A Comparison between Statecharts and State Transition Assertions

Nancy Day

Integrated Systems Design Laboratory, Department of Computer Science, University of British Columbia, Vancouver, BC, V6T 1Z2, Canada.

Abstract

This paper compares statecharts, a specification formalism for reactive systems, to state transition assertions, a verification method for hard real-time systems. While these two methods are used for different tasks and they take different points of view in describing a system, it is useful to compare them to determine what is necessary in a formal specification notation for real-time systems. In this paper, we conclude with a list of issues that need to be resolved when integrating formal verification with a specification notation. The future goal of this work is to provide a more readable front-end specification formalism which can be used for verification. The purpose of doing a formal verification of specifications is to check for correctness early in the system development process and discover errors which can prove costly in later stages. If a more readable notation like statecharts is embedded in the theorem-prover, HOL (Higher Order Logic), it would provide the tools necessary to do mechanized verification.

Keyword Codes: D.2.1;D.2.4;F.3.1
Keywords: Software Engineering, Requirements/Specifications; Program Verification; Logics, Specifying and Verifying and Reasoning about Programs

1 Introduction

Discovering errors early in the design and implementation of any system, reduces the cost of development and increases one's confidence in the reliability of the end result. Top-down system development begins with a very high-level, abstract description of the system, taking into account its environment and the results it is expected to produce. More and more details are added as the process moves closer to a final product. Each level can be viewed operationally as a way of moving from one moment in execution to the next or it can be viewed as a specification for what the next, more detailed level must produce at the end of a step or series of steps. Phases in this development process are commonly refered to as specification, design, and implementation.

Although formal specification and verification methods show promise for reducing or eliminating errors[13], the difficulty in learning these techniques makes industry reluctant to use them except possibly as an afterthought[13]. Perhaps before using the system, it is "verified" using some method, but this is separate from the steps of the development process. However, work in industry has already demonstrated the advantages of using formal specification notations early in the development process[13]. If we take the second point of view mentioned above, that one level of description is a specification for the next level, then there is room to use formal verification techniques to ensure that a lower level description correctly implements a higher level one. It would also be possible to show overall system properties, such as safety requirements, continue to hold as more implementation details are added.

The general purpose theorem prover HOL (Higher Order Logic)[5] makes it possible to machine-check proofs of this form. Previous work in verifying that level by level descriptions correctly interpret a more abstract level has been done for hardware[14][17], and is currently being completed for the complete "stack" of a system including the hardware, compiler, and software (SAFEMOS at SRI International). One of the advantages of using a theorem-prover is that a compositional approach is possible which is well-suited to viewing parts of the system at different levels of abstraction and reusing completed proofs.

This paper looks at two different approaches to system specification: statecharts (D. Harel [7]), and state transition assertions(STAs) (M. Gordon [4]). These two formalisms were chosen because they approach the problem from the two different points of view that we would like to see integrated. Statecharts have an elegant visual notation for describing the operation of a system and are supported by the commercial tool STATEMATE [10]. STAs are designed for use in formally verifying systems and expressing limitations on the system's implementation.

At the end of this comparison, we are able to conclude that five issues which need to be considered when looking for a real-time specification formalism to be used for verification are:

- Does it express operation or assertions?
- Can it handle complete system description at any level?
- Does it have a formal model of time?
- Can it integrate data and control descriptions?
- Is it a visual notation?

The first section of this paper defines the type of system these specification techniques are used for. Then a brief description of each method is given. An example described in each notation is presented in Section 5 to provide a basis for comparing them. Finally, we present the list of issues outlined above.

2 Terminology

Before beginning to look at methods for describing real-time reactive systems, we should define precisely the meanings of "reactive" and "real-time".

In Harel's original paper on statecharts[7], he defines reactive systems as being event driven, meaning they have to respond to both internal and external stimuli. In a later paper[10], he states that it is necessary to "specify the relationship between inputs and outputs over time". Gordon places the emphasis on the timing requirements in saying hard real-time systems "are required to meet explicit timing constraints, such as responding to an input within 100 milliseconds of a change" [4].

For our purposes, real-time systems are a type of reactive system where the response time to an event is an essential part of the description of a system. It is important to note that the word "system" is meant to describe the complete set-up including software, computing hardware, and any specialized peripherals.

3 Statecharts

Statecharts build on the ideas of finite state machines (FSMs), extending the notation to overcome some of its limitations. The system is described as being in one of a number of possible states, and it changes from state to state as a result of events which occur.

Since statecharts are a type of higraph, they are a *visual formalism.* Higraphs are a notation which combines the ideas of graphs and Venn diagrams [9]. The nodes (or blobs) in the graph represent states the system can be in. The edges indicate transitions between states. This movement is considered to happen instantaneously. Please note that in the following description, transition labels such as **T1** have no semantic meaning and are only used to reference certain transitions.

With reference to the statechart of Figure 1, the main features of statecharts are the following:

- *Concurrency:* The Cartesian product of states is represented by orthogonal components of a state. In the example, states **A** and **D** are orthogonal components which make up state **Y**. They are separated by a dashed line through state **Y**. (Also note that the labelling of the state **Y** is in a small box outside of its contour.) This means that for the system to be in state **Y**, it must be in state **A** *and* in state **D**. This notation prevents the explosion of states in a FSM resulting from the Cartesian product of state diagrams.

  Synchronization between the various substates the system is in simultaneously can be accomplished using events, conditions, and actions which are described below.

- *Hierarchy:* A state can be decomposed into substates (often called OR states) indicating more of the internal workings of a state. A state which is not refined is called a *basic state.* In the example, state **A** is decomposed (or refined) into the basic states **B** and **C**. This means that when the system is in state **A**, it is in either state **B** *or* state **C** (exclusive or). (**B** is called a descendant of **A** and **A** is an ancestor of **B**.) This hierarchy of states allows a system to be initially specified at a higher level of abstraction, and later decomposed, as the design process progresses.

- *Transitions:* Transitions are labelled with the notation $e[c]/a$. The occurrence of event e enables the transition (meaning it can be taken) providing the condition c is

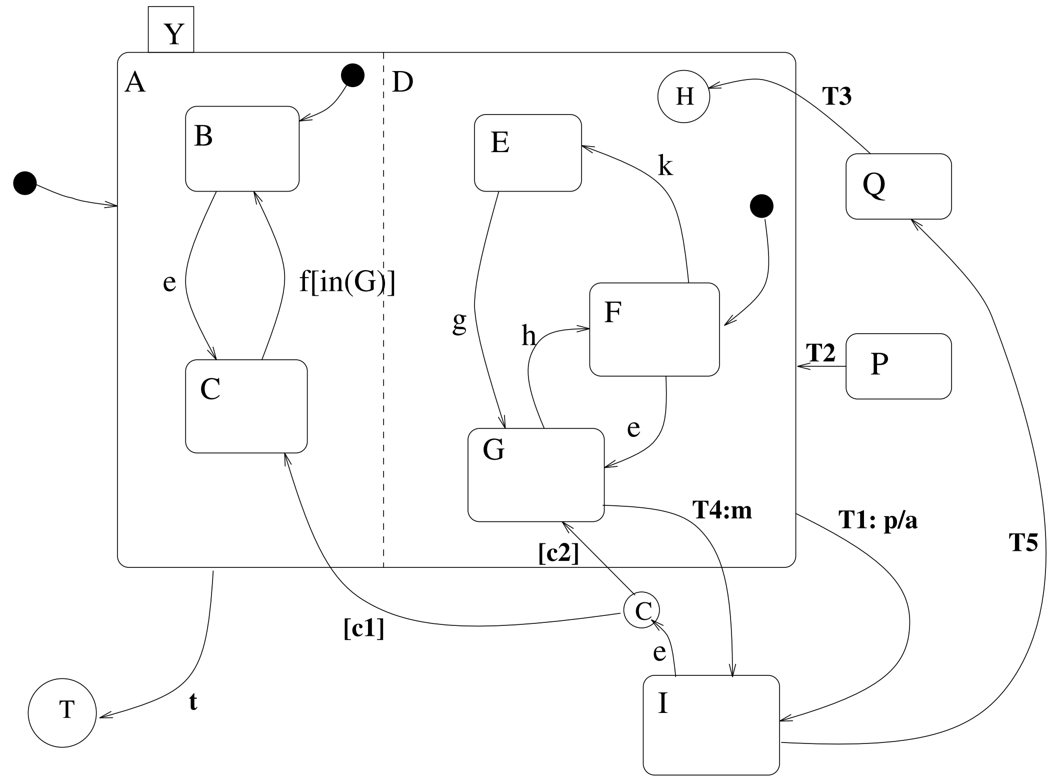

Figure 1: An Example of a Statechart

true. (If no condition is given then "true" is assumed.) A transition can be enabled if it originates in any descendant or ancestor of a state the system is currently in. If the transition is taken then the actions a are carried out. These actions can be to start and stop data processes (called *activities*), to generate events, such as internal events for synchronization, or to change the values of variables or conditions. For example, if the system is currently in the basic states (**B**,**F**) and the event **p** occurs, then transition **T1** is enabled. All parts of the transition label are optional. If neither an event nor a condition is given then the transition is always enabled.

- *Broadcast communication*: This type of communication is implicit in the operation of statecharts. This means that every external and internal event which occurs can be seen in all parts of the system.

- *Default states:* The small arrow originating from a filled-in circle pointing at state **B** in Figure 1 gives a default state for state **A**. It means that whenever a transition terminates at the outside boundary of a non-basic state the default arrow will be followed to enter a basic state. The transition labelled **T2** originating at state **P** is an example of this. Following this transition will cause the system to follow default entries for state **Y**, which in turn means default entries for both states **A** and **D**, leaving the system in state **B** and **F**.

- *History states:* A history connector, marked with an **H** can also be the destination of a transition. It dynamically represents the substate which this state was in at the time it was last exited. For example, if the transition labelled **T4** from state **G** triggered by event **m** was taken to leave state **D** and the system then followed **T5** to arrive in state **Q**, and **T3** terminating at the history connector, to return to **D**, it would re-enter state **G**. If there is no history (ie 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.

- *Terminal connectors:* Entering a terminal connector, labelled by **T** in a circle, stops all processing in the system.
- *Transition connectors:* These connectors are a way of decomposing transitions into smaller parts. For example the three transitions joined by the **C** connector in the diagram really represent two transitions from state **I**, one going to **G** and labelled **e[c2]**, and the other going to state **C**, labelled **e[c1]**.

3.1 The Semantics of Statecharts

While statecharts have the advantage of using pictures to portray a great deal of information about the system being specified, there are instances in which it is not clear from the diagram what is intended by the specification. These problems have been noted and discussed by Harel and others[6][12]. Harel's description of the semantics [6] differs from those used in STATEMATE somewhat. In particular, some operators which he describes for checking the state of the system in between "steps" (see the third section below) are not implemented. In this section, we characterize some of these subtleties and describe the semantics which STATEMATE uses in simulating the specification.

A great deal of work has been completed recently on incorporating timing models into statecharts[11] [15]. Since STATEMATE was developed before this work, it is based on the original form of statecharts. Because our future goal is to provide a connection between a commercial CASE tool and formal verification through the theorem-prover HOL, we have decided, at this time, to consider only the semantics of statecharts in STATEMATE.

3.1.1 Non-determinism

A non-deterministic situation exists when an event enables two or more possible transitions leaving the same state. Even if the transitions have the same target, there could be different actions associated with each one. The simulator of STATEMATE prompts the user to choose which transition to follow in a case like this.

3.1.2 Structural Non-determinism

Statecharts are very good at graphically describing a hierarchy of states. This allows for refinement of specifications, or the opposite, abstraction. This is accomplished using OR states. However, transitions can leave any state boundary and it is possible the same event will trigger a transition from both a state and one of its ancestor states. An example is given in Figure 2 where the event **e** triggers the transition **T1**, leaving state **B** as well as one from its parent, state **A**, labelled **T2**.

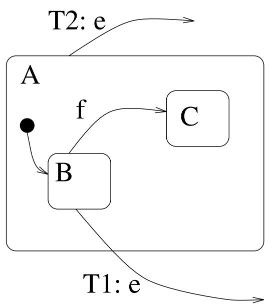

Figure 2: Structural Ambiguity

In STATEMATE, priority is given to transitions leaving states higher up in the hierarchy. In the preceding example, the transition **T2** would be taken.

3.1.3 Timing

There is no inherent model of timing associated with statecharts other than the movement between states by following transitions. STATEMATE offers both an asychronous and a synchronous model of time. The first stage in the algorithm for both models is to determine all the enabled compound transitions. A compound transition(CT) is the connection of all the transitions necessary to go from one basic state to another (through default and history connectors, etc). The trigger for a CT is the conjunction of the triggers for its component transitions. A maximal, non-conflicting set of enabled transitions for the system is then determined by randomly picking one when two or more transitions leave the same state.

In the synchronous model, time is incremented just before the set of CTs to be executed is determined and then the system performs these transitions in a random order. Performing a transition consists of doing the actions for exiting one state, entering the new state and then the actions for the transition itself. The execution of the complete set of CTs is called a "step" and the results of actions and any internal events generated are not available until the next step.

In the asynchronous model, the system repeatedly determines and executes all enabled CTs without incrementing time or considering any new external events generated until there are no more transitions enabled. This is called a "super step". This is the model Harel describes in his paper[6] but there is some question as to whether external events persist for the length of the entire super step or whether only internal events are relevant in later steps of a super step. The examples in Harel's paper indicate that the external events persist for the whole time. The author has yet to experiment with this in STATEMATE.

Another model of time which could have been used is to determine the set of enabled transitions after each execution of a compound transition. This way, internally generated events could have an effect on the next step (where step means executing only one CT). However, this would create more opportunities for race conditions.

These three possible models raise questions about how much time a transition should take. (Are they really instantaneous?) Also, when should the system react to external events?

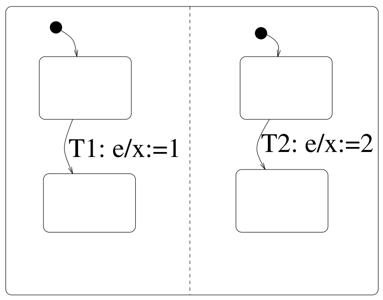

Figure 3: Race conditions

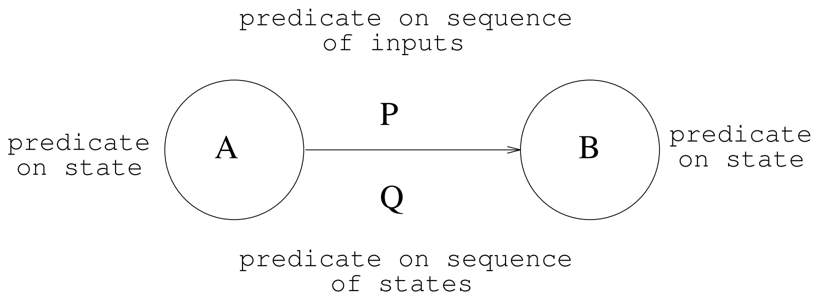
predicate on sequence of inputs
predicate on state
A
P
B
predicate on state
Q
predicate on sequence of states

Figure 4: State Transition Assertion

3.1.4 Race Conditions

Race conditions occur when two transitions are enabled at the same time but if executed in different orders sequentially will have different results. An example is given in Figure 3 when the transitions **T1** and **T2** are both enabled and the last one to finish execution will determine the value of **x**. STATEMATE chooses randomly which transition to follow first although this effect is repeatable.

4 State Transition Assertions

State transition assertions are logical statements of constraints on the sequence of state transitions of a machine. They use the semi-graphical notation described in Figure 4 (from Figure 4 in [4]).

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

STAs are formulated about the problem at the specification level. An STA can also describe the semantics of a single machine instruction of a simple microprocessor where the transition only takes one step to execute (=1).

For example, for a typical jump instruction, JMP n would be described by:

$$< pc, stk, mem > \quad \xrightarrow[=1]{[T]} \quad < n, stk, mem >$$

where *pc*, *stk*, and *mem* mean program counter, stack, and memory respectively. The predicate on the sequence of inputs is [T] (true) since this operation will hold true for any input.

Every transition (even ones composed of several steps) must take at least one time unit. Using rules in the axiomatic style, Gordon is able to compose lower level STAs to see if the higher level ones hold. The following is an example of one of these rules describing a form of transitivity:

$$\frac{A \xrightarrow[\leq \delta_1 \wedge [q_1]]{P_1} B \qquad B \xrightarrow[\leq \delta_2 \wedge [q_2]]{P_2} C}{A \xrightarrow[\leq (\delta_1 + \delta_2) \wedge ([q_1] \vee [q_2])]{P_1 \wedge P_2} C}$$

The timing restrictions on all these phases are given by the notation $\leq \delta$ or $= \delta$ in the STA to indicate the number of time steps or transitions taken between states.

One of the most important features of Gordon's work is his use of axiomatic semantics to compose STAs to produce higher level statements about the system. It also bridges the gap between the specification and the implementation in the machine code. This makes the timing more exact.

> Verification of the system implementation is accomplished by translating the low level machine code into primitive STAs and showing that these STAs correspond to the STAs of the specification [1].

This method could be used as a bottom-up approach to verification. Difficulties arise in attempting to use it as a top-down approach because of the strong constraints placed on inputs by the predicate **P**, which must hold in all transitions decomposed from a higher level STA.

5 An Example

To provide a basis for comparing statecharts and STAs, it is useful to look at a simple example. The device OP given by Figure 5 and its corresponding description are from Gordon's paper [4].

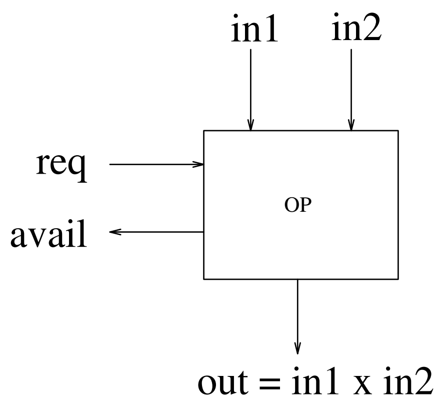

Figure 5: The binary device OP

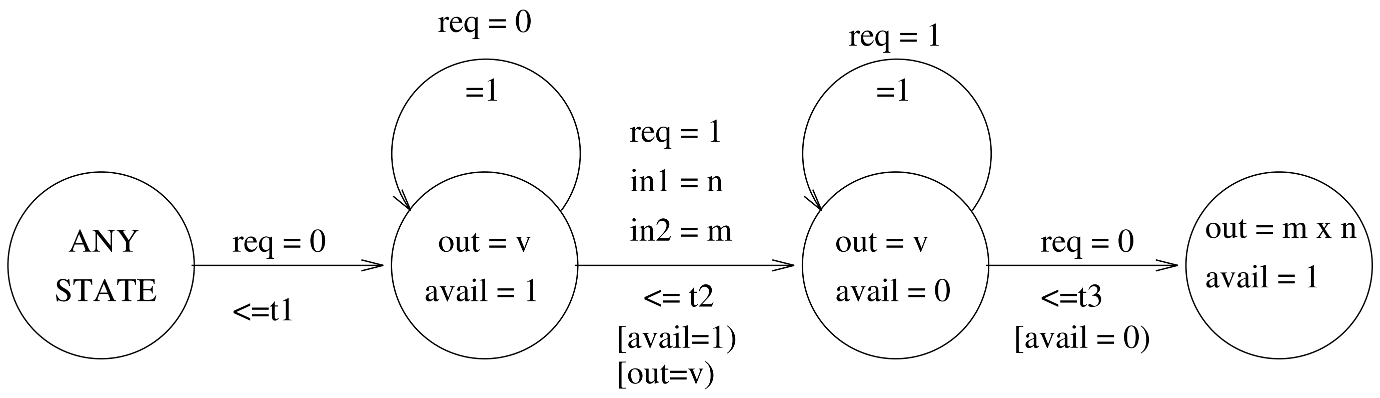

Figure 6: A STA specification of OP

OP is a device which calculates the operation **x**, given two operands **in1** and **in2**, and produces output **out**. (The **x** can be any binary operation.) Two signals **req** and **avail** control its execution. **Req** is provided by the user (or "client") and **avail** is driven by OP.

Whenever **avail** is 1, the client can initiate a request to perform the operation by setting **req** to 1. The client must keep its inputs stable and **req** at 1 until **avail** is set to 0. If the client continues to keep **req** at 1 then the output remains stable. After the client drops **req** to 0, the new output is available when **avail** goes to 1. The output then stays stable while **req** is 0.

Figure 6 gives five STAs which must hold in the implementation of the device. The left-most one says that from any state, if the **req** signal remains low, then the system must move into a state where **avail** is high in less than **t1** time units. The two main other timing restrictions are that the system receives the inputs and therefore moves into a state where **avail** is low in less than **t2** units after **req** goes high and that once **req** goes low again, the system will produce valid output and raise **avail** in less than **t3** time units. (The notation **[p)** means **p** will hold true for all states in the transition except the last one. The first state is included in the sequence of states which must satisfy the predicate below the line in an STA.)

In attempting to express the behaviour of this device in a statechart it is more intuitive

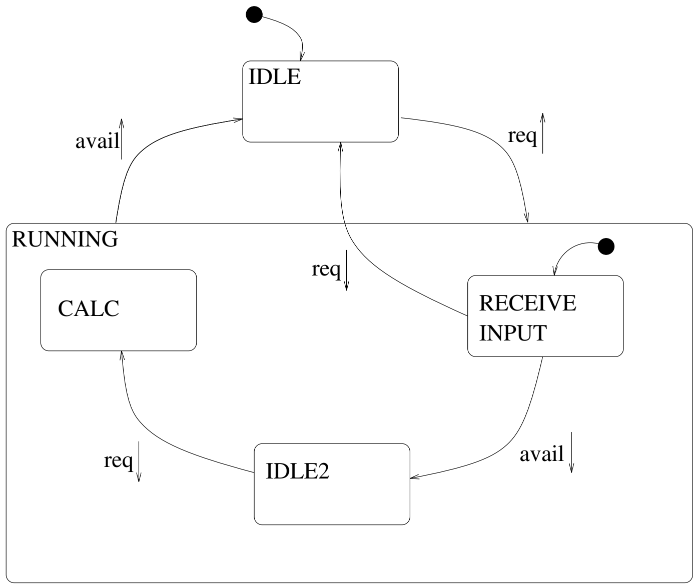

Figure 7: A statechart specification of OP

to think of the events which cause transitions to happen. Therefore we look at events like **req**↑ when the **req** signal goes high. Figure 7 is a statechart specification for OP. (There are many statecharts which could be used to describe the execution of this device.)

This second specification contains less information about the state of the system at each point since no predicates are attached to the states and we can only give them meaningful names. To associate these states with the operation of the device, some invariants on the control signals which would hold in these states are:

	avail	**req**
IDLE	1	0
RECEIVE INPUT	1	1
IDLE2	0	1
CALC	0	0

Time is passing and possibly work is being done (except for in the IDLE states) within the state and the transitions take no time.

We also need to distinguish between internal and external events. The **req** signal is controlled by the client but the changes to **avail** must be internally generated. This information would appear in a more detailed statechart where the states RECEIVE INPUT and CALC were decomposed further. For example, the last action in RECEIVE INPUT would be to generate the internal event **avail**↓. The constraints on how long the system can take to perform these tasks are also not expressed.

The set of STAs gives no indication of what the system should do if the client lowers **req** before the system has raised **avail** when reading in the inputs. Procedurally, it makes sense, to return to the IDLE state since the output has not yet been changed and in doing this, we satisfy the initial STA which must hold from any state.

The statechart highlights one constraint given by the STAs which is not immediately apparent from their visual representation. In the model of operation which the statechart presents, we can see that in attempting to check the first STA, which applies to "any state", it is only relevant in the three states of RUNNING because in IDLE the output is already valid and **avail** is high. From RECEIVE INPUT, when **req** goes low, we return to IDLE because the inputs have not yet been read in. This transition must take less than **t1** time units to satisfy the first STA. From IDLE2, when **req** goes low, the system will attempt to calculate the output going through CALC. This calculation phase is bounded in the last STA by time **t3**, but it must also be limited by **t1** since both assertions apply to the same path through the statechart in the execution of OP. Therefore **t3** must be less than or equal to **t1**. (The first STA has wider applicability than the last one therefore **t1** could be larger.)

6 Differences Between STAs and Statecharts

Many of the differences between statecharts and STAs are already apparent from the example. This section will briefly summarize these:

Events vs Conditions In statecharts, transitions can be triggered by both events and conditions. STAs rely entirely on conditions. Conditions are appropriate if, for example, as in hardware, an input is always available on an input line, but in other cases it may be more natural to speak of events, like a button being pushed. It is also possible to show inputs persisting over time by using events when they change their values. In the state RECEIVE INPUT of the example, the **req** signal must stay high otherwise the transition triggered by **req**$\downarrow$ would be taken.

Visual Notation The actual arrangement of states in a statechart has meaning, whereas the ideas of concurrency and modularity are not expressed visually with STAs. The example presented above shows how it is possible to group the states RECEIVE INPUT, IDLE2, and CALC in the state RUNNING.

State Conditions STAs allow the specifier to give a meaning to a state in terms of predicates which must hold, rather than just a mnemonic name.

Synchronization Between Components In statecharts, it is possible to synchronize concurrent components using events, conditions and actions. While concurrency is not really represented in STAs, the only tools available for synchronization are predicates on the variables.

Passage of Time In statecharts, time passes, and possibly work is done, while the system is in a state, since the transitions are instantaneous. In STAs, the state describes a moment in time, and time passes and work happens on transitions. Although one STA could be true at several moments in time. These are dual notations in this sense and each developer will have a different preference on the notation used for the passage of time.

Incrementing Time As stated earlier, statecharts have no particular model of time associated with them, whereas, STAs have a very clear synchronous model. STAs also can have predicates on the amount of time a transition can take. In statecharts, it is possible to check how long the system has been in a particular state using conditions. It is also possible to schedule events to happen in the future.

7 Issues for Connecting Real-time Specification and Verification Techniques

The purpose of this study was to look at how a real-time specification technique differs from a method designed for verification with the aim of connecting a readable specification formalism with verification. The results which can be presented at this time are a list of issues, drawn from looking at both techniques, which need to be resolved in a formal specification method which includes verification as an integral part.

It should be first noted though that different types of systems will have different requirements of the specification technique. In describing a general notation and method, it may become less useful for certain problems. However, the usefulness of having a common standard which allows for comparison and integration between components of a project or projects themselves may outweigh this disadvantage.

Following is a list of these issues and a brief discussion of each. This list is not intended to be exhaustive and the usual requirements for a specification technique like readability, expressibility and support tools are not examined here. (See [2][16] for a report on these.) This list is based on items particularly concerning an approach which integrates verification into the design process for real-time systems.

Does it express operation or assertions? The fundamental difference between the two notations is the point of view from which they approach the system. STAs are limitations on the implementation's operation whereas statecharts are high level programs for the operation of the system. Timing conditions are a good example of this. A STA on the number of steps in a transition is a constraint on how much time the system can take to complete the task described by the transition. A statechart condition checking how long the system has been in a state which triggers a transition means the system should follow that path when the condition becomes true, regardless of whether it has finished the task assigned in that state.

Can it handle complete system description at any level? Rather than speak of either "hardware" or "software" specification, real-time systems are usually a combination of the two working together. From this point of view, the method should be equally applicable to both, and, as the specification becomes more detailed, the developer can decide which parts will be implemented in either hardware or software. Ideally, the process should begin with a very high-level description which can be decomposed and have more details added to it using the same notation, right down to the gate level for hardware, and the microcode level for software. It should also be possible, to move in the other direction and compose parts for reusability or to look at higher levels of abstraction. This is also the direction verification would take after a level has been given more details.

Does it have a formal model of time? A major difference in these two methods is the way time passes. Statecharts are good at specifying required reactions to events, whereas STAs give a more precise idea of how many time units will pass. In real-time systems, both these elements are essential. It should be possible to state that the time it takes for a system to react to an event will satisfy a predicate on time. We might also want to say that the time it takes a particular process to complete its execution is limited by a certain time. Notice, that these statements do not say "within" a limited time since there may be cases requiring a minimum amount of time before the system should react. The ideas of timed transition systems [11] and timed statecharts [15] will be explored to see if these modifications of statecharts provide a sufficiently clear model of time which can be used to describe the operation of the system at all levels.

Can it integrate data and control descriptions? This is an area where neither of the methods score particularly highly. Statecharts can really only describe control of the system with a limited ability for assignment to variables. The tool STATEMATE completely separates control and data descriptions, although data process can be associated with being in a particular state. STAs integrate the two aspects of the system by having predicates on the state which are really predicates on the variables at the current time in the system. Any kind of large data description would become unmanageable in this notation. The argument against STATEMATE's division of interests is that the line between what is control information and what is data is not clear in most systems and each developer would make a different division.

Is it a visual notation? To make specification and accompanying verification more amenable to industry, a specification technique should use a meaningful, compact, visual notation in which modularity and concurrency can be represented, and which it is possible to animate.

8 Future Work

8.1 A Specification Formalism

Looking beyond statecharts and STATEMATE in particular, examining these two notations has provided many ideas for what needs to be in a formal specification notation used for verification. A formalism satisfying these conditions will probably combine ideas from both statecharts and STAs. In particular, this notation should be able to express:

- Hierarchy of states
- Concurrency of operation

These first two points should be represented visually.

- Operation so the specifications would be executable (this includes having default states and history states or some form of memory, although these are really just a notational convenience). A benefit of taking the operational approach is that more automatic techniques such as symbolic simulation or model checking can be applied to the specification to test conditions, particularly timing conditions.
- Events and conditions to trigger transitions
- Communication - ideally both broadcast and some form of local communication or the ability to hide internal events in a module. Local communication would make it possible to combine specifications, worrying only about the interface between the two, rather than also having to consider if the same names are used for local events or variables.
- Timing conditions and constraints
- Data and control specifications
- Axiomatic proof rules for composition

It should be possible to develop a notation which includes all these ideas. Given the two possible ways of indicating where time passes, the notation should be able to be used in either form (ie where time passes within a state or where time passes on a transition). The two notations would be complete duals and have the same semantic meaning.

8.2 Embedding the Formalism within HOL

The semantics of this formalism can be developed using HOL as a research tool to check their preciseness, and to check certain properties to ensure that the semantics give the correct meaning. Later it can be used to verify mechanically that lower level descriptions imply the same behaviour as a more abstract view of the system.

The ability to quantify over functions in higher order logic makes it possible to use functions in the predicates constraining the system, like those used in STAs. The type of hierarchy of states used in statecharts can also be represented using higher order functions. This could be particularly useful in data process specifications. It would also be possible to take advantage of inductive proofs in cases where there are multiple instances of a type of object in the system. Another benefit to using a theorem prover is to link this type of "control" specification with more detailed data specifications. Finally, given that we have chosen to use an operational system description, it will be within the theorem-prover that we can look at constraints which must apply at more than one state within the system, such as the predicate on "any state" given in the STA of the example.

9 Summary

The main goal of this work is to integrate a more readable specification language with formal verification. Accomplishing this will make it possible for verification to start earlier in the system development process. We present in this paper a study of two existing notations from the two areas which we are trying to connect, to determine what some of the issues are in working towards this main goal. We showed that statecharts offer the advantages of readability which we desire and the availability of a commercial tool to support the notation, however, their model of operation and timing is different from STAs which have the primary purpose of verification.

We can conclude by saying that several issues which need to be examined when working towards a specification formalism for verification are:

- Does it express operation or assertions?
- Can it handle complete system description at any level?
- Does it have a formal model of time?
- Can it integrate data and control descriptions?
- Is it a visual notation?

Combining a readable front-end specification formalism with a theorem- prover and possibly other verification tools will make formal methods more appealing to industry.

10 Acknowledgements

This work was completed while I was funded by a National Science and Engineering Research Council Post Graduate Scholarship and was written up while working at SRI International, Cambridge. I am indebted to the Integrated Systems Design Laboratory at UBC, the Hardware Verification Group at Cambridge, and my colleagues at SRI for interesting discussions where many of these ideas originated.

References

[1] Victor A. Carreno. Specification of real time reactive systems using state transition assertions. a research proposal. Computer Laboratory, Cambridge, UK.

[2] Paul C. Clements, Carolyn E. Gasarch, and Ralph D. Jeffords. Evaluation criteria for real-time specification languages. Technical Report NRL Memorandum Report 6935, Naval Research Laboratory, February 1992.

[3] 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.

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

[5] Mike Gordon. A proof generating system for higher-order logic. Technical Report No. 103, University of Cambridge Computer Lab, January 1987.

[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] David Harel and et al H. Lachover. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[11] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. Technical Report TR 92-1263, Department of Computer Science, Cornell University, Jaunary 1992.

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

[13] Victoria Stavridou Jonathon Bowen. Safety-critical systems, formal methods and standards. Technical Report No. PRG-TR-5-92, Oxford University Computing Laboratory, 1992.

[14] Jeffrey Joyce. *Multi-Level Verification of Microprocessor Based Systems.* PhD thesis, University of Cambridge Computer Laboratory, 1989.

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

[16] Place, Good, and Tudball. Survey of formal specification techniques for reactive systems. Technical Report CMU/SEI 90-TR-5, Carnegie Mellon University/Software Engineering Institute, May 1990.

[17] P.J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.