An Example of Linking Formal Methods with CASE Tools (A Model Checker for Statecharts)

Nancy Day*

September 2, 1993

Abstract

Computer-Aided Software Engineering (CASE) tools encourage users to codify the requirements 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 but others can test the model for reachability of conditions, nondeterminism, or deadlock. In this paper, we present an example of how commercial CASE tools can be linked with formal methods to build more thorough forms of analysis into these tools.

The CASE tool STATEMATE [12] makes use of an extended state transition notation called statecharts. We have formalized the semantics of 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 and translate it into a textual representation that can be directly input into the HOL system.

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 Voss.

This paper illustrates this method through two examples to show how our model checker may be used to aid in the analysis of the requirements of a system.

*Supported by a Natural Science and Engineering Research Council Post Graduate Scholarship

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 [13]. 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 how a model checker can be integrated with the CASE tool STATEMATE. 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.

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 [3].

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.

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 [11].

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 nondeterministic, 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.

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.

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 [9] 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.

In this paper we discuss the model checking algorithm and present some examples of its use. 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.

There are several situations where it is not obvious how a statechart should be interpreted. 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's Requirements State Machine Language (RSML) [13] 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.

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 [6][10][13][15]. 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.

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 rep-

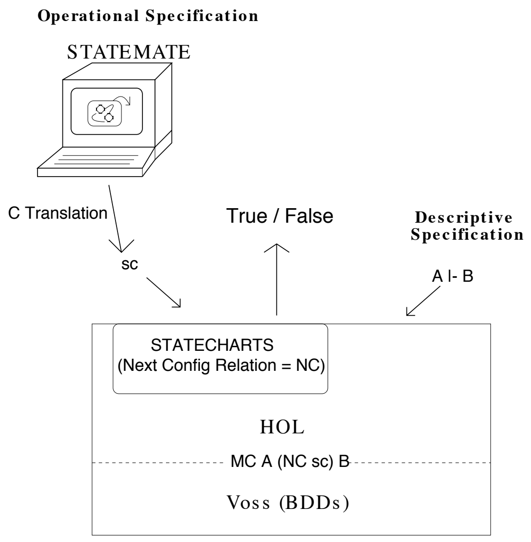

Operational Specification
STATEMATE
C Translation
sc
True / False
Descriptive Specification
A |- B
STATECHARTS (Next Config Relation = NC)
HOL
MC A (NC sc) B
Voss (BDDs)

Figure 1: The overall method

resentation of configurations. HOL-Voss [17] is a hybrid verification tool that combines an interactive proof-assistant, HOL [5], based on higher-order logic, with an efficient, automatic symbolic simulator, Voss [16], that uses ordered binary decision diagrams (BDDs) [1]. 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.

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 [17]. Our model checker is created in HOL-Voss so the theorem-prover is available for this type of use. The traffic light example presented later gives an example of how results from the model checker can be combined using induction to prove a stronger property.

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.

8 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 [6]

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 [8] 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 [11]. The statechart for the traffic light controller is given in Figure 2 (from Figure 3-22 in [11]) 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

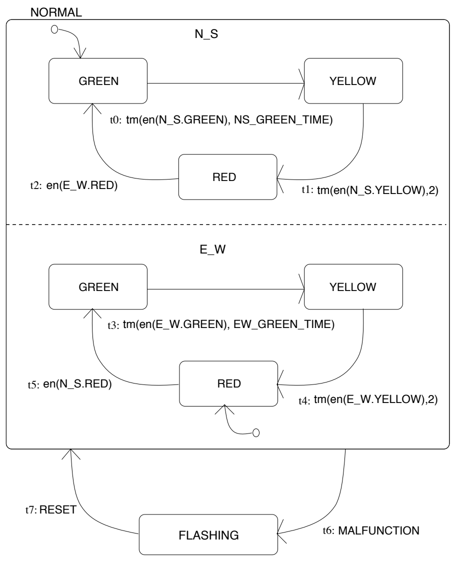


Figure 2: Traffic light statechart

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 *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 .

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

$$\xrightarrow{\quad event\ [condition]\ /\ action \quad}$$

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.

An *event* is generated when there is a change in a condition. This is a discrete version of "the instantaneous occurrence of a stimulus" [12]. 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.

9 The Semantics of Statecharts

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

The first effort towards a formal semantics for statecharts was by Harel et al. [6]. Pnueli and Shalev [15] 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. 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. [10][11]. We also considered the semantics presented by Leveson et al. [13] 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 [7][8][9][10], has been used to help determine the less obvious features of statecharts and formalize our interpretation of the semantics of statecharts.

[1]The STATEMATE manual calls this concept a *status* [11].

For our purposes, meaning is given to a statechart through a set of semantic functions. These functions translate the syntax sc into a Boolean relation over the variables of the current configuration and of the next configuration. This relation is given by $\mathsf{NC}\ sc$. The configuration of the system is completely represented by a set of Boolean variables which includes elements for the basic states (where true means the system is currently in that basic state), manipulated variables used in expressions (given by bit vectors), and events (often given by counters). The values of the basic states determine the meaning of all higher-level rounded boxes in the hierarchy.

Actions can assign Boolean or arithmetic expressions to variables. The meaning of these expressions is evaluated compositionally by examining the parts of the expression relative to the current configuration and then assigning the value to the variable in the next configuration. For example, the meaning of equality operator is given by:

$$\mathsf{SemEQUAL}\,(a_1, a_2) = \lambda cf : Config.$$
$$\mathsf{EQVAL}(a_1\, cf\ , a_2\, cf)$$

where a_1 and a_2 are expressions that take a configuration as an argument and return a value, as SemEQUAL does, and EQVAL is the equality test for bit vectors.

The value of conditions is determined in a similar manner, making reference only to the current configuration. To evaluate events, it is necessary to look at past configurations as well, since an event describes a change in a condition between the current and previous time step. Our interpretation matches the one used in STATEMATE, where events generated in this time step are not considered until the next step.

The difficulty in giving the semantics for this language 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. Unlike previous semantics, we base the priority of transitions on their source state. Combining the actions of the transitions taken depends on whether there are race conditions where more than one transition which is followed modifies the same variable. It is also necessary to express the condition that if a variable is not modified in a given step, then it retains its previous value.

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.

Because we have used total functions to give the semantic definitions, every statechart has an interpretation. The semantics consists of three parts:

- conditions on the set of transitions that can be taken, including hierarchy, priority and triggers,
- conditions on the variables in the next configuration, and
- conditions on the next complete system configuration which must remain legal.

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

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 section. Through this process, errors were discovered and fixed, and we have increased confidence in the result.

10 The Model Checker

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[2] when all configurations were explicitly represented. The symbolic model checking algorithm used here is a limited form of the one presented by McMillan [14] 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 following elements are required to carry out the model checking:

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

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

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

11 Descriptive Specifications

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

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

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

where $\mathbf{i}$ and $\mathbf{f}$ 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 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 14 shows how MC_A can be used as part of an inductive proof to show $\mathbf{f}$ is true for all times, not just within a time constraint.

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 [14]. 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. We have also looked at the possibility of using another notation called State Transition Assertions [4] to give more expressive descriptive specifications with only small changes in the model checking algorithm used here.

12 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 which determines if property holds along every execution path (MC_A) and then describe the simple changes to calculate MC_E.

Over a limited number of steps, the model checker determines the set of next configurations that do not satisfy the property $\mathbf{f}$ 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'.

$$\lambda cf'.\, \exists cf.\, \mathbf{i}\, cf \wedge \mathbf{NextConfig}\, cf\, cf' \qquad (1)$$

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

$$\begin{array}{l}\mathbf{let}\ \ check = \ \ \exists cf'.\, \exists cf. \\ \quad \mathbf{i}\, cf \ \wedge\ \mathbf{NextConfig}\, cf\, cf' \ \wedge\ \neg\mathbf{f}\, cf' \end{array} \qquad (2)$$

If this expression is false ($\neg check$) then the property is true in all current configurations 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 2) without the quantification over the next state:

$$\begin{aligned}&\textbf{let } nextX = \lambda cf'.\exists cf.\\ &\quad \mathbf{i}\, cf \;\wedge\; \textbf{NextConfig}\, cf\, cf' \;\wedge\; \neg\mathbf{f}\, cf' \qquad (3)\end{aligned}$$

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

$$\begin{aligned}&\mathsf{MC\_A}\ step\ \mathbf{i}\ \textbf{NextConfig}\ \mathbf{f} =_{def}\\ &\textbf{let } check = \exists cf'.\exists cf.\\ &\quad \mathbf{i}\, cf \;\wedge\; \textbf{NextConfig}\, cf\, cf' \;\wedge\; \neg\mathbf{f}\, cf' \ \mathsf{in}\ (2)\\ &\textbf{let } nextX = \lambda cf'.\exists cf.\\ &\quad \mathbf{i}\, cf \;\wedge\; \textbf{NextConfig}\, cf\, cf' \;\wedge\; \neg\mathbf{f}\, cf' \ \mathsf{in}\ (3)\\ &(step = 0) \Rightarrow \mathsf{False}\ \mid\\ &\neg(check) \vee\\ &\quad \mathsf{MC\_A}\ (step - 1)\ nextX\ \textbf{NextConfig}\ \mathbf{f}\end{aligned}$$

The algorithm to determine if the property **f** is ever true within a certain number of steps ($\mathsf{MC\_E}$) is similar except that it stops as soon as it finds a configuration where **f** is true and otherwise continues with all possible configurations.

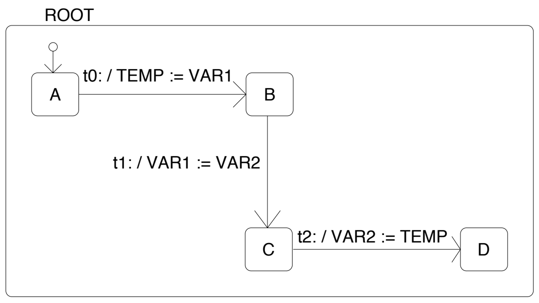


Figure 3: Swap operation

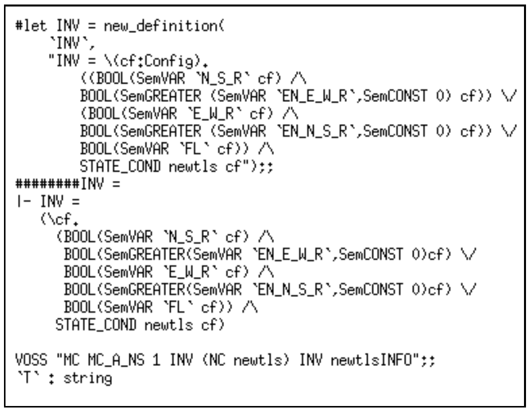
```
#let START = new_definition(`START`,
        "START = \cf.
                INITIAL swap cf /\
                BOOL(SemEQUAL(SemVAR `VAR1`,SemVAR `X`) cf) /\
                BOOL(SemEQUAL(SemVAR `VAR2`,SemVAR `Y`) cf) ");;

####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 4: Swap test

13 A Simple Example

The 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 3. 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. In our first test, we allocate one bit to all data items and symbolic values. The constant $swapINFO1$ holds the information about the representation of the configuration. The results of the test in Figure 4 show that after three steps the model successfully completes the operation.



```
#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 5: Checking the invariant for the traffic light

14 Traffic Light Example

For the statechart of Figure 2, we would like to show it has the safety property that a red light is on in one direction at all times. Even though the model checking algorithm only checks for a bounded number of steps, we can use it to prove the induction step in an inductive proof to show that this invariant is true for all times. Starting from the set of configurations which satisfy this safety property, we can show that within one step, all the configurations that can be reached also satisfy the safety property. Then we show that the property also holds in the initial configuration of the statechart and using induction, this means the property holds for all times. The difficulty with this approach is finding an invariant which limits the set of configurations sufficiently to exclude configurations which may be safe, but lead to unsafe configurations in the next step. These should not be reachable configurations. In this case, we had to strengthen the invariant to include conditions on the event counters which are used to determine when events like **en** and **tm** occur. Figure 5 shows the results of running the model checker for the invariant.[3] Again, by relying on induction, we have proven that the safety property holds true for all times during the model's execution.

[3]Abbreviated names were used for the states, i.e., **N_S.RED** is 'N_S_R'

15 Conclusions

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. 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.

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. A useful addition would be the ability to document a situation where the descriptive specification fails if the model checker is not able to prove the property.

The two examples presented here demonstrated the use of symbolic values to test a range of values in one run of the model checker and proving an invariant using the model checker to test the induction step. The semantic and model checking functions as well as more examples can be found in the more thorough explanation of this work [2].

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.

16 Acknowledgments

I would like to thank my supervisor Jeff Joyce and the members of the Integrated Systems Design Laboratory for many useful discussions on statecharts and several of the graduate students at UBC for helpful comments on drafts of this paper.

About the author

Nancy Day has just finished her master's degree with the Integrated Systems Design Group of the Department of Computer Science at the University of British Columbia. She can be reached at day@cs.ubc.ca.

References

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

[2] Nancy Day. A model checker for statecharts. Master's thesis, University of British Columbia, 1993. In preparation.

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

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

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

[6] 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.

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

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

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

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

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

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

[13] 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.

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

[15] 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.

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

[17] 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.