\ sc\ tname\ cf)\ flags\ tnames
\end{array}
$$

This predicate says that exactly one transition flag from the set given in *flags* must be set to true and if it is true then it must be enabled.

The priority of transitions is given by checking if there is any set of flags for transitions given by *here* that satisfy the function $\mathsf{ONE\_LEVEL}$:

$$\mathbf{let}\ prioritytest = \mathsf{EXISTSN}\ (\mathsf{LENGTH}\ here)\ (\mathsf{ONE\_LEVEL}\ cf\ sc\ here)$$

If the priority test is satisfied then the flags for the transitions in *here* are set by ONE_LEVEL and all flags for transitions originating within this state's substates should be set to false (ALL_FALSE). The labels for transitions originating within the substates of a state are determined by:

$$\textbf{let } \mathit{below} = \mathsf{FLAT}\,(\mathsf{MAP}\,(\mathsf{GET\_TRANS\_STATE}\; sc)\,(\mathsf{SUBSTATES}\; sc\; stname))$$

When the priority test at this level is false, the flags for transitions at this level are set to false, and EVERY substate of the OR-state is examined. Assuming that the system is already in a legal configuration, it should only be in one substate of an OR-state, therefore no transitions will be enabled in any of the other substates and their transition conditions will reduce to true.

This completes the explanation of the parts of TRANS_COND_AUX. Note that if the priority test is never satisfied then all the basic states will be considered. This means all the transition flags for transitions originating at descendant states of *stname* will be set to false. This is still a step as given in the sixth property at the beginning of this section.

5.2.1 Evaluating the Transition Condition

To illustrate the meaning of the auxiliary transition condition, we can look at the partial evaluation of it for the state **E_W**. Within this state, the transitions **t3**, **t4**, and **t5** could possibly be taken. The flags $x3$, $x4$, and $x5$ are used in the priority test:

$$
\begin{array}{l}
\textbf{let } \mathit{prioritytest} \;=\; \exists x3\; x4\; x5. \\
\quad ((x3 \land \neg x4 \land \neg x5) \lor (\neg x3 \land x4 \land \neg x5) \lor (\neg x3 \land \neg x4 \land x5)) \quad \land \\
\quad (x3 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_G} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{TM}(\mathrm{EN}(\mathbf{E\_W\_G}, \mathbf{EN\_E\_W\_G}),\; \mathbf{EW\_G\_T}))\; sc\; cf)) \quad \land \\
\quad (x4 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_Y} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{TM}(\mathrm{EN}(\mathbf{E\_W\_Y}, \mathbf{EN\_E\_W\_Y}), \mathrm{CONST}\; 2))\; sc\; cf)) \quad \land \\
\quad (x5 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_R} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{IN}\; \mathbf{N\_S\_R})\; sc\; cf))
\end{array}
$$

If the priority test holds then the transition flags are set appropriately for this level, otherwise these flags are set to false. Since the substates of E_W are all basic states, the transition condition reduces to true for each of them. Therefore, TRANS_COND_AUX partially evaluates to:

$$
\begin{array}{l}
(\mathit{prioritytest} \land \\
((t3 \land \neg t4 \land \neg t5) \lor (\neg t3 \land t4 \land \neg t5) \lor (\neg t3 \land \neg t4 \land t5)) \quad \land \\
(t3 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_G} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{TM}(\mathrm{EN}(\mathbf{E\_W\_G}, \mathbf{EN\_E\_W\_G}),\; \mathbf{EW\_G\_T}))\; sc\; cf)) \quad \land \\
(t4 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_Y} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{TM}(\mathrm{EN}(\mathbf{E\_W\_Y}, \mathbf{EN\_E\_W\_Y}), \mathrm{CONST}\; 2))\; sc\; cf)) \quad \land \\
(t5 \Longrightarrow (\mathsf{INSTATE}\; sc\; cf\; \mathbf{E\_W\_R} \land \\
\qquad\qquad \mathsf{SemEvent}\,(\mathrm{IN}\; \mathbf{N\_S\_R})\; sc\; cf)) \\
\lor \\
(\neg \mathit{prioritytest} \land (\neg t3 \land \neg t4 \land \neg t5))
\end{array}
$$

If the system is currently in the **E_W_R** state (the source of transition **t5**) and the **N_S_R** state in the other component of **NORMAL** then transition **t5** will be enabled. Making $x5$ true and $x3$ and $x4$ false will satisfy the priority test. There are no transitions from states below these and this part of the condition does not limit any of the other transition flags. Therefore, a bit vector for the transition flags that will satisfy the above condition is:

t0	t1	t2	t3	t4	t5	t6	t7
.	.	.	F	F	T	.	.

where "." represents a "don't care" value.

Using the same configuration cf, the following assignment of values to the transition flags:

t0	t1	t2	t3	t4	t5	t6	t7
.	.	.	T	F	T	.	.

would make TRANS_COND_AUX false since the exclusive-OR of taking transitions within a state is not satisfied.

5.2.2 The Complete Transition Condition

For the complete statechart, the transition condition has to hold starting from the root state:

$$\begin{aligned}&\mathsf{TRANS\_COND}\ cf\ \ sc\ tflags\ \ =_{def}\\&\quad \mathbf{let}\ tnames = \mathsf{GET\_TRANS\_NAMES}\ sc\ \ \mathbf{in}\\&\quad \mathsf{TRANS\_COND\_AUX}\ \ cf\ \ sc\ \ tnames\ \ tflags\ (\mathsf{ROOT}\ sc)\end{aligned} \tag{5.1}$$

5.3 State Condition (STATE_COND)

The system is in a legal state configuration, if it satisfies the following two conditions:

1. If the system is in an OR-state, it must be in exactly one of the OR-state's substates.

2. If the system is in an AND-state, it must be in all of its substates.

If the system tries to follow a set of transitions that result in a non-legal state configuration then these conditions will not hold. For some statecharts, there maybe not be any possible next configurations, because TRANS_COND can only be satisfied by a set of transitions that lead to states that violate the state conditions.

Starting at a particular state in the hierarchy ($stname$) that the system is currently in: if it is an AND-state, then we have to be in all of its components; if it is an OR-state, we must be in exactly one of its substates. For the substates that the system is not in then all descendent basic states must have the value false.

$$\begin{aligned}&\mathsf{ALL\_F\_STATES}\ sc\ \ cf\ \ stname =_{def}\\&\quad (\mathsf{TYP}\ sc\ \ stname = \mathsf{B}) \rightarrow \neg\mathsf{BOOL}\ (\mathsf{SemVAR}\ stname\ \ cf)\ \ |\\&\quad \mathsf{EVERY}\ (\mathsf{ALL\_F\_STATES}\ sc\ \ cf)\ (\mathsf{SUBSTATES}\ sc\ \ stname)\end{aligned}$$

$$\begin{aligned}&\mathsf{STATE\_COND\_AUX}\ sc\ \ cf\ \ stname =_{def}\\&\quad (\mathsf{TYP}\ sc\ \ stname = \mathsf{B}) \rightarrow \mathsf{BOOL}\ (\mathsf{SemVAR}\ stname\ \ cf)\ \ |\\&\quad ((\mathsf{TYP}\ sc\ \ stname = \mathsf{A}) \rightarrow\\&\qquad\qquad \mathsf{EVERY}\ (\mathsf{STATE\_COND\_AUX}\ sc\ \ cf)\ (\mathsf{SUBSTATES}\ sc\ \ stname)\ |\\&\quad \mathsf{ALT\_X\_EX}\ (\mathsf{STATE\_COND\_AUX}\ sc\ \ cf)\ (\mathsf{ALL\_F\_STATES}\ sc\ \ cf)\\&\qquad\qquad (\mathsf{SUBSTATES}\ sc\ \ stname))\end{aligned}$$

By starting at the root state, we ensure that the state condition holds for the entire statechart:

$$\mathsf{STATE\_COND}\ sc\ \ cf =_{def} \mathsf{STATE\_COND\_AUX}\ sc\ \ cf\ \ (\mathsf{ROOT}\ sc) \tag{5.2}$$

5.3.1 Evaluating the State Condition

Turning again to the traffic light example in Figure 2.1, the STATE_COND_AUX condition for the OR-state **N_S** would partially evaluate to:

$$\mathsf{ALT\_X\_EX}\ (\mathsf{STATE\_COND\_AUX}\ sc\ cf)\ (\mathsf{ALL\_F\_STATES}\ sc\ cf)\ (\mathsf{SUBSTATES}\ sc\ \mathbf{N\_S})$$

The substates of **N_S** are: **N_S_G**, **N_S_R**, **N_S_Y**. The system must be in exactly one of these basic states:

$$\begin{array}{l}(\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf\ \wedge \neg\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf\ \wedge \neg\mathsf{SemVAR}\ \mathbf{N\_S\_Y}\ cf)\ \vee \\ (\neg\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf\ \wedge \mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf\ \wedge \neg\mathsf{SemVAR}\ \mathbf{N\_S\_Y}\ cf)\ \vee \\ (\neg\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf\ \wedge \neg\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf\ \wedge \mathsf{SemVAR}\ \mathbf{N\_S\_Y}\ cf)\end{array}$$

Overall the STATE_COND limits the configuration to one of the columns of values for the basic states in the following table, where blanks entries have the value F:

FL	T									
N_S_G		T	T	T						
N_S_Y					T	T	T			
N_S_R								T	T	T
E_W_G		T			T			T		
E_W_Y			T			T			T	
E_W_R				T			T			T

5.4 Variable Condition (VAR_COND)

Given the set of transitions that can be taken, we now have to determine the effects of these transitions on the whole system. Their effects include modifying variables, generating events, and entering and exiting states. In order to resolve conflicts among transitions, all these modifications are collected and then resolved together. The function RESULT defined in the previous chapter gives the meaning of the actions, and entering and leaving states, as a set of pairs of variables and denotations of expressions, which can be evaluated relative to a particular configuration.

As stated in Chapter 2, the values for the variables in the next configuration must satisfy the following three properties:

1. If a given transition is taken, at the end of the step the system will be in a configuration that includes the destination state of the transition and all its actions will be carried out except where conflicts occur among the actions of all transitions.

2. If a variable is not modified by any transition in a step, then it retains its previous value.

3. If more than one modification is made to the same variable (i.e. a conflict occurs) then exactly one of these modifications will be true in the next configuration.

Only variables under this system's control should necessarily keep their previous value if they are not modified, i.e. internal variables. External data-items and external events may not retain their previous values between steps. The variables for the basic states are all internal, but the classification of events and data items as external or internal must be given. We assume that the argument *varlist* includes only the internal variables.

All modifications are considered together, whether they came from the same transition, perhaps in a BOTH statement, or from different transitions. The variable condition resolves conflicts among assignments

(CH defined in Section 5.4.2) and ensures the last property for variables that are not changed (UNCH defined in Section 5.4.1):

$$\begin{array}{l}\mathsf{VAR\_COND}\ sc\ cf\ cf'\ tflags\ varlist =_{def} \\ \quad \mathsf{EVERY}\,(\lambda v.\, \mathsf{UNCH}\ sc\ v\ cf\ cf'\ tflags \lor \\ \qquad \mathsf{CH}\ sc\ v\ cf\ cf'\ tflags)\ varlist\end{array} \tag{5.3}$$

where,
sc is the textual representation for the statechart,
$varlist$ is the set of internal variables,
cf is the current configuration,
cf' is the next configuration,
$tflags$ is a bit vector containing the flags for the transitions.

5.4.1 Unchanged Variables

Since the function RESULT returns a list of modifications to variables, we can determine the set of variables that are modified by taking the first element of each pair in the list of modifications:

$$\mathsf{CHANGEDVAR}\ modlist\ =_{def}\ \mathsf{MAP\ FST}\ modlist$$

The transition flags, $tflags$, give the set of transitions that are followed. For a given variable (v), the predicate UNCH will be true if, for each transition (1) the transition is not followed ($transcheck$), or (2) the transition does not modify v, and v keeps its previous value:

$$\begin{array}{l}\mathsf{UNCH}\ sc\ v\ cf\ cf'\ tflags =_{def} \\ \quad \textbf{let}\ tnames = \mathsf{GET\_TRANS\_NAMES}\ sc\ \textbf{in} \\ \quad \textbf{let}\ transcheck\ tname = \neg\mathsf{GET\_TR\_FLAG}\ tnames\ tflags\ tname \\ \quad \textbf{and}\ varcheck\ tname = \neg\mathsf{MEMBER}\ v\ (\mathsf{CHANGEDVAR}\ (\mathsf{RESULT}\ sc\ tname))\ \textbf{in} \\ \quad \mathsf{EVERY}\ (\lambda tname.(transcheck\ tname)\ \lor\ (varcheck\ tname))\ tnames\ \ \land \\ \quad \mathsf{BOOL}(\mathsf{EQVAL}\ (\mathsf{SemVAR}\ v\ cf',\ \mathsf{SemVAR}\ v\ cf))\end{array}$$

This predicate expresses the second property which states that variables keep their previous values if they are not modified by a transition.

5.4.2 Resolving Conflicts

When conflicts occur, there are several different possible results, as was discussed in Section 2.2.2. The interpretation that was chosen is that the actions are atomic and exactly one of the possible modifications to the variable occurs. Actions generated by BOTH statements fall into this category as well since no sequencing is assumed among multiple actions on a transition.

The function ACT looks at the list of modifications ($modlist$) and forms the disjunction of all possible modifications to a variable (v):

$$\begin{array}{l}\mathsf{ACT}\ v\ modlist\ cf\ cf' =_{def} \\ \quad \mathsf{EXISTS} \\ \quad (\lambda asn.(\mathsf{FST}\ asn = v)\ \land\ \mathsf{BOOL}(\mathsf{EQVAL}\ (\mathsf{SemVAR}\, v\ cf',\ (\mathsf{SND}\ asn)\ cf))) \\ \quad modlist\end{array}$$

Applying the ACT function to the results of each transition and then taking the disjunction of these clauses for all chosen transitions produces the effect of taking the disjunction of all possible modifications to a variable (v) in a step:

$$
\begin{array}{l}
\mathsf{CH}\ sc\ v\ cf\ cf'\ tflags =_{def} \\
\quad \textbf{let}\ tnames = \mathsf{GET\_TRANS\_NAMES}\ sc\ \ \textbf{in} \\
\quad \mathsf{EXISTS} \\
\quad\ (\lambda tname.\, \mathsf{GET\_TR\_FLAG}\ tnames\ tflags\ tname\ \wedge \\
\qquad\quad \mathsf{ACT}\ v\ (\mathsf{RESULT}\ sc\ tname)\ cf\ cf') \\
\quad\ tnames
\end{array}
$$

The variable condition results in a Boolean expression for each variable in one of two forms:

1. $cf'\ v = cf\ v$ when the variable has not been affected in this step, or
2. $(cf'\ \ v\ =\ e1\ \ cf)\ \ \vee\ \ (cf'\ \ v\ =\ e2\ \ cf)\ \ \vee\ \ldots(cf'\ \ v\ =\ en\ \ cf)$ where a variable takes on one of the modifications made to it in this step

A basic state variable will never be assigned multiple possible values for a given set of transitions.

5.4.3 Evaluating the Variable Condition

In the traffic light example, there are no conflicts in modifications to variables and only the values of basic states are changed in any step, but the expansion of the variable condition for one variable will provide an idea of what it reduces to for any statechart. Using $t0$ through $t7$ as the flags associated with the transitions in Figure 2.1, the predicate VAR_COND evaluated for the variable **FL** representing the basic state **FLASHING**, for the current configuration (cf) and the next configuration (cf'), partially evaluates to:

$$
\begin{array}{l}
(\neg t7\ \ \wedge\ \ \neg t6\ \ \wedge\ \ \mathsf{BOOL}\,(\mathsf{EQVAL}\,(\mathsf{SemVAR}\ \mathbf{FL}\ cf',\ \mathsf{SemVAR}\ \mathbf{FL}\ cf)))\ \ \vee \\
(t7\ \ \wedge\ \ \mathsf{BOOL}\,(\mathsf{EQVAL}\,(\mathsf{SemVAR}\ \mathbf{FL}\ cf', \mathsf{BFALSE})))\ \ \vee \\
(t6\ \ \wedge\ \ \mathsf{BOOL}\,(\mathsf{EQVAL}\,(\mathsf{SemVAR}\ \mathbf{FL}\ cf', \mathsf{BTRUE})))
\end{array}
$$

This means if the transitions that could modify **FL** are not taken ($t6$ and $t7$) then **FL** keeps its previous value. Otherwise the changes given by each transition when it is followed are stated.

5.5 Event Condition (EVENT_COND)

Evaluating events by examining counters means another another condition must be added to the next configuration relation to update the event counters. This condition looks at the events labeling each transition to update any event counters used by it:

$$
\begin{array}{l}
\mathsf{EVENT\_COND}\ sc\ cf\ cf' =_{def} \\
\quad \textbf{let}\ tnames = \mathsf{GET\_TRANS\_NAMES}\ sc\ \ \textbf{in} \\
\quad \mathsf{EVERY}\ (\lambda tname.\, \mathsf{UpdateEvent}\ \ (\mathsf{EVENT}\ \ (\mathsf{TRAN}\ \ sc\ tname))\ \ sc\ \ (cf, cf'))\ \ tnames
\end{array}
\tag{5.4}
$$

where $tnames$ is the set of transition names. This does not put any restrictions on the variables of the next configuration other than the event counters. The event counters are considered external so that they will not be affected by VAR_COND.

In order to define UpdateEvent, we need predicates that state the conditions that the counter is reset to zero when an event occurs and otherwise it is incremented providing it is not already at its maximum value. RESET states that the counter has the value zero in the configuration cf.

$$
\mathsf{RESET}\ counter\ cf =_{def}\ \ \mathsf{BOOL}(\mathsf{EQVAL}\ (\mathsf{SemVAR}\ counter\ cf,\ \mathsf{SemCONST}\ 0\ cf))
$$

INC expresses the relationship between the current and next configuration when the counter is incremented. Note that if the counter already holds its maximum value it is not changed:

$$
\begin{array}{l}
\mathsf{INC}\ \mathit{counter}\ \mathit{cf}\ \mathit{cf}' =_{def} \\
\quad (\mathsf{MAXVALUE}\ (\mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf}) \wedge \\
\quad\quad \mathsf{BOOL}\,(\mathsf{EQVAL}\ (\mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf}',\ \mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf}))) \ \vee \\
\quad (\neg(\mathsf{MAXVALUE}\ (\mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf})) \wedge \\
\quad\quad \mathsf{BOOL}\,(\mathsf{EQVAL}\ (\mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf}', \\
\quad\quad\quad \mathsf{PLUS}\ (\mathsf{SemVAR}\ \mathit{counter}\ \mathit{cf},\ \mathsf{SemCONST}\ 1\ \mathit{cf}))))
\end{array}
$$

For each single event, we can define a relation that holds true if the event occurs in this step. The following functions determine if the system enters or exists a state (stn):

$$\mathsf{SemEN}\ \mathit{sc}\ \mathit{stname}\ =_{def}\ \lambda(\mathit{cf},\mathit{cf}').\,\neg\mathsf{INSTATE}\ \mathit{sc}\ \mathit{cf}\ \mathit{stname}\ \wedge\ \mathsf{INSTATE}\ \mathit{sc}\ \mathit{cf}'\ \mathit{stname}$$

$$\mathsf{SemEX}\ \mathit{sc}\ \mathit{stname}\ =_{def}\ \lambda(\mathit{cf},\mathit{cf}').\,\mathsf{INSTATE}\ \mathit{sc}\ \mathit{cf}\ \mathit{stname}\ \wedge\ \neg\mathsf{INSTATE}\ \mathit{sc}\ \mathit{cf}'\ \mathit{stname}$$

A primitive event occurs when the variable representing it (evn) changes its value between cf and cf':

$$\mathsf{SemEV}\ \mathit{evn} =_{def}\ \lambda(\mathit{cf},\mathit{cf}').\,\neg\mathsf{BOOL}\,(\mathsf{EQVAL}\ (\mathsf{SemVAR}\ \mathit{evn}\ \mathit{cf}',\ \mathsf{SemVAR}\ \mathit{evn}\ \mathit{cf}))$$

The predicate to update the counters for single events, uses one of the three functions defined above, given as a parameter in *check*, to determine if it should reset or increment the counter:

$$
\begin{array}{l}
\mathsf{Update}\ \mathit{check}\ \mathit{counter}\ =_{def} \\
\quad \lambda(\mathit{cf},\mathit{cf}'). \\
\quad\quad (\mathit{check}\ (\mathit{cf},\mathit{cf}') \wedge \mathsf{RESET}\ \mathit{counter}\ \mathit{cf}') \vee \\
\quad\quad (\neg(\mathit{check}\ (\mathit{cf},\mathit{cf}')) \wedge \mathsf{INC}\ \mathit{counter}\ \mathit{cf}\ \mathit{cf}')
\end{array}
$$

Grouping these together, $\mathsf{UpdateSingleEvent}$, takes a pair of configurations and returns a predicate limiting the value of the counter in the next configuration for the single events:

$$
\begin{array}{l}
\mathsf{UpdateSingleEvent}\ (\mathsf{EV}\ \mathit{evn}\ \mathit{counter})\ \mathit{sc}\ =_{def} \\
\qquad\qquad \mathsf{Update}\ (\mathsf{SemEV}\ \mathit{evn})\ \mathit{counter} \\
\mathsf{UpdateSingleEvent}\ (\mathsf{EX}\ \mathit{stname}\ \mathit{counter})\ \mathit{sc}\ =_{def} \\
\qquad\qquad \mathsf{Update}\ (\mathsf{SemEX}\ \mathit{sc}\ \mathit{stname})\ \mathit{counter} \\
\mathsf{UpdateSingleEvent}\ (\mathsf{EN}\ \mathit{stname}\ \mathit{counter})\ \mathit{sc}\ =_{def} \\
\qquad\qquad \mathsf{Update}\ (\mathsf{SemEN}\ \mathit{sc}\ \mathit{stname})\ \mathit{counter}
\end{array}
$$

Finally, $\mathsf{UpdateEvent}$ is a recursively defined predicate that uses the above parts to express what it means to update any event:

$$
\begin{array}{l}
\mathsf{UpdateEvent}\ (\mathsf{NONE})\ \mathit{sc}\ =_{def}\ \lambda(\mathit{cf},\mathit{cf}').\,\mathsf{T} \\
\mathsf{UpdateEvent}\ (\mathsf{EVEXPR}\ \mathit{ev}\ \mathit{exp})\ \mathit{sc}\ =_{def}\ \mathsf{UpdateEvent}\ \mathit{ev}\ \mathit{sc} \\
\mathsf{UpdateEvent}\ (\mathsf{SE}\ \mathit{se})\ \mathit{sc}\ =_{def}\ \mathsf{UpdateSingleEvent}\ \mathit{se}\ \mathit{sc} \\
\mathsf{UpdateEvent}\ (\mathsf{TM}\ \mathit{se}\ \mathit{exp})\ \mathit{sc}\ =_{def}\ \mathsf{UpdateSingleEvent}\ \mathit{se}\ \mathit{sc} \\
\mathsf{UpdateEvent}\ (\mathsf{AND\_E}\ \mathit{ev}_1\ \mathit{ev}_2)\ \mathit{sc}\ =_{def} \\
\quad \lambda(\mathit{cf},\mathit{cf}').\ (\mathsf{UpdateEvent}\ \mathit{ev}_1\ \mathit{sc}\ (\mathit{cf},\mathit{cf}')\ \wedge\ \mathsf{UpdateEvent}\ \mathit{ev}_2\ \mathit{sc}\ (\mathit{cf},\mathit{cf}')\ ) \\
\mathsf{UpdateEvent}\ (\mathsf{OR\_E}\ \mathit{ev}_1\ \mathit{ev}_2)\ \mathit{sc}\ =_{def} \\
\quad \lambda(\mathit{cf},\mathit{cf}').\ (\mathsf{UpdateEvent}\ \mathit{ev}_1\ \mathit{sc}\ (\mathit{cf},\mathit{cf}')\ \wedge\ \mathsf{UpdateEvent}\ \mathit{ev}_2\ \mathit{sc}\ (\mathit{cf},\mathit{cf}')\ ) \\
\mathsf{UpdateEvent}\ (\mathsf{NOT\_E}\ \mathit{ev})\ \mathit{sc}\ =_{def}\ \mathsf{UpdateEvent}\ \mathit{ev}\ \mathit{sc}
\end{array}
$$

5.6 Summary

This chapter and the previous one together present an operational semantics for a working subset of statecharts as a next configuration relation. Because these are all total functions, every statechart has an interpretation. Some current configurations in statecharts that violate the assumptions made about their form may have no possible next configurations so the relation will always fail.

The difficulty in giving these semantics is that the components of statecharts are not completely compositional. The meaning of expressions and actions are expressed simply by examining their parts. Timeout events use counters so they can be evaluated relative to the current configuration only. But the overall conditions on transitions and values of variables in the next configuration have to consider all parts of the statechart.

Our definition of a step is simpler than that used by other versions of the semantics but is easier and clearer to express. It expresses non-determinism among transitions on the same level and priority among transitions from states related in the hierarchy. Interpretations for enabled transitions with conflicting destination states and those that go between orthogonal components have not yet been included. Race conditions and multiple conflicting actions on a transition are resolved by considering all actions together when assigning values to the variables in the next configuration.

Given a current configuration, it may be indeterminite as to which set of transitions will be chosen for this step. If there are conflicts among the actions of the transitions chosen for the step, it is also indeterminate as to what value a variable will take on. The result is that several next configurations may satisfy the relation for the same current configuration.

These semantics have been used in a model checker for statecharts which is presented in the next chapter. They could also form the basis for other types of analysis and simulation or to examine properties of the semantics themselves.

Chapter 6

The Model Checker

A model checker shows that a given property or descriptive specification is true of a model of a system. In this chapter we present a model checking algorithm implemented as a higher-order logic function. It takes as a parameter a next configuration relation characterizing the semantics of the model. This could be the NC relation defined in the previous chapter for statecharts. The descriptive specification is a predicate on configurations which should eventually be true within a bounded number of steps along either some or all execution paths starting from an initial set of configurations. To execute this function, we represent the configuration in Boolean values. The traffic light example is used to demonstrate the model checker. Invariants can be verified using the model checker to prove the induction step in an inductive proof that some property holds for all times.

6.1 Introduction

Many forms of model checking have been developed and used for different purposes. In general, a model checker tests whether a given property holds true in a finite state machine model of a system. A statechart can be considered as a finite "state" machine where the "states" of the machine are all the possible configurations and the "state" transition relation is given by the next configuration relation NC. The descriptive specification is the property to test.

Treating the values of elements of the configuration symbolically, it is possible to show that the property is true over a class of configurations in one run of the model checker. We can also examine the consequences of external events occurring at any time.

In the past, these tools have suffered from the configuration explosion problem[1] when all configurations were explicitly represented. The symbolic model checking algorithm used here is a limited form of the one presented by McMillan [24] where binary decision diagrams (BDDs) are used for efficient representation of the possible configurations. Boolean functions give characteristic functions for possible configurations under evaluation.

The sections in this chapter describe the following elements required to carry out the model checking:

- a way of representing the system configuration in Boolean variables so that it can be executed (Section 6.2)
- notation for expressing the descriptive specification (Section 6.3)
- an algorithm for doing the model checking (Section 6.4)

The model checker is independent of the next configuration relation which characterizes the semantics.

Two examples based on the traffic light are used to illustrate the model checker. Section 6.7 looks at using invariants to prove a property for all execution times in the model. The chapter concludes with discussions on efficiency and another descriptive specification notation that could be used.

[1] This is usually called the state explosion problem.

6.2 Representing the Configuration

The configuration can be represented in Boolean variables (bits) so that the next configuration relation, NC, defined in the previous chapter can be executed. By execution, we mean giving two configurations as arguments to the relation and automatically simplifying the relation to true or false.

In the previous chapter the configuration is described as a mapping from variables to values. These values are represented as vectors of Booleans:

$$Value \equiv (Bool)list$$

A configuration is created by applying the function MEM to a list of variable records, *varlist* and a bit vector *bv*. MEM *varlist bv* takes a variable name and returns a bit vector for the value of that variable. Since some variables require more than one bit, it needs information on how many bits are associated with each variable. The argument *varlist* is actually an ordered list of records that have fields for the variable name, the status of the variable as external or internal and the number of bits that should be used to represent that variable.

Using the function SPLITBV which, given a number of bits x and a bit vector bv, returns the pair where the first element is the first x bits of bv and the second element is the remaining bits of bv, the function MEM can be defined as:

$$\begin{array}{l} \mathsf{MEM}\ varlist\ bv\ varname = \\ \quad (varlist = [\,]) \rightarrow= [\mathsf{T}]\ \mid \\ \quad \mathbf{let}\ bits = \mathsf{SPLITBV}\ (\mathsf{SND}\ (\mathsf{SND}\ (\mathsf{HD}\ varlist)))\ bv\ \mathbf{in} \\ \quad ((\mathsf{FST}\,(\mathsf{HD}\ varlist) = varname) \Rightarrow \mathsf{FST}\ bits \mid \mathsf{MEM}\ (\mathsf{TL}\ varlist)\,(\mathsf{SND}\ bits)\ varname) \end{array}$$

For example, if *varlist* contains:

$$[(\mathbf{X}, \mathrm{Int}, 1); (\mathbf{Y}, \mathrm{Ext}, 5); (\mathbf{Z}, \mathrm{Ext}, 2)]$$

and bv is

$$[b0; b1; b2; b3; b4; b5; b6; b7]$$

then applying the configuration MEM *varlist bv* to the variable **Z** would return $[b6; b7]$.

Note that if a variable is not contained in *varlist*, the one element bit vector containing only the value true is returned. Obviously there is the potential for errors in generating the list of records for the variables and Chapter 8 will describe a tool that does this automatically for a particular statechart. This tool determines the status of the variable as internal or external by examining how the variable is used in the statechart. This field may be modified by the user. Making a variable external is a more conservative verification test than assuming that the variable maintains its value unless explicitly changed.

There are many different types of variables that can be used in statecharts. Basic states are represented by one bit vectors treated as one Boolean variable. Booleans and primitive event variables are also one bit vectors. Natural numbers and event counters can be vectors of any length which are interpreted as numbers. The tool automatically generates this information giving natural numbers and event counters a default length of one, since fewer bits are more efficient for the model checker to compute. This information can be changed easily before it is used and it is expected that the user would tailor the number of bits to the expected values of these variables.

In the traffic light example, the data-items are **EW_GREEN_TIME** and **NS_GREEN_TIME** (**EW_G_T** and **NS_G_T** respectively) which give the timeout step number for how long the light stays green in a each direction. If each of these could take on the possible values 0 through 3, then they can be represented in two bits. Their associated event counters would need three bits so that the legitimate value 3 is not the maximum value of the counter (Section 4.7). The event counters for the transitions that move the system into the yellow light state in each component must have at least two bits since they have to count up to the value 2. The other event counters (**EV_MALF**, **EV_RESET**) only need one bit since we only have to check if the event happened in the previous step or not. The complete information for the variables in the traffic light statechart of Figure 2.1 is called $tlsINFO$ and given by the following chart:

Variable Type	Name	# Bits	Internal/External
Basic states	**FL**	1	Int
	E_W_R	1	Int
	E_W_Y	1	Int
	E_W_G	1	Int
	N_S_Y	1	Int
	N_S_R	1	Int
	N_S_G	1	Int
Data items	**NS_G_T**	2	Ext
	EW_G_T	2	Ext
Primitive Event flags	**MALF**	1	Ext
	RESET	1	Ext
Event counters	**EV_MALF**	1	Ext
	EV_RESET	1	Ext
	EN_N_S_G	3	Ext
	EN_N_S_Y	2	Ext
	EN_E_W_G	3	Ext
	EN_E_W_Y	2	Ext

6.2.1 Operations on Values

The operations outlined in Section 4.3 can now be defined as operations on bit vectors. For example, BOOL returns the first element of the bit vector, to convert a value into a Boolean:

$$\mathsf{BOOL}\ (x : Value)\ =_{def} \mathsf{HD}\ x$$

The operations on Boolean values convert them to Booleans and then carry out the appropriate logical connective, as in:

$$\mathsf{BAND}\ (x : Value, y : Value)\ =_{def}\ \mathsf{BVAL}(\mathsf{BOOL}\, x\ \wedge\ \mathsf{BOOL}\, y)$$

NVAL is somewhat more complicated because it has to convert a natural number into a bit vector. Values for numbers are unsigned bit vectors of any length in little endian format.[2]

$$\begin{aligned}&\mathsf{NVAL}\ n =_{def}\\ &\quad (n = 0) \Rightarrow [\,]\ \mid\ \mathsf{CONS}\,(n\ \mathsf{MOD}\ 2 = 1)\,(\mathsf{NVAL}\ (n\ \mathsf{DIV}\ 2))\end{aligned}$$

The operations on numbers may involve resizing the bit vectors to accommodate carry bits. For example for PLUS, we have to call on an auxiliary function that adds two values and a carry bit where the first carry in is F:

$$\mathsf{PLUS}\ a\ b\ =_{def}\ \mathsf{PLUS\_AUX}\ a\ b\ \mathsf{F}$$

$$\begin{aligned}&\mathsf{PLUS\_AUX}\, a\, b\, c =\\ &\quad (a = [\,]) \Rightarrow \mathsf{PLUS2\_AUX}\, b\ c\ \mid\\ &\quad ((b = [\,]) \Rightarrow \mathsf{PLUS2\_AUX}\ a\ c\ \mid\\ &\quad \mathsf{CONS}\ \ ((\mathsf{HD}a) \wedge \neg(\mathsf{HD}\ b) \wedge \neg c\ \ \vee\ \ \neg(\mathsf{HD}\ a) \wedge (\mathsf{HD}\ b) \wedge \neg c\ \ \vee\\ &\qquad\qquad \neg(\mathsf{HD}\ a) \wedge \neg(\mathsf{HD}\ b) \wedge c\ \ \vee\ \ (\mathsf{HD}\ a) \wedge (\mathsf{HD}\ b) \wedge c\ )\\ &\quad (\mathsf{PLUS\_AUX}\,(\mathsf{TL}\ a)\,(\mathsf{TL}\ b)\ \ ((\mathsf{HD}\ a) \wedge (\mathsf{HD}\ b)\ \ \vee\ \ c \wedge (\mathsf{HD}\, b)\ \ \vee\ \ (\mathsf{HD}\ a) \wedge c)))\end{aligned}$$

The auxiliary function puts a true value at the head of the list if one or all of the first bits of the three values are true. The carry bit is true if any two of the first bits of the three values are true. The final function adds the carry bit to the result:

$$\begin{aligned}&\mathsf{PLUS2\_AUX}\ a\ c =\\ &\quad \mathsf{CONS}\,(c\ \wedge\ \neg(\mathsf{HD}\ a)\ \ \vee\ \ \neg c\ \wedge\ (\mathsf{HD}\ a))\,(\mathsf{PLUS2\_AUX}\ (\mathsf{TL}\ a)\ (c\ \wedge\ (\mathsf{HD}\ a)))\end{aligned}$$

[2] Little endian format means the rightmost bit is the most significant.

The equality operator sets the bits of its first argument $bv1$ equal to the bits of its second argument $bv2$. If $bv2$ has more bits than $bv1$ then these extra bits are not used:

$$\mathsf{EQUAL}\,(x : Value,\ y : Value)\ =_{def}\ \mathsf{BVEQUAL}\ x\ (\mathsf{SIZED}\ (\mathsf{LENGTH}\ x)\ y)$$

These operations are based on definitions given in a bit vector package for HOL [28] and the definitions for the remaining functions are given in Appendix B.

6.3 Descriptive Specifications

The descriptive specification language for this model checker includes bounded eventually temporal logic statements in one of two forms:

- Starting from an initial set of configurations **i**, the property **f** eventually holds within n steps on *all* execution paths.
- Starting from an initial set of configurations **i**, the property **f** eventually holds within n steps on *some* execution path.

where **i** and **f** are predicates on configurations which can be considered characteristic functions for a set of configurations.

The operational specification is given by the relation **NextConfig** which relates two configurations. To model check a particular statechart sc we would supply $\mathsf{NC}\ sc$ for **NextConfig** but the model checking algorithm would work for any other model whose operation can be described by a next configuration relation.

To show the first type of statement is true we show that the following predicate is true:

$$\mathsf{MC\_A}\ n\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f}$$

Similarly, the second statement is verified using:

$$\mathsf{MC\_E}\ n\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f}$$

The functions MC_A and MC_E implement the model checking algorithm and will be described in the next section. Section 6.7 shows how MC_A can be used as part of an inductive proof to show **f** is true for all times, not just within a time constraint.

6.4 The Model Checking Algorithm

The task for a model checker is to show that the descriptive specification is true. The model checking algorithm depends on the descriptive specification language. It uses the representation of the configuration of the system and tests if a given property holds in that configuration. If the property does not hold in all configurations that the system is currently in, then it determines the representation for the next configuration of the system and iterates this process until either the formula does hold along all paths leading to the current set of configurations or it has tried the number of iterations given by the timing constraint. We will begin by explaining the algorithm that determines if property holds along every execution path (MC_A) and then describe the simple changes to calculate MC_E.

To describe the model checking algorithm, we make use of the following:

i is the characteristic function for the set of initial configurations,

NextConfig is the next configuration relation, and

f is the characteristic function for the set of configurations that satisfy the property to verify

These are all predicates that take configurations as arguments and return true or false depending on whether their arguments satisfy the predicate.

Over a limited number of steps, the model checker determines the set of next configurations that do not satisfy the formula and therefore need to be checked further. A characteristic function for the set of next configurations (cf') is given by asking if there are any elements of the initial configuration(cf) that are related by **NextConfig** to cf'.[3]

$$\lambda cf'.\exists cf.\mathbf{i}\, cf \wedge \mathbf{NextConfig}\, cf\, cf' \tag{6.1}$$

To check if the property **f** holds in all possible next configurations, we formulate the question of whether any cf''s exist that do not satisfy **f**:

$$\mathbf{let}\ \ check = \exists cf'.\exists cf.\mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \neg\mathbf{f}\, cf' \tag{6.2}$$

If this expression is false then the property is true in all current configurations ($\neg check$) and the model checking process can stop. If, however, this expression is true, then **f** does *not* hold in all next configurations of the set of current configurations and the model checking process should continue to check if **f** will eventually be satisfied along all execution paths.

A representation for the set of next configurations that do not satisfy **f** is needed, since these are the only paths we need to continue to examine. The characteristic function for this set is given in the above expression (Equation 6.2) without the quantification over the next state:

$$\mathbf{let}\ nextX = \lambda cf'.\exists cf.\mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \neg\mathbf{f}\, cf' \tag{6.3}$$

This then becomes the initial state **i** used in the next iteration. The complete model checking process is given by the following recursively defined function, called MC_A, where *step* is a constant natural number giving the time constraint:

$$\begin{array}{ll}
\mathsf{MC_A}\ step\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f} =_{def} & \\
\quad \mathbf{let}\ check = \exists cf'.\exists cf.\ \mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \neg\mathbf{f}\, cf'\ \mathsf{in} & (6.2)\\
\quad \mathbf{let}\ nextX = \lambda cf'.\exists cf.\ \mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \neg\mathbf{f}\, cf'\ \mathsf{in} & (6.3)\\
\quad (step = 0) \Rightarrow \mathsf{False}\ \mid\ \neg(check)\ \vee\ \ \mathsf{MC_A}\ (step-1)\ nextX\ \mathbf{NextConfig}\ \mathbf{f} &
\end{array}$$

If the process has checked all steps without finding that the property is satisfied along all paths, it returns false.

To test whether the property **f** eventually holds along *some* execution path rather than along all paths, MC_E checks if the **f** holds true in any configuration ($check$), and if not continues iterating with all execution paths because one has not yet been found that satisfies the formula:

$$\begin{array}{l}
\mathsf{MC_E}\ step\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f} =_{def}\\
\quad \mathbf{let}\ check = \exists cf'.\exists cf.\ \mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \mathbf{f}\, cf'\ \mathsf{in}\\
\quad \mathbf{let}\ nextX = \lambda cf'.\exists cf.\ \mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf'\ \mathsf{in}\\
\quad (step = 0) \Rightarrow \mathsf{False}\ \mid\ (check)\ \vee\ \ \mathsf{MC_E}\ (step-1)\ nextX\ \mathbf{NextConfig}\ \mathbf{f}
\end{array}$$

6.4.1 Variations on these Algorithms

A variation of this process checks whether the descriptive specification is true in the initial configuration, rather than starting in the next configuration as above. For each of the tests given above, the corresponding functions MC_A_I and MC_E_I also check **f** in the initial configuration:

$$\begin{array}{l}
\mathsf{MC_A_I}\ step\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f} =_{def}\\
\quad \mathbf{let}\ check = \exists cf.\ \mathbf{i}\, cf \ \wedge\ \neg\mathbf{f}\, cf\ \mathsf{in}\\
\quad \mathbf{let}\ nextX = \lambda cf'.\exists cf.\ \mathbf{i}\, cf \ \wedge\ \neg\mathbf{f}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf'\ \mathsf{in}\\
\quad (step = 0) \Rightarrow \mathsf{False}\ \mid\ \neg(check)\ \vee\ \ \mathsf{MC_A_I}\ (step-1)\ nextX\ \mathbf{NextConfig}\ \mathbf{f}
\end{array}$$

[3]Throughout this explanation, the expression $\exists cf$ will be used to indicate existential quantification over all the bits of the configuration. This is actually implemented using the function EXISTSN defined in Chapter 4.

$$
\begin{array}{l}
\mathsf{MC\_E\_I}\ step\ \mathbf{i}\ \mathbf{NextConfig}\ \mathbf{f} =_{def} \\
\quad \mathbf{let}\ check = .\,\exists cf.\ \mathbf{i}\,cf\ \wedge\ \mathbf{f}\,cf'\ \mathsf{in} \\
\quad \mathbf{let}\ nextX = \lambda cf'.\,\exists cf.\ \mathbf{i}\,cf\ \wedge\ \mathbf{NextConfig}\,cf\,cf'\ \mathsf{in} \\
\quad (step = 0) \Rightarrow \mathsf{False}\ \mid\ (check)\ \vee\ \mathsf{MC\_E\_I}\ (step-1)\ nextX\ \mathbf{NextConfig}\ \mathbf{f}
\end{array}
$$

For the same step number, these will check execution paths one unit smaller in length than the algorithms that start at the next configuration. This variation is often useful when checking if a certain action is accomplished (liveness) and it is equally good if the action is already accomplished.

MC_A and MC_E will be called MC_A_NS and MC_E_NS respectively to indicate that they start the model checking in the next step from the initial configuration.

6.4.2 Running the Model Checker

The model checker can run any of these model checking functions. MC is a general function that precalculates the number of bits needed to represent the configuration (GETNUMBITS) and determines the internal variables (INTVAR) from the variable records ($frame$). The function MEM with the set of variable records is passed to the model checking functions. The particular model checking function to be used is given by mc. The next configuration relation to be used is ns.

$$
\begin{array}{l}
\mathsf{MC}\ mc\ step\ i\ ns\ f\ frame =_{def} \\
\quad \mathbf{let}\ nbits = \mathsf{GETNUMBITS}\ frame\ \mathbf{in} \\
\quad (\mathbf{let}\ intvar = \mathsf{INTVAR}\ frame\ \mathbf{in} \\
\quad mc\ step\ nbits\ (\mathsf{MEM}\ frame)\ i\ (ns\ intvar)\ f)
\end{array}
$$

6.5 Descriptive Specifications for Statecharts

The property to check is expressed in the target language making reference only to the current configuration. It must be given as a function that takes a configuration as an argument and returns true or false depending on whether the property is satisfied in that configuration. A safety property of the traffic light is that it never shows either a green or yellow light in both directions at the same time. A descriptive specification for a configuration where this situation might exist is:

$$
\begin{array}{l}
\mathsf{TROUBLE}\ =_{def} \\
\quad \lambda cf. \\
\qquad (\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_Y}\ cf)\ \vee\ \mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_G}\ cf))\ \wedge \\
\qquad (\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{E\_W\_Y}\ cf)\ \vee\ \mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{E\_W\_G}\ cf))
\end{array}
$$

The model checker could test whether this property ever becomes true (MC_E) within a limited number of steps.

6.5.1 Initial Configuration

The possible initial configurations of the system are specified by a characteristic function of a set of configurations. The model checking algorithm can begin from all possible system configurations, by giving $\lambda cf.\,\mathsf{T}$ as its initial set of configurations. Sometimes, however, all possible system configurations are not reachable within the model given by the statechart. For example, it is a legal state configuration to have both lights showing green, but we hope that this is not a reachable configuration. An overly large set of possible initial configurations could result in the model checker returning false negatives if the formula is shown to be false along a path of execution that began at an unreachable configuration.

Often the model checker should start in the starting configuration of the system. Similar to the semantic function for entering the source state of a transition ($\mathsf{ENTERSRC}$) which returns a list of assignment pairs to set the basic states to true, $\mathsf{ENTERDEF}$ is true if the basic states determined by following the defaults from a given state are all true:

$$
\begin{array}{l}
\mathsf{ENTERDEF}\; sc\; stname =_{def} \\
\quad \lambda cf. \\
\qquad (\mathsf{TYP}\; sc\; stname = \mathsf{B}) \Rightarrow \mathsf{BOOL}\,(\mathsf{SemVAR}\; stname\; cf) \mid \\
\qquad ((\mathsf{TYP}\; sc\; stname = \mathsf{A}) \Rightarrow \\
\qquad\quad \mathsf{EVERY}\; (\lambda stn.\, \mathsf{ENTERDEF}\; sc\; stn\; cf)\,(\mathsf{SUBSTATES}\; sc\; stname) \mid \\
\qquad \mathsf{ENTERDEF}\; sc\; (\mathsf{DEFAULT}\; sc\; stname)\; cf
\end{array}
$$

For any statechart that has the necessary default entrances, the predicate describing the starting configuration uses $\mathsf{ENTERDEF}$ starting from the root state and also ensures that the system is in a legal state configuration so that the values of basic states not set by $\mathsf{ENTERDEF}$ will be false:

```
              _  _      __      _          __     __
   |___     |__|     |  |     |        |__|   |__|
   |        |  |     |__|     |__      |__|   |__|

            Version 2.0, built on 15/7/92


        .... load definitions for semantics, model checking, traffic light


#let TROUBLE = new_definition(
        `TROUBLE`,
        "TROUBLE = \(cf:Config).
                EXISTS (\v.BOOL(SemVAR v cf)) [`E_W_G`;`E_W_Y`] /\
                EXISTS (\v.BOOL(SemVAR v cf)) [`N_S_G`;`N_S_Y`]");;
####TROUBLE =
|- TROUBLE =
   (\cf.
     EXISTS(\v. BOOL(SemVAR v cf))[`E_W_G`;`E_W_Y`] /\
     EXISTS(\v. BOOL(SemVAR v cf))[`N_S_G`;`N_S_Y`])

#VOSS "MC MC_E_NS 1 (INITIAL tls) (NC tls) TROUBLE tlsINFO";;
`F` : string

#VOSS "MC MC_E_NS 2 (INITIAL tls) (NC tls) TROUBLE tlsINFO";;
`F` : string

#VOSS "MC MC_E_NS 5 (INITIAL tls) (NC tls) TROUBLE tlsINFO";;
`T` : string
```

Figure 6.1: Running the model checker for the traffic light

$$\begin{array}{l} \mathsf{INITIAL}\ sc\ =_{def} \\ \quad \lambda cf. \\ \qquad \mathsf{ENTERDEF}\ sc\ (\mathsf{ROOT}\ sc)\ cf\ \wedge\ \mathsf{STATE\_COND}\ sc\ cf \end{array}$$

Data-items and events may be assigned symbolic values or constants in the initial configuration.

6.6 Traffic Light Example

Earlier we defined TROUBLE as a characteristic function for configurations where the traffic light is green or yellow in both directions. Using the MC_E function, we can check whether this property is ever true within a certain number of steps from the starting configuration of the traffic light statechart.

The definitions for the semantic and model checking functions have been defined in HOL. To run the model checker, we only need to load the traffic light statechart (tls) and its associated configuration representation information ($tlsINFO$) and then provide the definitions for the descriptive specification, before giving Voss the model checking expression to evaluate.

The output in Figure 6.1 shows **'T'** or **'F'** to indicate whether the expression is true or false. The results of the model checker tell us that five steps from the starting configuration, the traffic light could end up in a situation where both lights are showing either green or yellow. Referring back to Figure 2.1, we can see how this could occur. The characteristic function for the initial set of configurations does not limit the event counters at all so in the first step, **t0** can be followed since **NS_G_T** could be equal to the counter **EN_N_S_G** and the system moves into the states **N_S_Y** and **E_W_R**. In this first step the event counter **EN_N_S_Y** is reset to 0. For the next two steps, the system remains in this state configuration as the event counters get updated. In the fourth step, the event counter **EN_N_S_Y** has the value 2 so transition **t1** can be taken moving the system into both red light states. In the next step the triggers for both **t2** and **t5** are true so both of these transitions can be taken and after five steps the traffic light is showing a green light in both directions! In other words, we have used the model checker to demonstrate the property SAFE does not hold in the the traffic light statechart of Figure 2.1.

Obviously this is a situation we want to avoid and the solution is to strengthen the triggering events of **t2** and **t5** so that they can not both occur at the same time. As suggested in the discussion of this example in the STATEMATE manual [14], the model is revised so that the trigger for moving from a red to a green state is that the system has just *entered* the other component's red light state. This is given by Figure 6.2 (from Figure 3-22 in [14]). This should eliminate the previous problem because the system will have been in **E_W_R** for a number of steps before the **N_S** component arrives in its red state and the traffic changes direction.

6.7 Invariants

We could continue to check the new traffic light statechart to see if it ever enters a configuration that satisfies TROUBLE but there is a finite number of steps that we can test in this model checker. It would be better to show that the model is always in a safe configuration, which can be defined as the opposite of TROUBLE, where *tls* is the statechart:

$$
\begin{array}{l}
\mathsf{SAFE} \;=_{def} \\
\quad \lambda cf. \\
\qquad (\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf) \ \vee\ \mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{E\_W\_R}\ cf) \ \vee \\
\qquad\quad \mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{FL}\ cf)) \ \wedge \\
\qquad \mathsf{STATE\_COND}\ tls\ cf
\end{array}
$$

Using MC_A, we could show that starting from the initial configuration, after multiple steps the system always ends up in a configuration that satisfies this property. This approach still suffers from only being able to run the model checker for a finite number of steps when the execution of the traffic light could continue indefinitely.

An alternative approach is to use MC_A to prove that if the system starts in a configuration that satisfies this property then, after one step, it always ends up in one that also satisfies this property. If this property is also true in the starting configuration of the model then by induction the property holds in all reachable configurations of the system. The model checker can be used to automatically prove the induction step.

If we try verifying the above descriptive specification of SAFE for one step as in Figure 6.3, we find it does not hold. This does not necessarily mean that the invariant does not hold in the model. It just may not be strong enough. Figure 6.4 illustrates what is happening. We can limit the set of all possible configurations to those that have legal state configurations (L). The property SAFE describes a subset of these given by S. We used MC_A to try to show that all the configurations within S lead to configurations also in S in one step. This was not true because, as the model checker has shown, some next configurations fall outside of S resulting in X. For example, if **N_S_R** and **E_W_R** are both entered in a step then in the resulting configuration the event counters **EN_E_W_R** and **EN_N_S_R** both have the value 0. This configuration satisfies the predicate SAFE but in the next step both **t2** and **t5** can be taken, leading to an unsafe configuration.

The predicate SAFE may still be true in all reachable configurations of the system. If we find a subset of S, called INV, where all of its elements lead to configurations that also belong to INV as in Figure 6.5 and INV holds in the starting configuration of the statechart then by induction we have a property that holds for all reachable configurations.

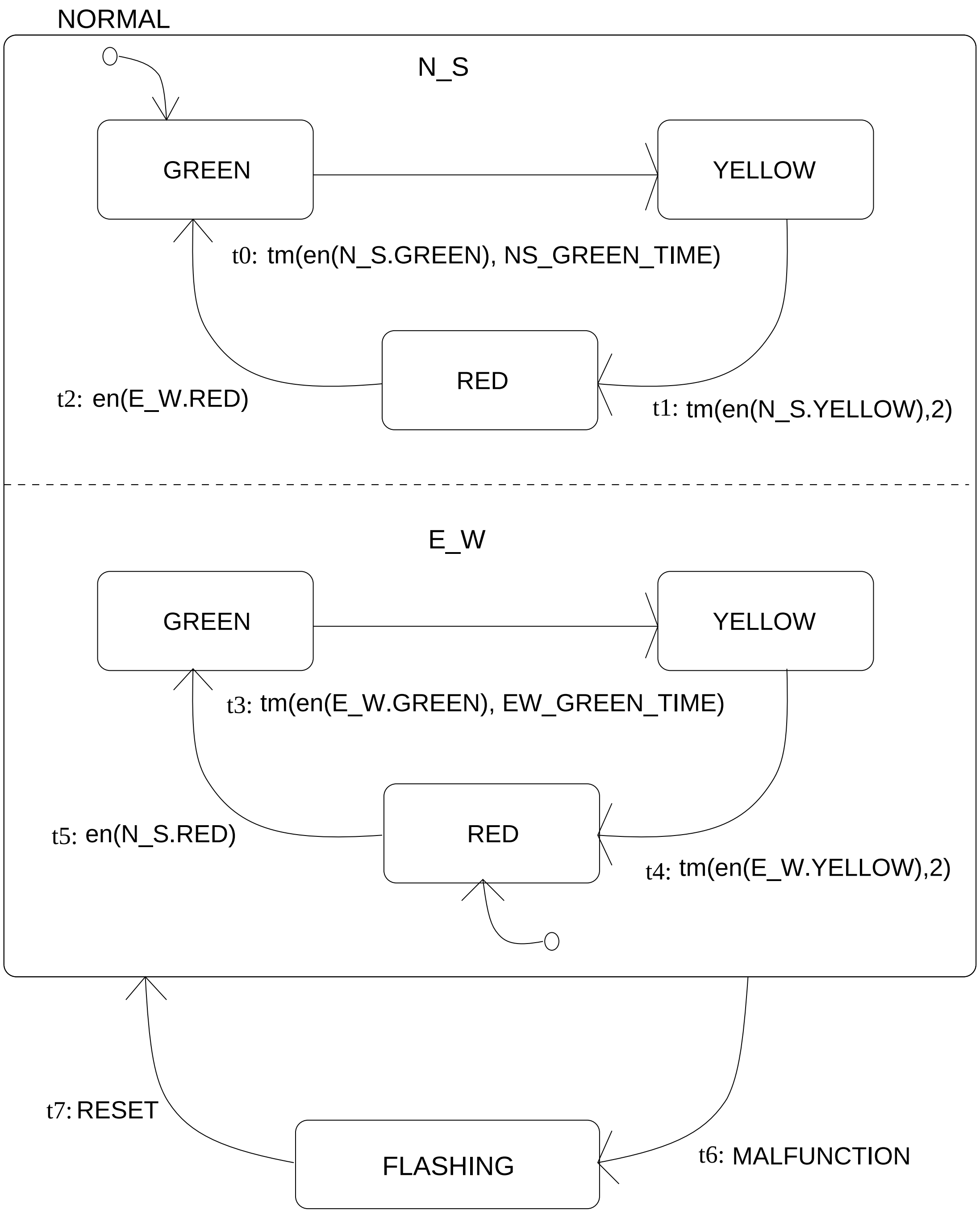

Figure 6.2: Corrected traffic light

```
               _ _      __      _        __     __
   |___     |__|     |  |     |       |__|   |__|
   |        |  |     |__|     |__     |__|   |__|

          Version 2.0, built on 15/7/92


     .... load definitions for semantics, model checking, new traffic light


#let SAFE = new_definition(
    `SAFE`,
    "SAFE  = \(cf:Config).
        (BOOL(SemVAR `N_S_R` cf) \/
        BOOL(SemVAR `E_W_R` cf) \/
        BOOL(SemVAR `FL` cf)) /\
        STATE_COND tls cf");;
######SAFE =
|- SAFE =
   (\cf.
     (BOOL(SemVAR `N_S_R` cf) \/
      BOOL(SemVAR `E_W_R` cf) \/
      BOOL(SemVAR `FL` cf)) /\
     STATE_COND tls cf)

VOSS "MC MC_A_NS 1 SAFE (NC newtls) SAFE newtlsINFO";;
`F` : string
```

Figure 6.3: First attempt at an invariant for the traffic light

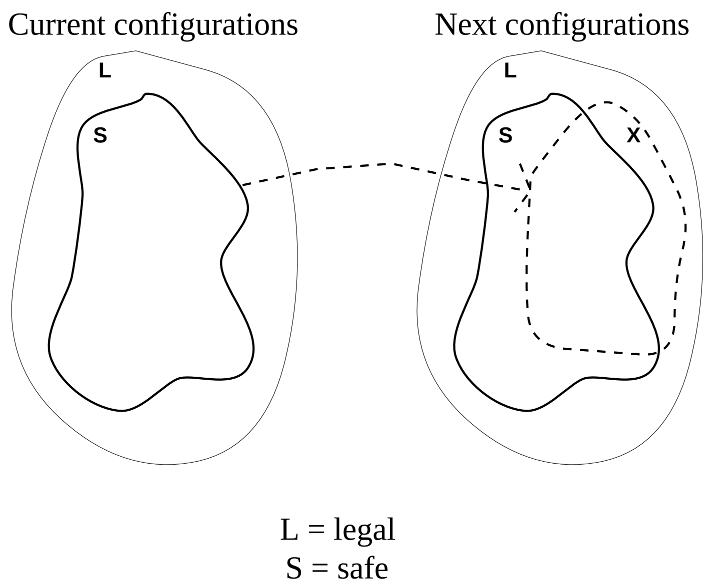

Figure 6.4: Why SAFE is not an invariant

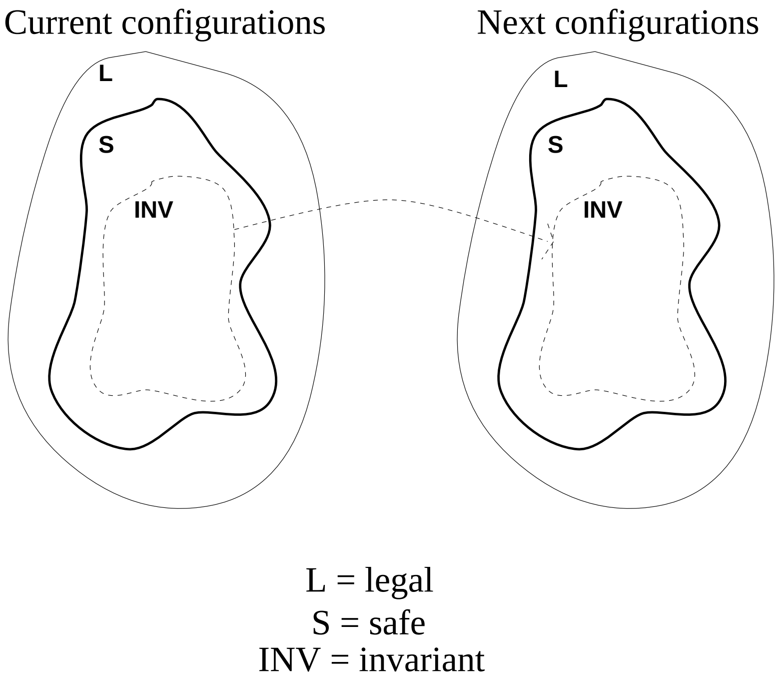

Figure 6.5: Invariants

In order to determine a possible candidate for INV we note that the property SAFE does not put any restrictions on the event counters that were triggering the transitions that led to an unsafe configuration as mentioned above. Changing the invariant to include the condition that if in the state **N_S_R** then the value of **EN_E_W_R** is greater than 0 means that transition **t2** can not be taken since the system has been in the state **E_W_R** for at least one step. Taking the disjunction of this condition and the corresponding one for **E_W_R** implies that the property SAFE will always be true. The system can still make progress because when it enters one of the red states, the disjuncts change as to which one is true:

$$
\begin{array}{l}
\mathsf{INV} \ =_{def} \\
\quad \lambda cf. \\
\quad\quad (\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf) \ \wedge \\
\quad\quad\quad \mathsf{SemGREATER}(\mathsf{SemVAR}\ \mathbf{EN\_E\_W\_R}, \mathsf{SemCONST}\ 0)\ cf) \ \ \vee \\
\quad\quad (\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{E\_W\_R}\ cf) \ \wedge \\
\quad\quad\quad \mathsf{SemGREATER}(\mathsf{SemVAR}\ \mathbf{EN\_N\_S\_R}, \mathsf{SemCONST}\ 0)\ cf) \ \ \vee \\
\quad\quad \mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{FL}\ cf) \wedge \\
\quad\quad \mathsf{STATE\_COND}\ newtls\ cf
\end{array}
$$

The property INV includes restrictions on the event counters. To use induction, we must ensure that the starting configuration also satisfies INV. Only the counters for the events of entering the default states should have the value 0. All other event counters should be greater than 0. This satisfies our invariant because,

$$
\begin{array}{l}
\mathsf{BOOL}(\mathsf{SemVAR}\ \mathbf{N\_S\_R}\ cf) \ \wedge \\
\mathsf{SemGREATER}(\mathsf{SemVAR}\ \mathbf{EN\_E\_W\_R}, \mathsf{SemCONST}\ 0)\ cf
\end{array}
$$

will be true. Therefore by induction, we have shown the invariant to be true for all reachable configurations of the model. Since the invariant implies SAFE, this property is always true for systems whose variables can take on the same range of values as those given in the variable records.

```
#let INV = new_definition(
    `INV`,
    "INV = \(cf:Config).
        ((BOOL(SemVAR `N_S_R` cf) /\
        BOOL(SemGREATER (SemVAR `EN_E_W_R`,SemCONST 0) cf)) \/
        (BOOL(SemVAR `E_W_R` cf) /\
        BOOL(SemGREATER (SemVAR `EN_N_S_R`,SemCONST 0) cf)) \/
        BOOL(SemVAR `FL` cf)) /\
        STATE_COND newtls cf");;
########INV =
|- INV =
   (\cf.
     (BOOL(SemVAR `N_S_R` cf) /\
      BOOL(SemGREATER(SemVAR `EN_E_W_R`,SemCONST 0)cf) \/
      BOOL(SemVAR `E_W_R` cf) /\
      BOOL(SemGREATER(SemVAR `EN_N_S_R`,SemCONST 0)cf) \/
      BOOL(SemVAR `FL` cf)) /\
     STATE_COND newtls cf)

VOSS "MC MC_A_NS 1 INV (NC newtls) INV newtlsINFO";;
`T` : string
```

Figure 6.6: Checking the correct invariant for the traffic light

6.8 Efficiency

Executing the model checker can take a long time (10-15 minutes real-time) on machines with lots of memory (48 meg) even for small problems. The number of steps affect the execution time and proving an invariant over one step can be fairly quick.

Running the model checker is in effect executing the operational semantics given in the previous chapter. These functions were not written with efficiency as a priority, therefore there is room for reworking them to speed up the execution of the model checker.

Executing a Boolean expression in a BDD package can also be made more efficient by changing the variable ordering for the BDDs. Using Voss through HOL, we have not taken advantage of this ability. One can argue that for small problems, time for execution is not a problem and therefore the convenience gained from letting the package take care of this is well worthwhile. For larger problems, execution time may become more of a factor and this is another way the model checker could work faster.

We believe that there is room to exploit the hierarchy of statecharts to reduce the configuration space that must be explored to verify a property of the model. For example, it should be possible to ignore details of substates and transitions among substates if they are not needed for a particular property. This would reduce the size of the configuration that needs to be represented. Given the priority of transitions in the hierarchy, decomposing states should not violate a proven property.

6.9 Other Descriptive Specification Notations

Higher-order functions can be used to write more complex descriptive specifications for our model checker but these must be given relative to the current configuration only and within a bounded time. Properties that depend on future configurations as well as the current one can be expressed in formalisms used by other model checkers. Computational Tree Logic (CTL) is a branching temporal logic that has operators to express properties for all times, and paths [24]. For those familiar with CTL, the MC_A function is closest to the AF f operator of CTL which means *for all paths eventually* f, and correspondingly, MC_E is like EF f. The major difference is that CTL checks for all lengths of paths using a fixed point operator where as our model checking tests only within a restricted time. For expressing properties of hard real-time systems, bounded temporal operators should be sufficient.

CTL requires a more complex model checking algorithm than the one presented here. Section 9.5 will look at the possibility of using another notation called State Transition Assertions [6] to give more expressive descriptive specifications with only small changes in the model checking algorithm used here.

6.10 Conclusions

This chapter describes the language for descriptive specifications and its associated model checking algorithm. The configuration of the system is represented as Boolean values. The process is symbolic over the range of values that the bits can represent. We have taken advantage of BDDs for efficient representation of the configuration but the process may still suffer from the configuration explosion problem for large models.

If the result of checking a property using the model checking algorithm returns false then it is very useful to give an example showing the complete set of steps leading up to a configuration where the property does not hold. This scenario is called a counter-example. Currently, our model checker does not provide this.

Determining an invariant that limits the set of configurations enough to prove a property for all reachable configurations is not an easy process. The benefits are that the model checker only has to be run for one step. It would be useful to formalize the induction argument made using invariants. Section 9.4 describes how to do this within HOL.

The errors in the traffic light statechart can also be found using the reachability of conditions test in STATEMATE. In the next chapter, more examples of model checking are presented. The first of these demonstrates the use of symbolic values to prove a functional property of a system.

Chapter 7

Examples

This chapter presents two examples where the model checker has been used to prove properties about small systems. In each case, the operational and descriptive specifications of the model are given along with the output of a run of the tests. The first example demonstrates the use of symbolic values and how different information about the representation of the configuration can affect the results of the model checker.

7.1 Swap Operation

This first example demonstrates the use of symbolic values to prove the functionality of an operation. The swap operation just interchanges the values of two variables using one temporary value. The statechart describing its operation is given in Figure 7.1. The three actions are all placed on separate transitions so that they will happen sequentially. These transitions are enabled as soon as their source state is entered since there are no events to trigger them.

To verify that the model accomplishes the swap operation, we should prove that beginning from the starting system configuration, within three steps the model always results in a state where the values have been swapped. **VAR1** and **VAR2** are given the values of **X** and **Y** which can be constants or symbolic values depending on how they are set in the starting configuration. They must be considered part of the configuration so that they can be referred to in both the characteristic function for the initial configuration (START) and the property showing the operation has been accomplished (END). They are considered internal so that they can not change their values between steps. The definition of the statechart is given by the constant *swap* used in these definitions.

In our first test, we allocate one bit to all data items and symbolic values. The constant $swapINFO1$ holds the following information:

Variable Type	Name	# Bits	Internal/External
Basic states	**A**	1	Int
	B	1	Int
	C	1	Int
	D	1	Int
Data Items	**TEMP**	1	Int
	VAR1	1	Int
	VAR2	1	Int
Symbolic Values	**X**	1	Int
	Y	1	Int

The results of the test in Figure 7.2 show that after three steps the model successfully completes the operation.

To demonstrate the importance of the variable records, we can run the model checker with different sets of variable information. Making all the data-items and symbolic values two bits, the constant $swapINFO2$

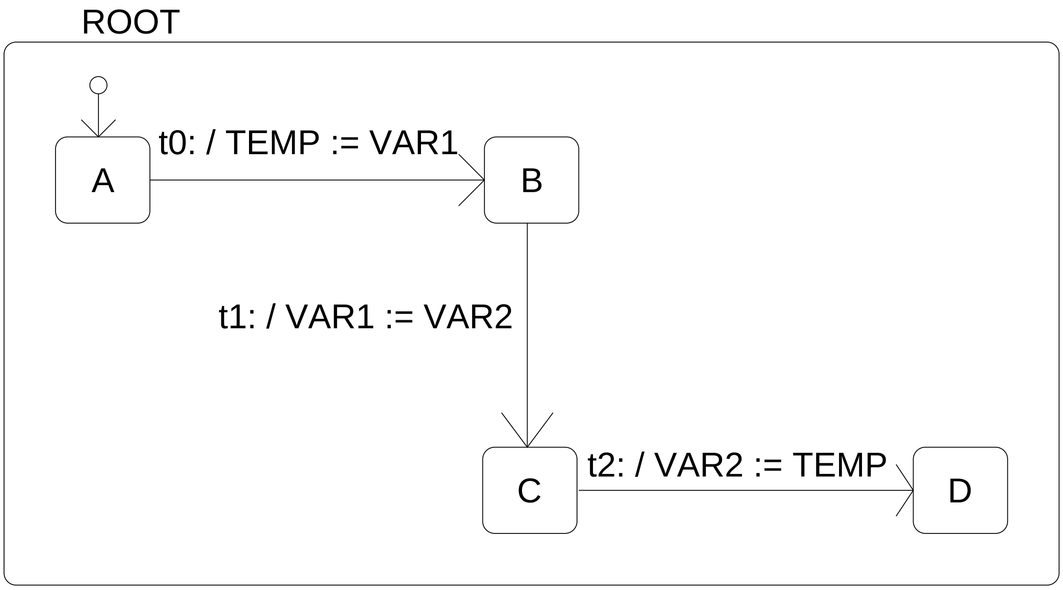

Figure 7.1: Swap operation

contains the following variable records:

Variable Type	Name	# Bits	Internal/External
Basic states	**A**	1	Int
	B	1	Int
	C	1	Int
	D	1	Int
Data Items	**TEMP**	2	Int
	VAR1	2	Int
	VAR2	2	Int
Symbolic values	**X**	2	Int
	Y	2	Int

Figure 7.3 shows the results of running the model checker with the revised variable records.

If **TEMP** has fewer bits than the other data-items then values will be lost in execution. For example, if **TEMP** is only one bit and the others are two bits. The following variable records are contained in $swapINFO3$:

Variable Type	Name	# Bits	Internal/External
Basic states	**A**	1	Int
	B	1	Int
	C	1	Int
	D	1	Int
Data Items	**TEMP**	1	Int
	VAR1	2	Int
	VAR2	2	Int
Symbolic values	**X**	2	Int
	Y	2	Int

The results of the model checker are shown in Figure 7.4.

If, however, the symbolic values have only one bit as well, then $swapINFO4$ contains the following information:

```
#let START = new_definition(`START`,
        "START = \cf.
                INITIAL swap cf /\
                BOOL(SemEQUAL(SemVAR `VAR1`,SemVAR `X`) cf) /\
                BOOL(SemEQUAL(SemVAR `VAR2`,SemVAR `Y`) cf) ");;

####START =
|- START =
   (\cf.
     INITIAL swap cf /\
     BOOL(SemEQUAL(SemVAR `VAR1`,SemVAR `X`)cf) /\
     BOOL(SemEQUAL(SemVAR `VAR2`,SemVAR `Y`)cf))

#let END = new_definition(`END`,
        "END = \cf.
                BOOL(SemVAR `D` cf) /\
                BOOL(SemEQUAL(SemVAR `VAR1`,SemVAR `Y`) cf) /\
                BOOL(SemEQUAL(SemVAR `VAR2`,SemVAR `X`) cf) ");;
####END =
|- END =
   (\cf.
     BOOL(SemVAR `D` cf) /\
     BOOL(SemEQUAL(SemVAR `VAR1`,SemVAR `Y`)cf) /\
     BOOL(SemEQUAL(SemVAR `VAR2`,SemVAR `X`)cf))

VOSS "MC MC_A_NS 1 START (NC swap) END swapINFO1";;
`F` : string

VOSS "MC MC_A_NS 2 START (NC swap) END swapINFO1";;
`F` : string

VOSS "MC MC_A_NS 3 START (NC swap) END swapINFO1";;
`T` : string
```

Figure 7.2: Swap test #1

```
VOSS "MC MC_A_NS 3 START (NC swap) END swapINFO2";;
`T` : string
```

Figure 7.3: Swap test #2

```
VOSS "MC MC_A_NS 3 START (NC swap) END swapINFO3";;
`F` : string
```

Figure 7.4: Swap test #3

```
VOSS "MC MC_A_NS 3 START (NC swap) END swapINFO4";;
`T` : string
```

Figure 7.5: Swap test #4

Variable Type	Name	# Bits	Internal/External
Basic states	**A**	1	Int
	B	1	Int
	C	1	Int
	D	1	Int
Data Items	**TEMP**	1	Int
	VAR1	2	Int
	VAR2	2	Int
Symbolic values	**X**	1	Int
	Y	1	Int

This time no information is lost and the tests are successful (Figure 7.5).

7.2 Arbiter

The second example is a model of one node in an arbiter circuit described by Staunstrup and Greenstreet [29]. The complete arbiter is a binary tree where one token is passed down to a leaf node that is the only one to have access to the shared resource. Each node has a request (**REQP**) and grant (**GRP**) signal connected to its parent and the same signals for each of its children (**REQL**, **GRL**, **REQR**, and **GRR**). The node sets **REQP** to true to request the token. It receives the token when **GRP** becomes true. It can then pass it to one of its children and the left child has precedence over the right. It returns the token to its parent by setting **REQP** to false and can again make a request once **GRP** becomes false. The signals **GRP**, **REQR**, and **REQL** are external to this node. However once the token has been granted **GRP** must remain true.

Staunstrup and Greenstreet model the node using Synchronized Transitions which is a notation that describes a program or circuit by a set of guarded transitions operating atomically but in any order. Their specification can be directly translated into a statechart by having transitions loop around one state. These transitions all have equal priority so any one of them could be taken providing its trigger is satisfied. Figure 7.6 gives the statechart operational specification for the model.[1] The signal **GRP** is modeled using an extra component since the signal is set to true outside of this arbiter. This action is caused by the external event **SET** which would be generated by a parent node following its **t1** or **t2** transition. Once the token is given to a node, it can not be taken away until **REQP** is false. Other transitions rely on knowing that **GRP** does not change its value from true. Making **GRP** internal ensures that it maintains its value unless directly changed within the statechart. In this way part of the environment is included as a concurrent component of the arbiter statechart.

Informally, the transitions in the statechart, describe the following operations:

- **t0** ask for the token
- **t1** pass the token to the left child
- **t2** pass the token to the right child
- **t3** get the token back from the left child, if there is no longer a request for it
- **t4** get the token back from the right child, if there is no longer a request for it
- **t5** pass the token back up the tree since no children want it

These transitions correspond directly to the synchronized transition description of the arbiter.

The variable records for this model ($arbiter1INFO$) gives each signal one bit:

[1] **tr!** and **fs!** are short for **MAKE_TRUE** and **MAKE_FALSE** respectively.

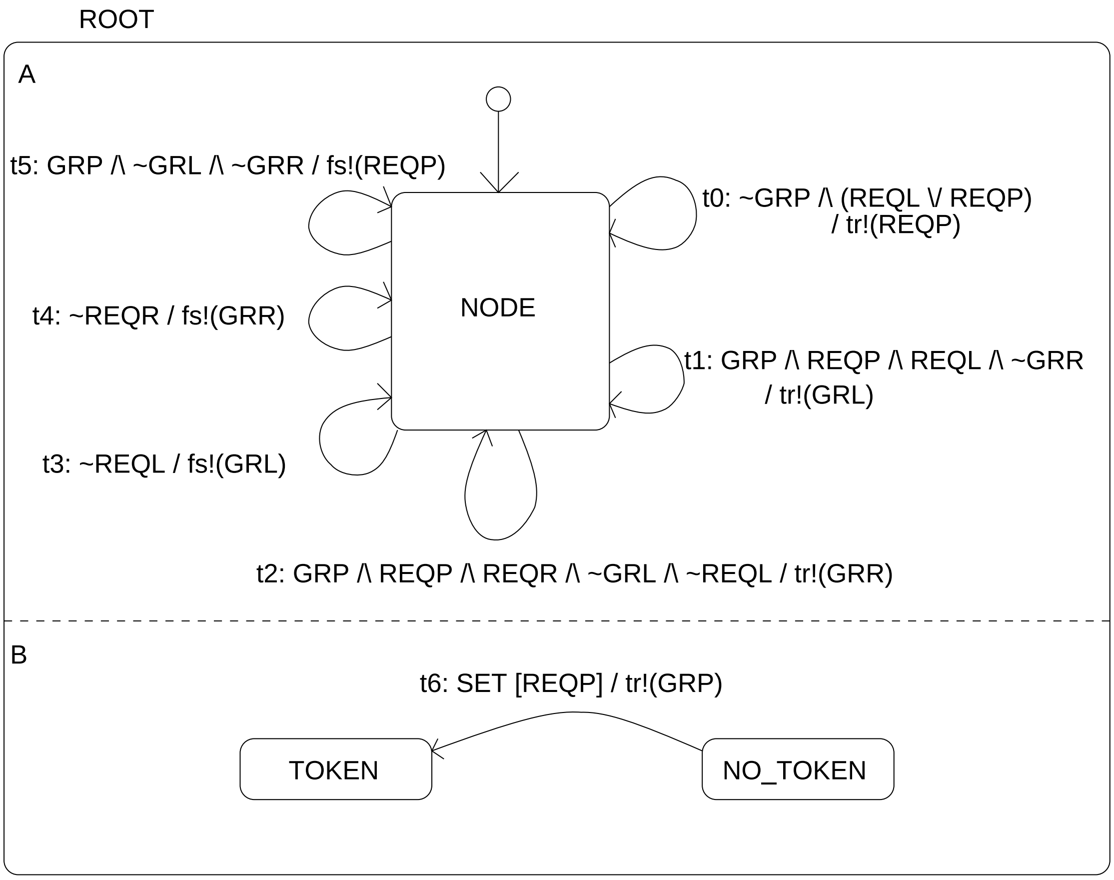

Figure 7.6: Arbiter statechart

```
         _  _      __     _        __    __
|___    |__|    |  |    |        |__|  |__|
|       |  |    |__|    |__      |__|  |__|

        Version 2.0, built on 15/7/92


    .... load definitions for semantics, model checking, arbiter1


#let INV = new_definition(
    `INV`,
    "INV = \cf.
        ~(BOOL (SemVAR `GRL` cf) /\ BOOL(SemVAR `GRR` cf))");;
###INV = |- INV = (\cf. ~(BOOL(SemVAR `GRL` cf) /\ BOOL(SemVAR `GRR` cf)))

VOSS "MC MC_A_NS 1 INV (NC arbiter1) INV arbiter1INFO";;
`T` : string
```

Figure 7.7: Model checking the arbiter invariant

Basic states	**TOKEN**	1	Int
	NO_TOKEN	1	Int
	CIRCUIT	1	Int
Data items	**GRP**	1	Int
	REQP	1	Int
	GRR	1	Int
	GRL	1	Int
	REQR	1	Ext
	REQL	1	Ext
Events	**SET**	1	Ext
Event counters	**EV_SET**	1	Ext

It is important that the arbiter does not grant the token to both of its children at the same time because this would violate exclusive access to the shared resource. The property that we would like to prove is:

$$\lambda cf. \neg(\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRL}\,cf) \;\wedge\; \mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRR}\,cf))$$

Figure 7.7 shows the results of proving the induction step for this invariant. For the initial configuration of the statechart, the signals would have to be set appropriately to satisfy the invariant.

7.3 Sequential Arbiter

The previous model for an arbiter was developed directly from the synchronized transition program so it was not necessary to study in detail the meaning of the transitions. One result of executing all transitions in parallel is that the triggers must be very detailed to ensure that operations progress in order. For the arbiter, an alternate, more sequential model was developed to understand its operation. This statechart is presented in Figure 7.8. In this model the state names give some mnemonic information about what the node is currently attempting to do:

IDLE: the node does not have the token and is not currently asking for it

REQ_IN: the node is asking for the token

TOKEN: the node has the token

LEFT: the left child has the token

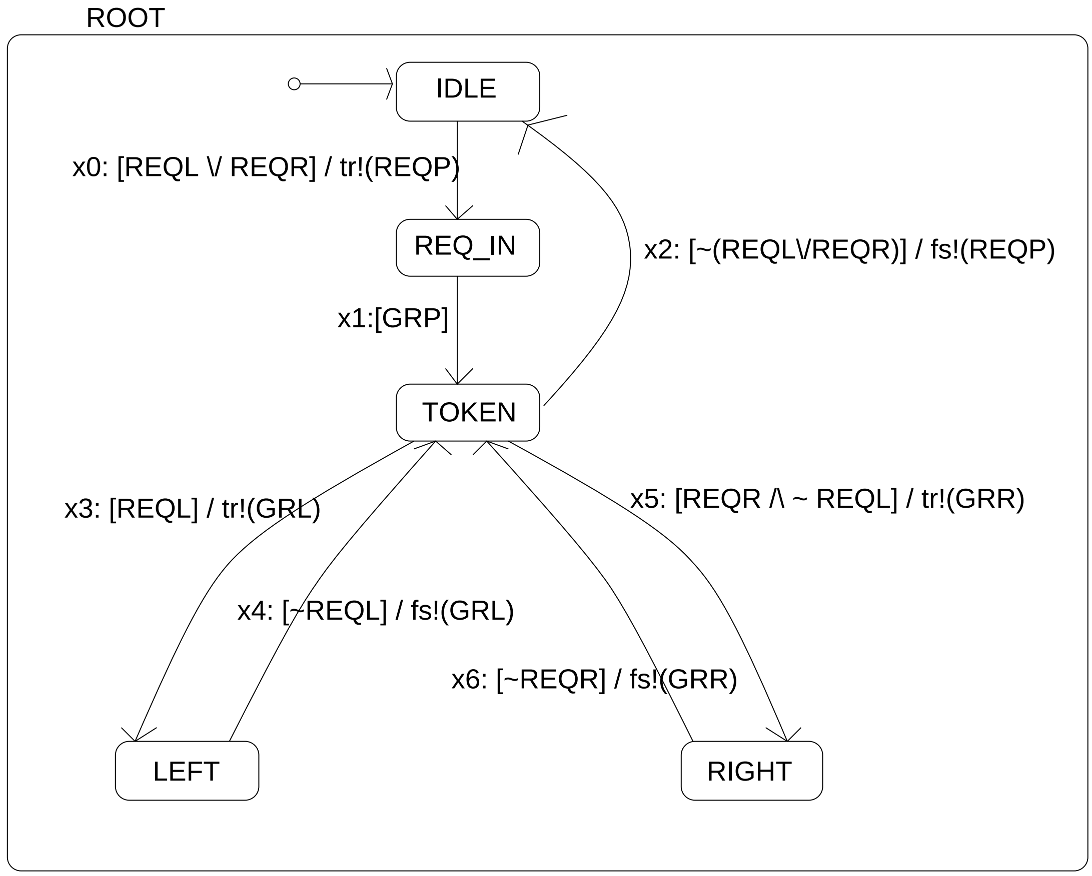

Figure 7.8: Sequential arbiter statechart

```
 |___    |__|    |‾‾|    |       |__|  |__|
 |       |  |    |__|    |__     |__|  |__|

         Version 2.0, built on 15/7/92


     .... load definitions for semantics, model checking, arbiter2


#let INV = new_definition(
    `INV`,
    "INV = \cf.
        ~(BOOL (SemVAR `GRL` cf) /\ BOOL(SemVAR `GRR` cf))");;
###INV = |- INV = (\cf. ~(BOOL(SemVAR `GRL` cf) /\ BOOL(SemVAR `GRR` cf)))

VOSS "MC MC_A_NS 1 INV (NC arbiter2) INV arbiter2INFO";;
`F` : string
```

Figure 7.9: Checking the invariant in the sequential arbiter

RIGHT: the right child has the token

The intent is that this model describes the same operation as the one in Figure 7.6. We will not attempt to verify that the functionality of these two models is the same but we can verify that the same invariant holds.

Slightly different bit vector information is used for this second model ($arbiter2INFO$). **GRP** can be considered external since it is not referenced again once it has been granted and we can just assume that the parent node of this one changes **GRP** just as this node changes **GRR** and **GRL**.

Basic states	**TOKEN**	1	Int
	NO_TOKEN	1	Int
	CIRCUIT	1	Int
Data items	**GRP**	1	Ext
	REQP	1	Int
	GRR	1	Int
	GRL	1	Int
	REQR	1	Ext
	REQL	1	Ext

Figure 7.9 shows that the same invariant does not hold. As with the traffic light, this does not necessarily mean that the invariant does not hold of the reachable configurations. We need to strengthen the invariant to limit the set of configurations under consideration.

This model assumes certain information once a particular state has been reached. This means the triggers on the transitions are much less complicated. For example,

$$\textbf{x3: [REQL] / tr!(GRL)}$$

corresponds to,

$$\textbf{t1: GRP} \wedge \textbf{REQP} \wedge \textbf{REQL} \wedge \neg\textbf{GRR / tr!(GRL)}$$

in the first arbiter where the token is passed to the left child. Given that we are in the state **TOKEN**, we can assume that the signals **GRP**, **REQP** are true and **GRR** is false.

These assumptions need to be built into the invariant. We can rephrase the property depending on what state the system is in:

$$\begin{array}{l} \mathsf{INV2} =_{def} \\ \quad \lambda cf. \\ \quad\quad (\mathsf{EXISTS}\,(\lambda v.\, \mathsf{BOOL}(\mathsf{SemVAR}\; v\; cf))\,[\textbf{REQ\_IN}; \textbf{IDLE}; \textbf{TOKEN}] \end{array}$$

```
#let INV2 = new_definition(`INV2`,
        "INV2 = \cf.
        ( (EXISTS (\v. BOOL(SemVAR v cf)) [`REQ_IN`;`IDLE`;`TOKEN`])
              ==> (~BOOL(SemVAR `GRR` cf) /\ ~BOOL(SemVAR `GRL` cf))) /\
        ( BOOL(SemVAR `LEFT` cf) ==> ~BOOL(SemVAR `GRR` cf)) /\
        ( BOOL(SemVAR `RIGHT` cf) ==> ~BOOL(SemVAR `GRL` cf)) /\
###        STATE_COND arbiter2 cf");;
###INV2 =
|- INV2 =
   (\cf.
     (EXISTS(\v. BOOL(SemVAR v cf))[`REQ_IN`;`IDLE`;`TOKEN`] ==>
      ~BOOL(SemVAR `GRR` cf) /\ ~BOOL(SemVAR `GRL` cf)) /\
     (BOOL(SemVAR `LEFT` cf) ==> ~BOOL(SemVAR `GRR` cf)) /\
     (BOOL(SemVAR `RIGHT` cf) ==> ~BOOL(SemVAR `GRL` cf)) /\
     STATE_COND arbiter2 cf)

VOSS "MC MC_A_NS 1 INV2 (NC arbiter2) INV2 arbiter2INFO";;
`T` : string
```

Figure 7.10: Revised invariant for the sequential arbiter

$$\Longrightarrow \neg\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRR}\,cf) \wedge \neg\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRL}\,cf)) \wedge$$
$$(\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{LEFT}\,cf) \Longrightarrow \neg\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRR}\,cf)) \wedge$$
$$(\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{RIGHT}\,cf) \Longrightarrow \neg\mathsf{BOOL}(\mathsf{SemVAR}\,\mathbf{GRL}\,cf))$$

Figure 7.10 shows the result of checking this invariant. Since all possible states are covered in this descriptive specification, it implies the previous invariant and therefore both models satisfy the property.

7.4 Conclusions

The swap example demonstrated the symbolic verification of a simple functional property. If the operation took many steps, then it could take the model checker quite a long time to check the property. We also showed how the ranges of the values of variables are an important part of the specification of the system. The user has to be aware that the property verified only holds for a configuration that can be described in the given number of bits.

The arbiter demonstrated how statecharts can also be used to model hardware. In this example, we directly translated a Synchronized Transition program into a statechart and verified an invariant automatically using the model checker. There are interesting possibilities for using this technique to model check parallel programs.

The sequential arbiter demonstrated that by using the sequential aspects of statechart the operational specification may be more understandable. It may be more difficult to determine the invariant of a sequential program since certain aspects of the computation are assumed to have happened once a particular state has been reached.

We have only proved an invariant for one node in the arbiter circuit. It would be very useful if the result from one node could be combined with other nodes of the same structure to prove that the token only belongs to one node in the complete tree at any one time. McMillan looks at doing inductive proofs over the structure of a system in the model checker SMV [24].

Chapter 8

Implementation

This chapter presents a textual representation for statecharts as an implementation of the abstract data type given in Chapter 3. The software interface that extracts information about the statechart directly from the STATEMATE database and produces this textual representation is described. The final sections discuss embedding the semantic functions in HOL and the tool that extracts information about the variables used in a particular statechart.

8.1 Introduction

This chapter presents the concrete details of the representation used for the statecharts and the implementation of the semantic functions in HOL. This information is not necessary to understand the main ideas presented in this work.

8.2 Textual Representation of Statecharts

The abstract data type presented in Chapter 3 can be given a textual representation. Textual representations for more complicated languages like hybrid statecharts have been given previously [19] but these contain more detail than is necessary here.

The following sections will present the concrete syntax for transitions and states which together create a statechart. The names of all elements of the configuration are represented as strings. These are written as a word in single quotes, as in ‘X‘ for the variable X.

8.2.1 Transitions

Transitions are given unique numeric names and are made up of source and destination state names, a triggering event, and an action. They have the type:

$$\begin{array}{ccccccccccc} \text{Trans} & \equiv & \text{Num} & \# & \text{Variable} & \# & \text{Event} & \# & \text{Action} & \# & \text{Variable} \\ & & \uparrow & & \uparrow & & & & & & \uparrow \\ & & \text{name} & & \text{source} & & & & & & \text{destination} \\ & & & & \text{state name} & & & & & & \text{state name} \end{array}$$

For example, the transition originated in N_S.RED from Figure 2.1 looks like:

(2,‘N_S.RED‘,(EVEXPR NONE (IN ‘E_W.RED‘)), NILL, ‘N_S.GREEN‘)

8.2.2 States

Every state has the type basic, AND, or OR.

$$\text{Typ} \equiv \text{B} \mid \text{A} \mid \text{O}$$

A state consists of its name, type, the name of its default state, a list of its immediate substates, and a list of transitions that originate at its substates.

All information about a state can be grouped together as an element with type:

$$\begin{array}{ccccccccccc}
\text{State} & \equiv & \text{Variable} & \# & \text{Typ} & \# & \text{Variable} & \# & \text{(Variable)} & \# & \text{(Trans)} \\
 & & & & & & & & \text{list} & & \text{list} \\
 & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\
 & & \text{state} & & \text{B}|\text{A}|\text{O} & & \text{default} & & \text{substates} & & \text{transitions} \\
 & & \text{name} & & & & \text{state} & & & &
\end{array}$$

8.2.3 The Complete Statechart

A statechart is a list of states. The complete model begins at the highest level state which is assumed to be the first element in the list. The order in the rest of list is irrelevant since the hierarchy is given through the lists of substates.

$$\text{Sc} \equiv \text{(State) list}$$

Using abbreviated names for some of the states, the textual representation for the traffic light example of Figure 2.1 is:

```
[
(`ROOT`, O, `N`, [`N`; `FL`],
        [(7,`FL`,SE (EV `RESET` `EV_RESET`), NILL, `N`);
        (6,`N`, SE (EV `MALF` `EV_MALF`), NILL, `FL`)]);
(`N`, A, ``, [`N_S`; `E_W`], []);
(`N_S`, O, `N_S_G`, [`N_S_G`; `N_S_R`; `N_S_Y`],
        [(0,`N_S_G`, (TM (EN `N_S_G` `EN_N_S_G`) (VAR `NS_G_T`)), NILL, `N_S_Y`);
        (1,`N_S_Y`, (TM (EN `N_S_Y` `EN_N_S_Y`) (CONST 2)), NILL, `N_S_R`);
        (2,`N_S_R`,(EVEXPR NONE (IN `E_W_R`) ), NILL, `N_S_G`)]);
(`N_S_G`, B, ``,[],[]);
(`N_S_R`, B, ``,[],[]);
(`N_S_Y`, B, ``,[],[]);
(`E_W`, O, `E_W_R`, [`E_W_G`; `E_W_Y`; `E_W_R`],
        [(3,`E_W_G`, (TM (EN `E_W_G` `EN_E_W_G`) (VAR `EW_G_T`)), NILL, `E_W_Y`);
        (4,`E_W_Y`, (TM (EN `E_W_Y` `EN_E_W_Y`) (CONST 2)), NILL, `E_W_R`);
        (5,`E_W_R`,(EVEXPR NONE (IN `N_S_R`)), NILL, `E_W_G`)]);
(`E_W_G`, B, ``,[],[]);
(`E_W_Y`, B, ``,[],[]);
(`E_W_R`, B, ``,[],[]);
(`FL`, B, ``,[],[]);
]
```

8.3 Selector Functions

The selector functions hide the concrete details of the implementation of the ADT. In this section, we give definitions for the functions described in Chapter 3.

Given an element of type Trans, the following selectors may be defined:

$$\begin{array}{rcl}
\mathsf{LABEL}\,(tr : \mathsf{Trans}) & =_{def} & \mathsf{FST}\, tr \\
\mathsf{SRC}\,(tr : \mathsf{Trans}) & =_{def} & \mathsf{FST}\,(\mathsf{SND}\, tr) \\
\mathsf{EVENT}\,(tr : \mathsf{Trans}) & =_{def} & \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\, tr)) \\
\mathsf{ACTION}\,(tr : \mathsf{Trans}) & =_{def} & \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\, tr))) \\
\mathsf{DEST}\,(tr : \mathsf{Trans}) & =_{def} & \mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\, tr)))
\end{array}$$

The remaining functions isolate parts of the complete statechart sc. The first function is used only by the selector functions to return all the information about a state given its name, $stname$:

$$\begin{array}{l} \mathsf{FMEMBER}\, sc\; stname =_{def} \\ \quad (\, sc = [\,]) \rightarrow [\,] \;\mid\; (\,(\mathsf{FST}\,(\mathsf{HD}\; sc) = stname) \rightarrow [\,\mathsf{HD}\; sc\,] \mid \mathsf{FMEMBER}\,(\mathsf{TL}\; sc)\; stname) \end{array}$$

The following functions return the type, default state name, substates, and transition list of a state, given its name. If the state name is not found in the statechart, they return the null values NOTYP, NOSTATE, NOSUBSTATES, and NOTRANS respectively.

$$\begin{array}{l} \mathsf{TYP}\; sc\; stname =_{def} \\ \quad \textbf{let}\;\; state = \mathsf{FMEMBER}\;\; sc\;\; stname \;\; \textbf{in} \\ \quad (state = [\,]) \rightarrow \mathsf{NOTYP} \mid \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{HD}\; state)) \end{array}$$

$$\begin{array}{l} \mathsf{DEFAULT}\; sc\; stname =_{def} \\ \quad \textbf{let}\;\; state = \mathsf{FMEMBER}\;\; sc\;\; stname \;\; \textbf{in} \\ \quad (state = [\,]) \rightarrow \mathsf{NOSTATE} \mid \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{HD}\; state))) \end{array}$$

$$\begin{array}{l} \mathsf{SUBSTATES}\; sc\; stname =_{def} \\ \quad \textbf{let}\;\; state = \mathsf{FMEMBER}\;\; sc\;\; stname \;\; \textbf{in} \\ \quad (state = [\,]) \rightarrow \mathsf{NOSUBSTATES} \mid \mathsf{FST}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{HD}\; state)))) \end{array}$$

Beginning from the state representing the complete system, usually called 'ROOT', and using the above function, it is possible to traverse the complete hierarchy of states.

$$\begin{array}{l} \mathsf{TRANSOFSTATE}\; sc\; stname =_{def} \\ \quad \textbf{let}\;\; state = \mathsf{FMEMBER}\;\; sc\;\; stname \;\; \textbf{in} \\ \quad (state = \mathsf{TRANS}) \rightarrow \mathsf{NOTRANS} \mid \mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{HD}\; state)))) \end{array}$$

The functions ROOT and GET_TRANS_NAMES are defined as:

$$\mathsf{ROOT}\; sc \quad =_{def} \quad \mathsf{FST}\,(\mathsf{HD}\; sc)$$

$$\begin{array}{l} \mathsf{GET\_TRANS\_NAMES}\; sc =_{def} \\ \quad (sc = [\,]) \rightarrow [\,] \mid \\ \quad \mathsf{APPEND}\,(\mathsf{MAP\;NAME}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{HD}\; sc)))))) \\ \quad (\mathsf{GET\_TRANS\_NAMES}\,(\mathsf{TL}\;\; sc)) \end{array}$$

The function TRAN which returns a transition associated with a particular name also has to traverse the complete statechart. The following two functions are used to do this. The second one collects all the transitions in the statechart and then the first one goes through the list to find the appropriate one.

$$\begin{array}{l} \mathsf{TRAN\_AUX}\; sc\;\; tname =_{def} \\ \quad (sc = [\,]) \Rightarrow (\mathsf{NOLABEL}, \mathsf{NOSTATE}, \mathsf{NONE}, \mathsf{NILL}, \mathsf{NOSTATE}) \mid \\ \quad (\mathsf{FST}\,(\mathsf{HD}\; sc) = tname) \rightarrow (\mathsf{HD}\; sc) \mid \mathsf{TRAN\_AUX}\,(\mathsf{TL}\; sc)\; tname \end{array}$$

$\mathsf{TRAN}\ sc\ tname =_{def}$
$\quad$ **let** $translist = \mathsf{FLAT}\,(\mathsf{MAP}\,(\lambda state.\,\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\,(\mathsf{SND}\ state))))\ sc)$ **in**
$\quad \mathsf{TRAN\_AUX}\ translist\ \ tname$

Finally, the selector function which returns the names of all transitions originating at descendants of a particular state also traverses the hierarchy of states and makes use of previously defined functions for getting the transitions of each state.

$$
\begin{array}{l}
\mathsf{GET\_TRANS\_STATE}\ sc\ stname =_{def} \\
\quad (\mathsf{TYP}\ sc\ stname = [\,]) \Rightarrow [\,]\ | \\
\quad \mathsf{APPEND}(\mathsf{MAP}\ \mathsf{NAME}\ (\mathsf{TRANSOFSTATE}\ sc\ stname)) \\
\quad (\mathsf{FLAT}(\mathsf{MAP}(\mathsf{GET\_TRANS\_STATE}\ sc)(\mathsf{SUBSTATES}\ sc\ stname)))
\end{array}
$$

These are not the most efficient implementations of these functions. There is potential for increasing the speed of the model checker by optimizing these functions.

8.4 Translation Process

STATEMATE provides a set of C functions to access its database and extract information about the states and the transitions of the model [16]. The textual representation for a statechart is generated automatically by a program that uses these functions to access the STATEMATE database, replaces symbols with words (like **PLUS** for '+'), and organizes the information in the above format. This program was written using the parsing tools Lex and Yacc [17][20]. The event counters are generated by concatenating the name of the event (EN, EV, etc) with the state name or variable for the event.

The output from this translation program is a file of ML code which includes a statement that creates a HOL definition for a statechart in the above syntax. This can be given directly as an argument to the semantic functions. This process does not check for completeness of the statechart at all. We rely on STATEMATE to ensure this before the textual representation is extracted from its database.

8.5 Embedding the Semantics in HOL

Most of the semantic definitions can be input directly into HOL, but in a few cases recursive definitions over the hierarchy of the statechart are used. Since HOL does not support general recursion, we have used primitive recursion. It is possible to do this by providing an extra argument to recurse over that initially is the length of the list of states in the statechart. This is an upper bound on the recursion since the statechart hierarchy would have to be a degenerate tree to reach this bound. Definitions like **INSTATE** become:

$$
\begin{array}{l}
(\mathsf{INSTATE}\ 0\ sc\ cf\ \ stname = \mathsf{F})\ \wedge \\
(\mathsf{INSTATE}\,(\mathsf{SUC}\,n)\ sc\ cf\ \ stname = \\
\quad ((\mathsf{TYP}\ sc\ stn = \mathsf{B}) \wedge \mathsf{BOOL}(\mathsf{SemVAR}\ stname\ \ cf)) \vee \\
\quad ((\mathsf{TYP}\ sc\ stname = \mathsf{A}) \wedge \mathsf{EVERY}\,(\mathsf{INSTATE}\ n\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stname)) \vee \\
\quad ((\mathsf{TYP}\ sc\ stname = \mathsf{O}) \wedge \mathsf{EXISTS}\,(\mathsf{INSTATE}\ n\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stname))
\end{array}
$$

8.6 Determining Configuration Information

A tool has been written in a combination of HOL and ML that parses a given statechart and determines: the list of all variables (including state names), whether they are external or internal, and allocates one bit to each one. It sets this information up in the format expected by the model checker as a list of records. This information is then used by the MEM configuration function to return the bits associated with a particular variable. The model checking function uses the status of the variable as internal or external to determine the set of internal variables that must be given as an argument to the next configuration relation.

In HOL, we use recursively defined functions to check each expression, event, and action of all transitions in the statechart for variables. By default these variables are considered external unless they are modified in some action, in which case, they are then considered internal. All basic states are considered internal.

These HOL definitions are "executed" using a rewriting conversion. The list produced may have some duplicates in it so a small amount of ML code is used to eliminate the duplicates for efficiency but this could be implemented in HOL. The result is a definition containing a list of variable records which can be modified by the user before it is supplied as a parameter to the model checking function.

Chapter 9

Future Work

This chapter looks at possible extensions to the semantics of statecharts and other uses for the model checker. The first sections cover elements of statecharts that are not included in our syntax. We also discuss situations where destination states of transitions conflict. These violate our assumption about the syntactic form of the statechart and require a revision to the transition condition of the semantics. The final two sections describe possible uses for the model checker.

9.1 The Complete Statechart Notation

In order to justify our claim that we have formalized the semantics for statecharts and not just a subset of statecharts, we will briefly explain how elements of statecharts found in STATEMATE that we have not included can be formalized within this framework.

The first three sections on static reactions, conditional connectives, and scheduled actions demonstrate how these elements can be written in the subset of statecharts that we have formalized and therefore they require no changes to the semantics. These are followed by discussions about elements that would require changes to the semantics.

9.1.1 Static Reactions

Static reactions are event and action pairs associated with a particular state that are examined in each step provided that the state is not exited. If their event has occurred then the associated action is carried out, even if transitions take place among substates of the state.

It is difficult to represent these graphically and they would require a new type of element to be introduced into the statechart notation since they are considered separately from transitions. We can produce the same effect by using a concurrent component that has the static reactions as loops around a state. Providing the system stays within the AND-state, these can be executed in parallel with the transitions in the other component.

9.1.2 Conditional Connectives

A useful feature of the statechart notation that we have not considered are transitions joined by connectors. Conditional connectives, as seen in Figure 9.1, are a way of decomposing transitions into smaller parts when multiple transitions share the same event but perhaps have different conditions in their label. The intention is that only one path from the source to the destination will be followed. In STATEMATE [14], one of the transitions leaving the connector must be enabled or else it is considered an error when executing the transition. We can construct separate transitions for each path of the connected transition thereby giving an interpretation for these error situations because no transition will be taken if none are enabled. The statechart given in Figure 9.1 would be equivalent to the one in Figure 9.2.

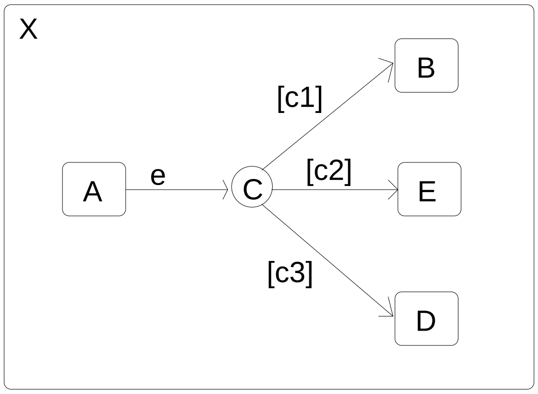

Figure 9.1: Conditional connectives

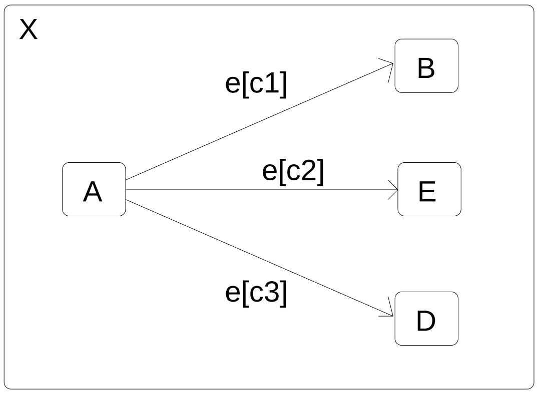

Figure 9.2: Removing conditional connectives

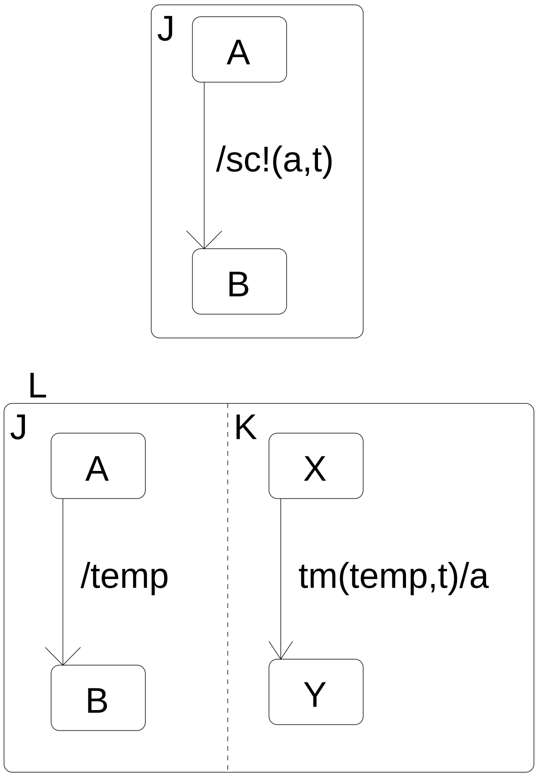

Figure 9.3: Scheduled actions

9.1.3 Scheduled Actions

In STATEMATE, the possible set of actions include $schedule(act, t)$ which means the action *act* will be performed t steps from now. There is no explicit provision within our semantics for scheduling events since the only values manipulated are those of the current and next configurations. One can imagine that there might be a transition taken in the step where the action is to be performed, that affects the same variable as the scheduled action. This would have to be resolved in the way that conflicting actions are resolved for the present step.

Rather than trying to determine a way to include an action like this in our syntax, we can write a statechart that will accomplish the same task by generating an event on the transition whose label had the *schedule* action. This event is the timeout event for a transition in an orthogonal component. The timeout step number is how many steps are to occur before the scheduled action. The action of this transition is the scheduled action. Figure 9.3 shows two corresponding statecharts using this method.

9.1.4 History

Statecharts often include history connectors, marked with an **H** which can also be the destination of a transition. It dynamically represents the substate this state was in at the time it was last exited. If there is no history (i.e. the system has never been in this state before or the history has been cleared) then a transition from the history connector is followed if it exists or else the default transition is taken.

As it currently stands, the functions ENTERDEST and EXITSRC always return exactly the same information for a given transition. With history states the values returned by these functions would change depending on the configuration at the time they were called.

Initially, we tried representing the state information in the configuration using one variable for each OR-state whose value gives the current substate. The advantage in this approach is that only one value could ever be assigned to this variable at any point in time, which automatically creates the exclusive-OR

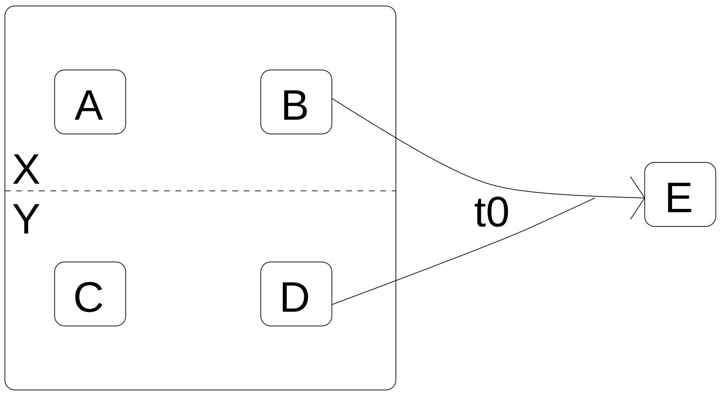

Figure 9.4: Transition with multiple source states

of substates. This method was not used because everything has to be converted to an underlying Boolean representation so the state names would have to be encoded. This makes it more difficult to determine if a condition like IN(*statename*) is true.

This encoding of states could be used to incorporate history states into our semantics. A history connector, represented by a variable, could be the destination of a transition and functions could interpret the value of this variable in terms of the basic states that should be entered. Provisions would have to made for updating these history variables when states are exited.

Not surprisingly, the use of history connectors can create some interesting situations which the semantics must be able to handle. A description of some of these can be found in the STATEMATE manual [13].

9.1.5 Transitions with Multiple Sources and Destinations

Transitions with multiple sources (or destinations) must originate from (or lead to) states that are in orthogonal components as in Figure 9.4. A transition should exit (or enter) all of its source (or destination) states. Even though the source and destination of each transition can easily be extended to be sets of states within our semantics, these situations violate our assumption about the syntactical form of the statechart. The revisions proposed in Section 9.3 should make it possible to allow these transitions. Rules would have to be established about the priority of transitions with multiple source states.

9.1.6 Super-steps

For simplicity, our semantics do not incorporate the idea of super-steps, however, Pnueli and Shalev [26] present an example of a system where micro-steps and the associated distinction between external and internal events is very important. The relevant part of the example is a three bit counter, where as the first bit toggles its value based on a tick of a clock occurring, the effect should ripple through the other bits. This statechart is given in Figure 9.5 (from Figure 1 in [26]). The value returned by the counter is not accurate until the effects of a change in the bit have been passed along to the other bits. This can take a maximum of three steps. If the external clock is only incremented every three steps, then this may be acceptable.

The idea of a super-step could be added to our semantics using a super-step relation that enforces the NC relation between each step. A step still incorporates all the effects of assignments but the system repeats steps until no more transitions are enabled.

9.2 Useful Extensions to Statecharts

9.2.1 Communication

Leveson et al. [21] has pointed out in their work on the TCAS II system that individual components can be described using a formalism similar to statecharts, but the interface between components requires

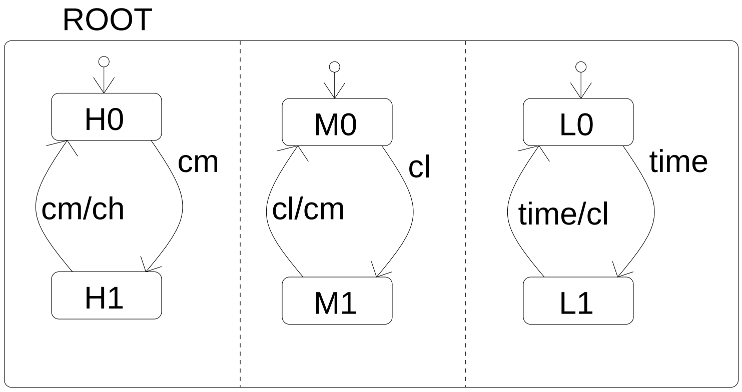

Figure 9.5: Statechart for a counter

more description. In particular, they chose to use point-to-point message communication between system components since this is a more realistic representation of the way the system actually operates [21]. Ways of integrating other communications methods into statecharts would be useful but would require additions to the semantics.

9.2.2 Continuous Systems

The process being controlled by a statechart may also have to be modeled to check properties of the overall system. This may be difficult because the physical process may operate in the continuous domain. Work is underway at both UBC and Cornell to develop interfaces with these types of models [1][22][30], and it will be interesting to see what role automatic verification tools can play in these hybrid domains.

9.2.3 Multiple Objects

Many components in a system may have the same behaviour. For example sensors used to monitor the position of a train on track can all be described by similar statecharts. Leveson et al. [21] describes statechart arrays which model this kind of situation. The statechart for a particular element is referenced by a unique number. Harel and Kahana [12] describe extensions to the semantics where statecharts could overlap. Another method is called Objectcharts [4] where the behaviour of a class of objects is given by a statechart. The CASE tool ObjecTime models system in a similar manner [25].

Extensions for multiple objects create interesting difficulties for the model checker. It might have to treat each object separately in the configuration leading to an explosion in the size of the configuration. McMillan looks at doing inductive proofs over the structure of the structure of a system in the model checker SMV [24] which could be a solution to this problem.

9.2.4 Dynamic Objects

There may also be systems where objects enter and leave the system dynamically and we only need to model their behaviour for a finite amount time. An example of this is modeling only a section of train track. Trains enter and leave the model and temporarily the system must be aware of their behaviour.

9.3 Conflicting Destination States

The semantics presented in Chapter 5 assume that there are no conflicts in the destination states of transitions chosen by the transition condition. While it is unusual to have destination conflicts occur, they are possible, especially when we begin to consider transitions with multiple source and destination states. This section describes the problem and outlines a possible way of expressing the statechart's behaviour in these situations. We outline how the semantic functions would be written to maintain the priority of transitions.

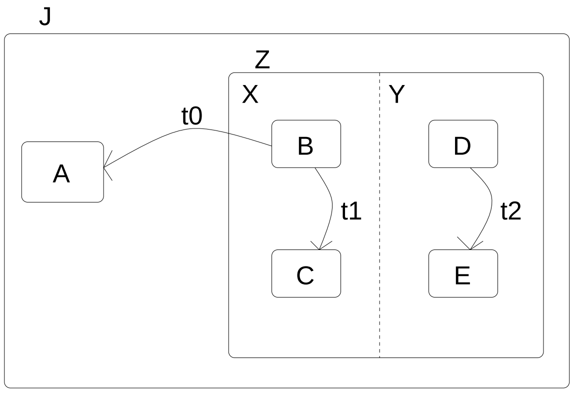

Figure 9.6: First example of destination conflicts

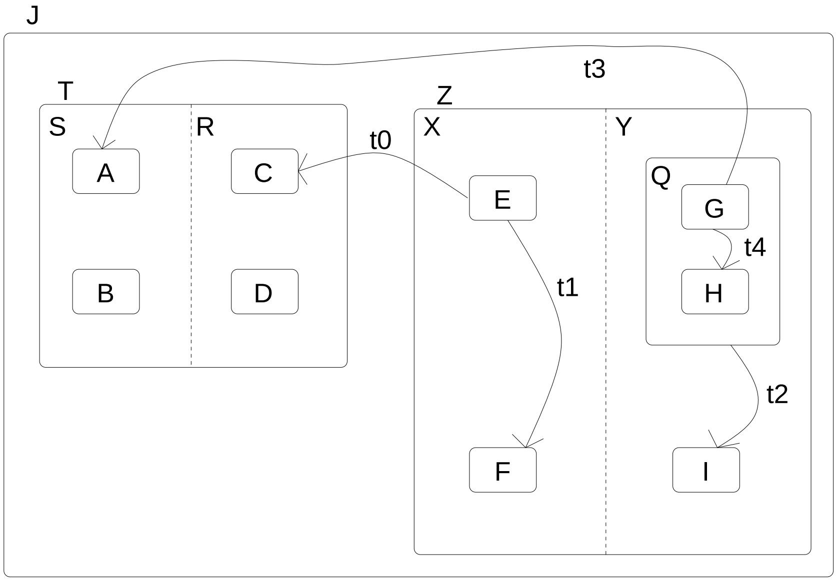

Figure 9.7: Second example of destination conflicts

In Figure 9.6, we can see that following transitions **t0** and **t2** moves the system to a configuration that includes states **A** and **E** which is not a legal state configuration, even though these transitions have source states in orthogonal components. Figure 9.7 presents a slightly more complicated example where following **t0** and **t3** will move the system to a legal state configuration but **t3** is at a lower level in the hierarchy than **t2** and **t2** may be enabled. The questions of priority also become important in these situations.

The semantics we presented in Chapter 5 do not check for destination conflicts and therefore decide on the set of transitions to follow based on those that are enabled, the AND-state hierarchy and the priority among OR states. For Figure 9.7 if all transitions are enabled, and the system is currently in the states **E** and **G**, the transition condition would return true for the sets of transition flags where **t0** and **t2** are true and also for **t1** and **t2**.

Without regard for priority, the sets of transitions that do not conflict are the following:

1. **t1**, **t2**
2. **t0**, **t3**
3. **t1**, **t4**

The approach taken by Pnueli and Shalev, and STATEMATE of only choosing transitions from orthogonal

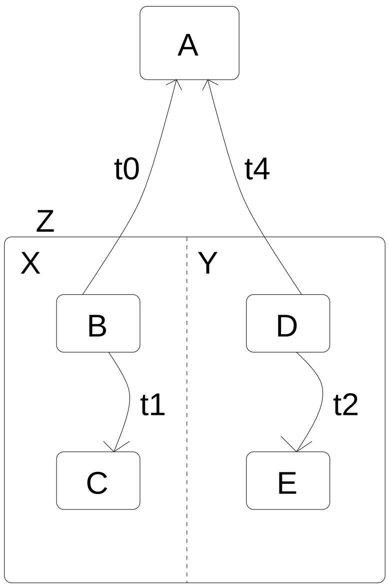

Figure 9.8: Non-conflicting destination states

scopes would never allow the set (**t0**, **t3**) since both these transitions have the same scope.

We can see that each orthogonal component can not be considered individually since the destinations of transitions chosen separately may conflict. Basing priority on source state, the set of transitions (**t1**, **t2**) should be chosen before (**t1**, **t4**) when **t2** is enabled. This is because the source of **t2** is an ancestor of the source of **t4** and the other transitions in each set are the same. The set (**t0**, **t3**) should have equal priority with the other two sets.

Pnueli and Shalev do not explicitly describe how priority of transitions is expressed in a statechart, however, they do discuss using negation of events. For example, if **t1**: a and **t2**: b are both enabled and **t1** is to have priority over **t2** then modifying the label of **t2** to be $b \wedge \neg a$ would be sufficient.

Their description of priority as the negation of events will not provide the type of priority outlined above when destination states conflict because the set (**t0**, **t3**) could never be chosen when **t2** is enabled because the trigger of **t3** would include the negation of the event enabling **t2**.

To rule out accepting transitions where the destination sets of transitions conflict but to maintain the priority based on source state, we can revise the transition condition given in Section 5.2 is the following ways:

- Include with the requirement that a transition be enabled when chosen (ENABLED) that its destination state (by following defaults) also be true in the next configuration.
- The STATE_COND must still hold so any transitions whose destination states conflict can not be chosen as part of the same set.
- Separate priority from the expression by removing the priority test. The transition condition in this form will admit all sets of enabled but non-conflicting transitions.
- To deal with priority, we can say that a given set of transitions is admissible if there is no way to set one transition flag to false and a flag for a transition of higher priority to true and still satisfy the transition condition. To test this, we must existentially quantify over the basic states of the next configuration.

Conflicting destination states are defined as those where having both of them assigned the value true in the next configuration will not satisfy the state condition. For example, in Figures 9.8, the destinations of **t0** and **t4** do not conflict even though they have the same destination state. In Figure 9.9, however, **t0** and

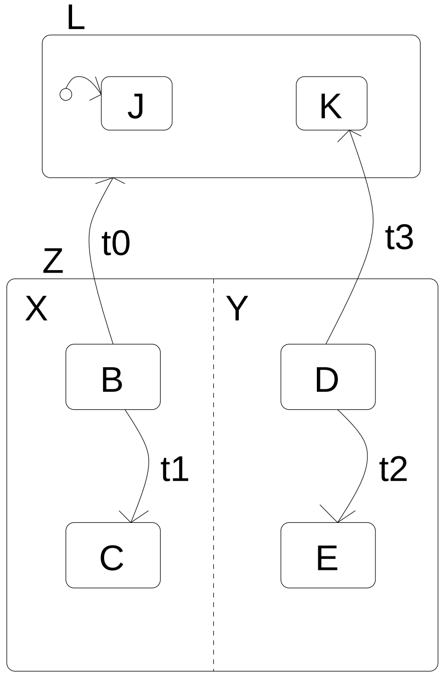

Figure 9.9: Conflicting destination states

t3 do conflict because following **t0** would lead to state **J** and **t3** leads to **K**. **J** and **K** are not in orthogonal components.

This solution is under development, but we also have to consider that if a transition is followed that exits an AND-state, the orthogonal components of the state must all be exited. Also, if a transition is followed that enters one component of an AND-state, the default entrances for the other components must be entered. This is called the *default completion* by Pnueli and Shalev [26].

We hope to take all these factors into consideration when presenting a revised transition condition to handle these situations. The problems described in Section 2.2.5 also fall under this category. Giving a solution to this problem will also give an interpretation for transitions that cross AND-state boundaries and the assumption that a statechart does not do this can be eliminated.

9.4 Linking the Model Checker with HOL

Section 6.7 discussed the use of invariants where the model checker proves the induction step and induction is used to show that the property holds for all times. Since our model checker is implemented in HOL-Voss, the framework is available to formally prove this induction in a theorem-prover.

The HOL-Voss tool has been used in this manner for hardware verification where checking the circuit description using Voss is considered a tactic[1] within an HOL proof [28]. The tactic uses the "make theorem" primitive to return the results to HOL. This method must trust Voss to return a legitimate theorem since it has not been proven by secure HOL proof steps.

In the same way, the model checker could be a tactic available for use within the theorem prover as a tactic. The primitives for induction could use its result to formally prove the invariant for all times in the model's execution.

[1] A tactic is proof step built up from a small secure set of inference rules.

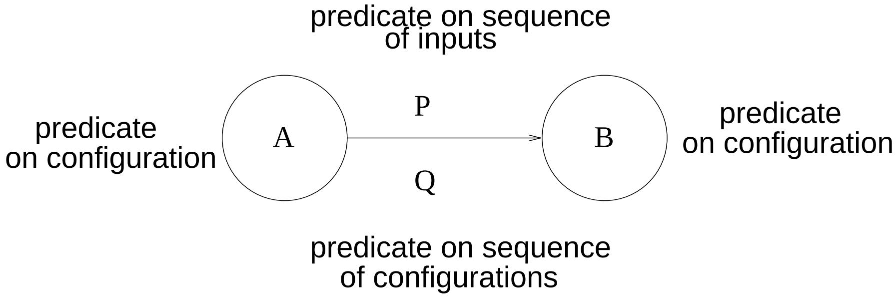

Figure 9.10: State Transition Assertion

9.5 State Transition Assertions

The descriptive specification language used for our model checker only permits statements evaluated relative to the current configuration. Another possible language for giving descriptive specifications is State Transition Assertions (STAs), developed by Gordon [6], where higher-order logic expressions describe both a starting state and invariants that must hold through a sequence of states if the concluding formula is expected to be satisfied. They can be given in the semi-graphical notation described in Figure 9.10 (similar to Figure 4 in [6]). [2]

The notation means that if the system is ever in a configuration satisfying the predicate **A** and the next sequence of inputs satisfies **P**, then the system will arrive in a configuration satisfying **B**, having gone through a series of configuration satisfying **Q**. The STA must hold true everywhere in the system.

Our model checking algorithm can be adapted to check properties given as STAs providing a more expressive descriptive specification formalism.

[2] Gordon has revised the notation slightly since this reference.

Chapter 10

Conclusions

The main conclusion of this work is that integrating formal techniques into the system development process is an effective method of providing more thorough analysis of specifications than can be achieved by conventional methods employed by commercial CASE tools. In completing this research, the semantics of statecharts have been clarified and a model checker for statecharts was created in the hybrid verification tool HOL-Voss.

10.1 Contributions and Conclusions

This work is original in the way existing tools are combined to create ways of doing useful analysis of specification models for real-time systems. An automatic model checker is linked to the CASE tool STATEMATE to perform checking of symbolic properties of the model. This link is based on a formal semantics for statecharts.

While we can argue that the STATEMATE methodology is an intuitive representation of a reactive system, there is still no guarantee that the model that is created, accurately represents what the system should do. We gain confidence that the operational specification does indeed describe the intended behaviour of the system by showing that it satisfies a set of global properties.

The main conclusions of this work can be summarized in the following points:

System Development. This work has demonstrated that formal techniques can be used to provide more thorough analysis of specifications than conventional methods employed by most commercial CASE tools. Using a model checker, we have provided a way to check a descriptive specification against an operational model given by a statechart developed in the CASE tool STATEMATE. This work has demonstrated that an operational semantics expressed as a next configuration relation is an appropriate choice for semantics when using automatic verification techniques.

A Semantics for Statecharts. The operational semantics given here for statecharts are simpler and more rigorously defined than previous published versions referred to in this work. A relation models their non-deterministic behaviour. By using total functions, the semantic definitions cover all cases. We believe the definition of a step is easier to understand while still taking into account non-determinism and priority of transitions based on source state. Race conditions and multiple actions on a single transitions are interpreted in an intuitive manner. Assignments and events are treated consistently since they are both actions. The meaning of variables is precisely given by the limits of the bit vectors used to represent them.

The semantic definitions treat triggering events as Boolean expressions over variables eliminating the need to have events as a separate type of element. Because of this, broadcast communication is expressed completely as shared variables. To characterize the priority of transitions, our semantics use existential quantification over the possible set of transitions. This technique shows promise for expressing priority where the destination states of transitions conflict as outlined in Chapter 9.

Throughout this thesis, examples of interesting statecharts can be found that could be used as a benchmark for statechart simulators. These semantics could form the basis for other tools or one could examine the semantics themselves and prove theorems about the behaviour of any statechart model.

While there are features of statecharts, such as history states and transition connectors, that are not included in our working subset of the notation, Chapter 9 gave an outline for how some of these can be incorporated into the semantics.

The Model Checker. Our model checker automatically checks simple properties of any model whose semantics can be given as a next configuration relation. It provides the foundation for implementing algorithms that can check more expressive temporal properties or for using induction to achieve results previously unattainable by automatic techniques alone.

An important result is the ease of implementing a model checker in a hybrid verification tool. This work has demonstrated the useful combination of:

- BDD-based support for the efficient manipulation of large Boolean expressions,
- a functional programming language with higher-order functions, and
- a general-purpose reasoning environment such as the HOL system.

The framework now exists for using the theorem-prover to combine the results generated automatically by the model checker to prove global constraints for all times and all execution paths of the model.

10.2 Summary

This work shows the possibility of making formal verification tools easily accessible to non-experts by linking a fully automatic verification tool to an existing CASE tool that uses a graphical formalism. Once the work of formalizing the semantics has been completed, the output from the CASE tool can be given to the model checker and the result returned automatically. In this way, we have also provided a graphical user interface for developing the models used in formal methods.

This work has served three purposes. The first is to demonstrate the usefulness of integrating commercial CASE tools with formal methods. The second and the perhaps most useful for other researchers is a clarification of the operational semantics for statecharts given as a next configuration relation. The third is to create a model checker for a hierarchical graphical specification language where realistic assertions can be verified automatically.

Appendix A

The Target Language

The target language is an executable subset of higher-order logic which consists of the following elements:

1. Variables (of any defined type)
2. Boolean constants: F, T
3. Boolean logical operators: $\wedge$ (and), $\vee$ (or), $\neg$ (not)
4. Existential quantification over Booleans ($\exists x. \ldots$)
5. Conditional expressions: A $\rightarrow$ B | C where A does not contain any symbolic values
6. Natural numbers: $\mathsf{0}$, 1, 2, ...
7. Operations on natural numbers: SUC
8. Pairs (x,y). An expression of the form:
$$(x0, x1, x2, \ldots, xn)$$
is equivalent to, (using brackets to show the pairs):
$$(x0, (x1, (x2, \ldots, xn) \ldots)$$
9. Operations on pairs: FST, SND
10. Operations on lists: HD, TL, CONS, $[\,]$, MEMBER, MAP , FLAT, EL, APPEND
11. Function application $(f\ \ x)$
12. λ-expressions $(\lambda x. f\ (x))$
13. **let** expressions

Some definitions for the list processing functions are included below.

MEMBER is a predicate which returns true if an element e is a member of a list x:

$$\mathsf{MEMBER}\ e\ x\ =_{def}\ (x = [\,]) \rightarrow \mathsf{F} \mid ((e = (\mathsf{HD}\ x)) \rightarrow \mathsf{T} \mid \mathsf{MEMBER}\ e\ (\mathsf{TL}\ x))$$

MAP applies a function to each element in a list:

$$\mathsf{MAP}\ f\ x\ =_{def}\ (x = [\,]) \rightarrow [\,] \mid \mathsf{CONS}\,(f\ (\mathsf{HD}\ x))\,(\mathsf{MAP}\ f\ (\mathsf{TL}\ x))$$

FLAT reduces a list of lists to one list:

$$\mathsf{FLAT}\ x\ =_{def}\ (x = [\,]) \rightarrow [\,] \mid \mathsf{APPEND}\,(\mathsf{HD}x)\,(\mathsf{FLAT}(\mathsf{TL}\ x))$$

EL returns the nth element of the list x:

$$\mathsf{EL}\ n\ x\ =_{def}\ (n = 0) \rightarrow (\mathsf{HD}\ x) \mid \mathsf{EL}\ (n-1)\ x$$

Appendix B

Bit Vector Operations

These functions are part of a bit vector package prepared for use with HOL-Voss [28].

Definitions

`BV2NUM` $\vdash (\mathsf{BV2NUM}\,[\,] = 0) \wedge (\forall h\,t.\ \mathsf{BV2NUM}\,(\mathsf{CONS}\,h\,t) = (h \Rightarrow 1 \mid 0) + 2 \times \mathsf{BV2NUM}\,t)$

`NUM2BV_AUX` $\vdash (\forall m.\ \mathsf{NUM2BV\_AUX}\,0\,m = [\,]) \wedge (\forall n\,m.\ \mathsf{NUM2BV\_AUX}\,(\mathsf{SUC}\,n)\,m = ((m = 0) \Rightarrow [\,] \mid \mathsf{CONS}\,(m\ \mathsf{MOD}\ 2 = 1)\,(\mathsf{NUM2BV\_AUX}\,n\,(m\ \mathsf{DIV}\ 2))))$

`NUM2BV` $\vdash \forall n.\ \mathsf{NUM2BV}\,n = \mathsf{NUM2BV\_AUX}\,n\,n$

`ZEROS` $\vdash (\mathsf{ZEROS}\,0 = [\,]) \wedge (\forall n.\ \mathsf{ZEROS}\,(\mathsf{SUC}\,n) = \mathsf{CONS}\,\mathsf{F}\,(\mathsf{ZEROS}\,n))$

`SIZED` $\vdash (\forall b.\ \mathsf{SIZED}\,0\,b = [\,]) \wedge (\forall n\,b.\ \mathsf{SIZED}\,(\mathsf{SUC}\,n)\,b = (\mathsf{NULL}\,b \Rightarrow \mathsf{ZEROS}\,(\mathsf{SUC}\,n) \mid \mathsf{CONS}\,(\mathsf{HD}\,b)\,(\mathsf{SIZED}\,n\,(\mathsf{TL}\,b))))$

`BVPLUS2_AUX` $\vdash (\forall c.\ \mathsf{BVPLUS2\_AUX}\,[\,]\,c = [c]) \wedge (\forall h\,r\,c.\ \mathsf{BVPLUS2\_AUX}\,(\mathsf{CONS}\,h\,r)\,c = \mathsf{CONS}\,(c \wedge \neg h \vee \neg c \wedge h)\,(\mathsf{BVPLUS2\_AUX}\,r\,(c \wedge h)))$

`BVPLUS_AUX` $\vdash (\forall b\,c.\ \mathsf{BVPLUS\_AUX}\,[\,]\,b\,c = \mathsf{BVPLUS2\_AUX}\,b\,c) \wedge (\forall h\,r\,b\,c.\ \mathsf{BVPLUS\_AUX}\,(\mathsf{CONS}\,h\,r)\,b\,c = ((b = [\,]) \Rightarrow \mathsf{BVPLUS2\_AUX}\,(\mathsf{CONS}\,h\,r)\,c \mid \mathsf{CONS}\,(h \wedge \neg \mathsf{HD}\,b \wedge \neg c \vee \neg h \wedge \mathsf{HD}\,b \wedge \neg c \vee \neg h \wedge \neg \mathsf{HD}\,b \wedge c \vee h \wedge \mathsf{HD}\,b \wedge c)\,(\mathsf{BVPLUS\_AUX}\,r\,(\mathsf{TL}\,b)\,(h \wedge \mathsf{HD}\,b \vee c \wedge \mathsf{HD}\,b \vee h \wedge c))))$

`BVPLUS` $\vdash \forall a\,b.\ a\ \mathsf{BVPLUS}\ b = \mathsf{BVPLUS\_AUX}\,a\,b\,\mathsf{F}$

`BVMULT` $\vdash (\forall av.\ av\ \mathsf{BVMULT}\ [\,] = [\mathsf{F}]) \wedge (\forall av\,h\,r.\ av\ \mathsf{BVMULT}\ \mathsf{CONS}\,h\,r = \mathsf{MAP}\,(\lambda v.\ h \wedge v)\,av\ \mathsf{BVPLUS}\ \mathsf{CONS}\,\mathsf{F}\,(av\ \mathsf{BVMULT}\ r))$

`BVEQUAL_ZERO` $\vdash (\mathsf{BVEQUAL\_ZERO}\,[\,] = \mathsf{T}) \wedge (\forall h\,t.\ \mathsf{BVEQUAL\_ZERO}\,(\mathsf{CONS}\,h\,t) = \neg h \wedge \mathsf{BVEQUAL\_ZERO}\,t)$

`BVEQUAL` $\vdash (\forall b.\ [\,]\ \mathsf{BVEQUAL}\ b = \mathsf{BVEQUAL\_ZERO}\,b) \wedge (\forall h\,t\,b.\ \mathsf{CONS}\,h\,t\ \mathsf{BVEQUAL}\ b = (\mathsf{NULL}\,b \Rightarrow \mathsf{BVEQUAL\_ZERO}\,(\mathsf{CONS}\,h\,t) \mid ((h = \mathsf{HD}\,b) \wedge t\ \mathsf{BVEQUAL}\ \mathsf{TL}\,b)))$

`BVGREATER_AUX` $\vdash (\forall b\,res.\ \mathsf{BVGREATER\_AUX}\,[\,]\,b\,res = res \wedge \mathsf{BVEQUAL\_ZERO}\,b) \wedge (\forall h\,t\,b\,res.\ \mathsf{BVGREATER\_AUX}\,(\mathsf{CONS}\,h\,t)\,b\,res = (\mathsf{NULL}\,b \Rightarrow (res \vee h \vee \neg\mathsf{BVEQUAL\_ZERO}\,t) \mid \mathsf{BVGREATER\_AUX}\,t\,(\mathsf{TL}\,b)\,(h \wedge \neg \mathsf{HD}\,b \vee res \wedge (h = \mathsf{HD}\,b))))$

`BVGREATER` $\vdash \forall a\,b.\ a\ \mathsf{BVGREATER}\ b = \mathsf{BVGREATER\_AUX}\,a\,b\,\mathsf{F}$

`BOOL` $\vdash \forall a.\ \mathsf{BOOL}\,a = \mathsf{HD}\,a$

`BVAL` $\vdash \forall a.\ \mathsf{BVAL}\,a = [a]$

`NVAL`	$\vdash \mathsf{NVAL} = \mathsf{NUM2BV}$
`NPLUS`	$\vdash \forall a\, b.\, \mathsf{NPLUS}\,(a, b) = a\, \mathsf{BVPLUS}\, b$
`NMULT`	$\vdash \forall a\, b.\, \mathsf{NMULT}\,(a, b) = a\, \mathsf{BVMULT}\, b$
`NGREATER`	$\vdash \forall a\, b.\, \mathsf{NGREATER}\,(a, b) = \mathsf{BVAL}\,(a\, \mathsf{BVGREATER}\, b)$
`BAND`	$\vdash \forall a\, b.\, \mathsf{BAND}\,(a, b) = \mathsf{BVAL}\,(\mathsf{BOOL}\, a \wedge \mathsf{BOOL}\, b)$
`BOR`	$\vdash \forall a\, b.\, \mathsf{BOR}\,(a, b) = \mathsf{BVAL}\,(\mathsf{BOOL}\, a \vee \mathsf{BOOL}\, b)$
`BNOT`	$\vdash \forall a.\, \mathsf{BNOT}\, a = \mathsf{BVAL}\,(\neg \mathsf{BOOL}\, a)$
`BFALSE`	$\vdash \mathsf{BFALSE} = \mathsf{BVAL}\, \mathsf{F}$
`BTRUE`	$\vdash \mathsf{BTRUE} = \mathsf{BVAL}\, \mathsf{T}$
`EQVAL`	$\vdash \forall a\, b.\, \mathsf{EQVAL}\,(a, b) = \mathsf{BVAL}\,(a\, \mathsf{BVEQUAL}\, \mathsf{SIZED}\, (\mathsf{LENGTH}\, a)\, b)$
`MAXVALUE`	$\vdash (\mathsf{MAXVALUE}\,[\,] = \mathsf{T}) \wedge (\forall h\, t.\, \mathsf{MAXVALUE}\,(\mathsf{CONS}\, h\, t) = h \wedge \mathsf{MAXVALUE}\, t)$

Bibliography

[1] Rajeev Alur, Costa Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata:an algorithmic approach to the specification and verification of hybrid systems.

[2] Andrei Borshchev. Private communication, June 1992.

[3] Randel E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] 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.

[5] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[6] Mike Gordon. A formal method for hard real-time programming. Computer Laboratory, Cambridge, UK.

[7] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[8] 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.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

[10] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[11] David Harel. Biting the silver bullet. *IEEE Computer*, 25(1):8–20, January 1992.

[12] David Harel and Chaim-Arie Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, October 1992.

[13] i-Logix Inc., Burlington, MA. *The Semantics of Statecharts*, January 1991.

[14] i-Logix Inc., Burlington, MA. *Statemate 4.0 Analyzer User and Reference Manual*, April 1991.

[15] i-Logix Inc., Burlington, MA. *Statemate 4.0 User and Reference Manual*, April 1991.

[16] i-Logix Inc., Burlington, MA. *Statemate Dataport 4.0*, August 1991.

[17] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler.* Bell Laboratories, Murray Hill, New Jersey.

[18] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. Technical Report No. 178, University of Cambridge Computer Laboratory, September 1989.

[19] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. Weizmann Institute of Science.

[20] M.E. Lesk and E. Scmidt. *Lex - A Lexical Analyzer Generator.* Bell Laboratories, Murray Hill, New Jersey.

[21] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. Technical Report 92-106, University of California, Irvine, Information and Computer Science, 1992.

[22] Andrew K. Martin. Discrete conservative models of continuous systems. Phd work underway at the University of British Columbia.

[23] Andrew K. Martin. Private communication, March 1993.

[24] Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992.

[25] ObjecTime Limited. *Introduction to ObjecTime*, 1992.

[26] 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.

[27] C. Seger. Voss — a practical formal verification system based on symbolic trajectory evaluation. In preparation.

[28] 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.

[29] J. Staunstrup, editor. *Formal Methods for VLSI Design*, chapter 2. North-Holland, 1990.

[30] Zhang Ying. A formal model and logic for robotic systems and behaviours: A proposal for doctoral dissertation. Department of Computer Science, University of British Columbia.