A Dynamic Assertion-based verification platform for UML Statecharts over Rhapsody

A. Banerjee  S. Ray  P. Dasgupta  P.P. Chakrabarti  
Department of Computer Sc. & Engg. IIT Kharagpur, India - 721302
S. Ramesh  P. Vignesh V. Ganesan
General Motors India Science Labs

Abstract— For quite some time, the Unified Modeling Language (UML) has been adopted by designers of safety critical control systems such as automotive and aviation control. This has led to an increased emphasis on setting up a validation flow over UML that can be used to guarantee the correctness of UML models. In this paper, we propose a dynamic Assertion-based verification (ABV) framework for validation of UML Statecharts over the Rhapsody platform of i-logix. We present an extension of Linear Temporal Logic (LTL), named Action-LTL that allows assertions to be specified over data attributes and events of UML models. We present a methodology for automatic generation of Rhapsody Statecharts from Action-LTL specifications. These generated Statecharts are added as simulation observers to an existing UML model to detect specification violations during simulation. In view of the capacity limitations of existing formal assertion-based verification tools, we believe that our methods are of immediate practical value to the UML-based design community.

I. INTRODUCTION

Assertion-Based Verification (ABV) is assuming a significant role in the design validation flow of chip design companies. In recent times, active participation from the design and EDA industries have led to the adoption of several formal languages for assertion specification. These include PSL [9] and System Verilog Assertions (SVA) [12].

Present day assertion-based verification falls in two broad categories - static and dynamic. In static verification, the design-under-test (DUT) is modeled as a finite state machine and assertions are written to capture the correctness requirements. These assertions are then exhaustively checked on the finite state machine model of the design using graph-theoretic algorithms. Static verification techniques have gained immense acceptance in the verification community (more in the hardware domain) for its ability of automatic and exhaustive reasoning. However, the major impediment that has eluded researchers is the problem of state space explosion due to the inherent capacity issue involved in representing a finite state structure of the DUT. For validating large and complex designs, dynamic verification has therefore been the preferred choice due to scalability factors. In dynamic verification, the assertions are written as earlier, and added as checkers to monitor the response of the design during simulation.

One of the main requirements of dynamic ABV is the concept of an interface (Figure 1), through which the variables of the design are made visible to the assertion checker. The interface plays the role of establishing the communication between the design-under-test and the verification framework without having to change the source code of the design. Languages like System Verilog [12] have built-in constructs for defining the interface through which one binds the assertion checker with the appropriate design variables.

In the context of software, where the state spaces are far larger than those for hardware systems (software systems are typically infinite space), the application of static techniques have obvious limitations. To adopt assertion-based verification in software, the dynamic approach is certainly the preferred choice. One of the major difficulties in using dynamic verification is the need for defining a software interface for accessing design variables for purpose of validation. This is due to the fact that to describe correctness properties of software, one needs handles to both program variables and internal events, and this cannot be easily done without adding appropriate hooks inside the source code. These hooks may introduce bugs into the software specifically when real time safety critical systems are being designed.

For the past few decades, the Unified Modeling Language (UML) [7] has been one of the preferred choices in the software community for the design of a wide variety of applications at a higher level of abstraction, ranging from automotive control to medical instrumentation. The advantage of developing a dynamic ABV-based framework over UML is the fact that UML provides features for defining and accessing events and data attributes of a design at a higher level of abstraction. These features can be utilized for defining correctness specifications and checking them without much additional overhead since UML simulators provide sufficient external handles for accessing variables and events relevant to the assertions. One of the main difficulties, however, in developing any validation platform over UML is the fact that the dynamic semantics of UML is specified only informally, and several companies have come up with their own semantics. Our work is based on the semantics of Statecharts, as employed by Rhapsody [5].

The main contribution of this work is to formalize and develop a dynamic assertion-based verification platform for verifying behavioral properties of communicating concurrent systems described using Statecharts in Rhapsody. In Rhapsody, communication among concurrent Statecharts proceeds through actions (events) which can represent function calls, requests and acknowledgments, etc. These communications can be data-dependent and carry data on its channels. To describe behaviors over such systems, we need a specification language that can describe properties over data attributes and events as well. Property specification languages that have been widely used in the verification community are pre-dominantly either state-based or event-based. Our work is inspired by the fact that for the
verification of software, one needs to specify both state information and events (communication among components). For example, the Local Interconnect Network (LIN) [4] protocol specification has the following requirement: In slave node, detection of break/synch frame shall abort the transfer in progress and processing of the new frame shall commence. As this shows, both states and events (break/synch event) are required to capture the desired behavior.

The fundamental challenges behind developing a dynamic assertion-based validation platform over a modeling tool like Rhapsody are as follows:

- The computation model underlying a set of Rhapsody processes has to be formalized.
- The assertion language to be used for formal specification of properties must be able to express both states and events. The semantics of the language to be defined carefully in accordance to Rhapsody definitions of states and events.
- The assertions must be easily integrable into an already existing Rhapsody design environment. This requires one to translate the formal specifications into Rhapsody constructs so that the assertion monitors can work synchronously with the design.

As explained above, the Rhapsody simulation semantics plays a crucial role for defining the semantics of the specification language. This motivated us to extend LTL [8] with the ability to express both states and events simultaneously. It may be noted that Action-LTL is not a new language altogether. Our main focus is in developing a dynamic ABV platform over Rhapsody. The main contributions of this paper are as follows:

- We have proposed a new language, Action-LTL, for describing behavioral requirements over states and events of UML Statechart-based designs.
- We have developed algorithms for translating Action-LTL specifications into observers for monitoring correctness behaviors during simulation. The concept of generating simulation monitors from formal specifications is not new. However, the novelty of our approach is in allowing to translate the formal requirements into Rhapsody Statecharts that can be integrated into an already existing Statechart-based UML design in Rhapsody without any extra overhead. The task of tracking events and data variables can be easily done through the Rhapsody event manager.
- We have developed the complete property suite of the Local Interconnect Protocol (LIN) [4] in Action-LTL and tested our concept of dynamic Assertion-based verification on Rhapsody-based LIN models developed by General Motors.

II. RELATED WORK

The idea of combining state-based and event-based formalisms for the verification of systems is not new. A detailed discussion of the different approaches in combining states and events for property specification can be found in [3]. One of the most recent developments in this direction has been that proposed in [3], where the authors present a static verification framework in which both state-based and event-based properties can be expressed, combined and verified. The specification logic, SE-LTL is a state-event derivative of Linear Temporal Logic (LTL) [8].

A number of different approaches for the verification of UML Statecharts have been developed. The most prominent of these approaches is ROOM [11] which has been inducted into the real time extension, UML/RT of UML. However, non-functional system properties with respect to dynamic behavior are only rudimentarily supported in UML/RT. In recent times, there has been some research on formal verification for validation of UML Statecharts. A detailed discussion can be found in [1]. The main principle behind this approach is to translate the Statecharts into some format amenable to formal verification tools, and on the other hand, use the power of temporal logics like Computation Tree Logic (CTL), Real Time CTL (RTCTL), clocked CTL (CCTL) and Timed CTL (TCTL) for specifying behavioral requirements. A model checker is then invoked to establish the truth or falsify the specification on the model. Formal verification of UML models has been reported in [10]. Graf et al. [6] perform a verification of UML models extending an automata-based validation framework, called IF, and employing model checkers.

III. WHY DO WE NEED ACTION-LTL?

The main focus of this work is to develop a dynamic ABV platform for validating behavioral descriptions of UML models in Rhapsody. In this context, the primary motivations for this work are:

- Rhapsody has no support for OCL.
- Using a purely formal verification approach would be non-scalable as in case of most formal verifiers.
- Using any temporal logic for assertions will not work unless its semantics is not subsumed by Rhapsody semantics. In addition, the verification flow has to be redefined.
- Coding the high-level correctness requirements of a system directly as Statecharts is non-intuitive and not easy.
- Checkers for the properties have to be automatically synthesized in accordance to Rhapsody semantics.

This motivated us to extend LTL to comply to the simulation semantics of Rhapsody. The logic, Action-LTL is quite generic and allows one to specify requirements over UML Statecharts using a combination of data attributes and events as in [3]. Since our primary target is to set up a validation platform for UML Statecharts in Rhapsody, events in our language are provided an extended semantics that correspond to the different kinds of Rhapsody events. Our verification approach, unlike that in [3] is simulation-based, and hence more scalable and attractive. The additional novelty is the way in which we efficiently integrate the Action-LTL specifications into Rhapsody.

IV. RHAPSODY SEMANTICS OF STATECHARTS

Our validation framework is based on Rhapsody. We describe below briefly the concept of Rhapsody Statecharts. A detailed discussion of the Rhapsody semantics of Statecharts can be found in [5].

An object-oriented system is composed of classes. A Statechart describes the modal behavior of a class, i.e. how it reacts to messages it receives by defining the actions taken and the new mode entered. A class can have an associated Statechart describing its behavior.

A Statechart consists of 3 types of states, OR-states, AND-states and basic states. The OR states have substates related to each other by exclusive or, AND-states have orthogonal components that are related by and, while basic states have no substates, and are the lowest in the state hierarchy. Figure 2(a) shows the hierarchy and the three types of states that can be used in a Statechart. States S, B and C are OR-states, state A is an AND-state and states B1, C1, C2 and E are basic states. When building a Statechart in Rhapsody, an additional state is created implicitly, called the root state (the highest in the state hierarchy). In Example 1, the root state has S as a substate. The general syntax of a transition in a Statechart is m(c)/a (all optional) where m is the message that triggers the transition, c is a condition that guards the transition being taken unless it is true when m occurs, and a is an action that is carried out when the transition is taken. An example event-triggered transition is shown in Figure 2(b).
In Rhapsody, there is a single trigger, which can be an event or a triggered operation. Events are for asynchronous communication and triggered operations for synchronous communication. It is possible to have a transition without a trigger, called a null transition. Another kind of message used in Rhapsody is a primitive operation, which corresponds to an invocation of a method call in the underlying programming language. A trigger can also be a special event timeout, or \( \text{null}(t) \), where \( t \) is the time (milliseconds) until the event occurs.

Besides actions that appear along transitions, they can also appear associated with the entrance to (Entry action) or exit from (Exit action) a state, (any state, on any level). Actions associated with the entry of a state \( S \) are executed in the step in which \( S \) is entered, as if they appear on the transition leading into \( S \). Similarly, actions associated with the exit from \( S \) are executed in the step in which \( S \) is exited, as if they appear on the transition exiting from \( S \). A state can also have static reactions (SRs).

1) Events: Events are used to describe asynchronous communication. Each class defines the set of events it can receive. The main motivation for using events is that the sender object can continue its work without waiting for the receiver to consume the event. Events are sent by applying the GEN method to the destination object: \( O \rightarrow GEN(event(p_1;p_2;\ldots;p_N)) \). The sending object should be able to refer to the destination object \( O \). Here \( p_1;p_2;\ldots;p_N \) are event parameters that match the event's formal arguments (data). The GEN method creates the event instance and queues it in the queue of \( O \).

Events in Rhapsody are managed via queues (FIFO). In a single thread system, a single event queue exists. Events are dispatched by a dispatcher. Once an event gets to the top of the queue, the dispatcher delivers the event to the proper object. On receiving an event, an object processes it after which the event is deleted.

2) Steps and Microsteps: The behavior of a system described in Rhapsody is a set of possible runs. The semantics of our proposed language is defined with respect to a Rhapsody run. A run consists of a series of detailed snapshots of the system’s situation. Such a snapshot is called a status. The first in the sequence is the initial status, and each subsequent one is obtained from its predecessor by executing a step (Fig 3) triggered by the dispatch of an event. Each step is composed of microsteps. The system, being in a certain status and as a response to an event, undergoes a series of microsteps, until it reaches a final status, and at which point it is ready for the next event dispatch [5]. A special case is that of null transitions, that is, transitions without a trigger, and these can be taken spontaneously.

A status contains information about the present condition of all the objects in the system, history information for states, values of data members, connections of relations and aggregations and event queues. The concept of status and step are of utmost importance since properties in Action-LTL are expressed over data values at status points and events at steps.

3) The Simulation Step Algorithm: In this section we present a schematic description of the algorithm that executes a step. This is of utmost importance for our work, since our property checkers are defined in accordance with this semantics. At each step, if the event queue is non-empty, the top event is taken and dispatched to its destination. For a dispatched event, the following actions are taken:

- Determine the transitions/static reactions (non-conflicting) that will fire in response to this event.
- For each transition, do
  - Perform the exit actions of the exited states and update their histories.
  - Perform the actions on the transitions and Static reactions.
  - Perform the entry actions of the entered states.
  - Update the active configuration.

Dealing with null transitions: After reacting to an event, the Statechart may reach a state configuration where some of the states have null transitions. In such a case further steps need to be taken until the Statechart reaches a stable state configuration where no null transitions are enabled.

The semantics of each step follows the Rhapsody Run-to-completion (RTC) rule. It means that an event can only be dispatched if the processing of the previous event is fully completed (including all microsteps and null transitions).

V. ACTION-LTL

In this section, we first describe our assumed model of Rhapsody Statecharts, followed by the syntax and semantics of Action-LTL. The following assumptions are made on the Rhapsody execution model:

- We consider a single thread, and hence, a single event queue (FIFO) for the entire system consisting of multiple Statecharts. This ensures that we do not have interleaving of events.
- Data values and data changes are sampled only at the end of each Rhapsody step at each Rhapsody status.
- We do not take account of data changes due to null transitions, or precisely, changes that occur in the microsteps, are not considered, unless the microsteps stabilize to a system step.
- The propagation of events inside a state machine and among state machines is considered loss-free [5].

A. Action-LTL: Syntax and Semantics

The semantics of Action-LTL is defined with respect to a run \( \pi = \langle \eta_0, \eta_1, \eta_2, \eta_3, \ldots \rangle \) of Rhapsody, where \( \eta_i \) corresponds to a Rhapsody status (\( \eta_0 \) being the initial status), and \( \eta_i \) are events. The concept of status, steps, and events have been explained already. We denote by \( \pi^t \), the suffix of \( \pi \) starting from the status \( \eta_0 \).

As mentioned already, a status contains information about all the objects in the system, history information for states, values of data members, connections of relations and aggregations and event queues. Action-LTL properties are expressed over data values at these status points and over events that are dispatched at each step (between two status points). The values of the data attributes available at a status point are the ones corresponding to the stable states of each object Statechart in the system. The following example illustrates the concept of status and steps.
Example 1: Consider the system in Figure 4 consisting of objects M1 and M2. x is a data member of M1, y is a data member of M2. e is an external event which remains at the top of the event queue at the start of simulation. f, e2, e3 are events. State A is the initial state of M1 and State P is the initial state of M2. Hence, the initial status of the system consists of (M1.A, M2.P). Consider the following execution of the system: M1 receives event e from the user. The transition from state A to state B is fired. The exit actions of state A and the entry action of state B are executed. This transition also involves sending event f to object M2 (by placing a new event in the event queue), as specified by the action M2 → GEN(f). Once the transition to state B of M1 is completed and the system is stabilized, the status variable shows (M1.B, M2.P). In the next step, event f is removed from the event queue, and is dispatched to object M2, causing it to take the transition from its state P to state Q. A sample snapshot of the Rhapsody execution trace of the system (consisting of status and steps) is shown in Figure 5. □

![Fig. 4. An Example System](image)

![Fig. 5. The Rhapsody Execution Trace](image)

We now state the syntax and semantics of our proposed language Action-LTL. Given a system consisting of Objects O1, O2, ..., On, each having its own set of data attributes Di, the syntax of Action-LTL is defined over the set of data attributes Di and the set of events E. We first define the concept of a data predicate.

Definition 1: A data predicate is an expression of the form (exp) where (exp) is any well formed Rhapsody expression involving class attributes, constants, variables local to the formula, arithmetic and logical operators. The symbol (exp) may assume any form from the set \{ ≤, ≥, =, ≠, <, >, \}. □

A data predicate can have value true or false. Action-LTL properties allow users to use local variables in the properties (as in SVA) [12]. This is a major difference to ordinary temporal logics like LTL, and is a very useful feature in the context of software. Consider a property for a FIFO as follows: If X and Y are any two data items such that X was pushed before Y, then X will come out of the queue before Y. In this property, the values of X and Y are not important – they are placeholders for any two data items entering and leaving the queue. The concept of local variables in Action-LTL enables us to express such properties formally. This feature makes our language distinctly more expressive. In addition, the events allowed in Action-LTL have a semantics that follows the underlying Rhapsody computation model. Hence, properties in Action-LTL can express a wide range of requirements. The syntax of Action-LTL is as follows:

- Oi.e is an Action-LTL formula, (here Oi.e stands for the event dispatched to object Oi).
- If f and g are Action-LTL formulas, then so are ¬f, f and g, next f, always f, future f, until g.
- The normal laws of Boolean Algebra apply over Boolean operators in Action-LTL formulas (DeMorgan’s law etc.). In addition, we also use the usual short forms: future/for [true until f], and always/for ¬future¬f.

The semantics of Action-LTL formulas is as follows:

- \( n_k \models O_i.p \) iff \( O_i.p \) is true in the status \( n_k \).
- \( \pi \models O_i.p \) iff \( O_i.p \) is true in the first status of \( \pi \).
- \( \pi \models O_i.p \) iff \( \eta = \pi \) and \( \eta \) is dispatched to \( O_i \) at the first step,
- \( \pi \models \neg f \) iff \( \pi \not\models f \)
- \( \pi \models f \) and \( g \) iff \( \pi \models f \) and \( \pi \models g \)
- \( \pi \models \text{next } f \) iff \( \pi \not\models f \)
- \( \pi \models \text{future } f \) iff \( \exists i, \pi_i \models f \)
- \( \pi \models \text{always } f \) iff \( \forall i, \pi_i \models f \)
- \( \pi \models f \) until \( g \) iff \( \exists i \) such that \( \pi_i \models g \), and \( \forall 1 \leq j < i, \pi_i \not\models f \)

Example 2 explains the semantics of Action-LTL.

Example 2: Consider the system in Figure 4. Below, we present some correctness requirements to be satisfied by this system and the corresponding Action-LTL encodings.

- \( M_{1.x} = 0 \): Does \( M_{1} \) have \( x=0 \) in the initial status?
- \( M_{1.x} = 0 \) and next \( M_{2.y} = 0 \): Is \( x = 0 \) in \( M_{1} \) true in the initial status and followed by \( y = 0 \) in \( M_{2} \) in the next status?
- future \((M_{1.x} = 1 \text{ and } M_{2.y} = 1)\): In some future status, can we have \( x = 1 \) in \( M_{1} \) and \( y = 1 \) in \( M_{2} \)?
- \((M_{1.e} \text{ and } M_{1.x} = 1)\): When the event \( e \) is dispatched to \( M_{1} \), does \( M_{1.x} \) equal 1?
- always \((M_{1.x} = 1 \text{ and } M_{2.y} = 1)\): Always when \( M_{1.x} \) is 1 and \( f \) is dispatched to \( M_{2} \), in the next state \( M_{2} \) moves to a state where \( y \) equals 1.

As the above properties demonstrate, several interesting properties involving state attributes and events may be expressed. □

VI. THE VERIFICATION PROCEDURE

For validating a system S consisting of a set of communicating concurrent objects, each having a Statechart capturing its behavior, the overall approach is:

- Encode the properties of S in Action-LTL.
- Each Action-LTL property is translated into a Statechart (we call these observers).
- The observer Statecharts are added into an existing Rhapsody model of S.
- Overload the event dispatcher (overloading the RICGEN macro in Rhapsody) to ensure that at every step, a copy of every dispatched event is also delivered instantaneously to the observer Statecharts.
- Run Rhapsody simulation.
- The observers sample data attribute values at every status point and check for specification violations.

Among the above steps, the only non-trivial step is the translation of Action-LTL specifications into Rhapsody Statecharts. This is also the most important hallmark of our approach since it ensures that our proposed methodology can be seamlessly integrated into any existing Rhapsody project without much effort. In the following subsection, we describe the translation algorithm.
A. Translation Procedure

Our algorithm for observer Statechart generation takes as input an Action-LTL formula and outputs the corresponding Rhapsody Statechart. In our method, we assume:

- A checker of any property invokes a primitive operation \( \text{Err}() \) to file an error report whenever a specification violation is detected.
- The primitive operation \( \text{SaveH}((\arg)) \) saves the present context specified in the argument list \( \{\arg\} \). \( \{\arg\} \) represents a sequence of one or more data attribute(s) of the model.
- The checker implicitly maintains image variables for each data attribute by using the history feature of Rhapsody Statecharts.
- Checkers can have local variables.

In the following discussion, we first present the observer Statecharts for the basic formula constructs \( p \) and \( O_i, \eta \). The observer Statecharts of temporal formulas like next \( f \), always \( f \) and \( f \) until \( g \) are constructed using the basic observers and some additional constructs, which we explain later. Figure 6 shows the Statecharts for \( p \) and \( O_i, \eta \) where \( p \) is any data predicate and \( \eta \) is an event. We write \( E \) to denote any event defined in the model and \( E - \eta \) to denote any event in the model other than \( \eta \).

![Fig. 6. Observers for \( p; O_i, \eta \)](image)

Each observer Statechart in Figure 6 has two junction connectors, one placed vertically below the other inside the state at the top of state hierarchy (child of root state). These two connectors have two different roles: control reaches the upper one when a witness of the formula is found, whereas control reaches the lower one when a violation is detected. From each such connector, one dangling transition has been drawn. These transitions play the role of ‘stitching’ Statecharts of two component subformula together to produce the Statechart for a compound formula. This stitching technique has been illustrated in Fig 7, Fig 9 and Fig 10.

We first briefly explain the intuitive idea behind the observers for the basic constructs shown in Fig 6. When control enters into the Statechart monitoring data predicate \( p \), it reaches a state named \( \text{Check}_p \). It is an OR state and thus enters into its default sub-state \( \text{check} \). From this state, control reaches any one of the two junction connectors depending on the truth value of the data predicate \( p \). If \( p \) is false, the observer files an error report. The checker for the event \( \eta \) works similarly, the only difference being that it branches upon availability of event \( \eta \) and saves the present status during the transition.

Fig 7 shows the Statechart for Action-LTL formula of the form \( f \) until \( g \). The checker for the until property works as follows. It first checks if property \( g \) is true. If yes, then it declares a witness, else checks if \( f \) is true. If yes, then it checks the next status analogously. Otherwise it files an error report. Fig 8 gives the Statecharts for next and always properties. Fig 9 gives the Statechart for composition rules for Boolean connectives. In each of these Statecharts, we assume that the observer Statecharts of the two Action-LTL formulas \( f \) and \( g \) are already built. In Fig 7 and in Fig 8, the label \( E/\text{SaveH}(\arg) \) on some of the self-loops is put into a pair of angular brackets to indicate that these self-loops are event-triggered. In these cases, self-loops marked with such labels indicate that the corresponding transition waits for an event to occur. In the remaining cases, such self-loops go unlabeled meaning default transitions (the translation algorithm is written accordingly).

![Fig. 7. Observer for \( f \) until \( g \)](image)

![Fig. 8. Observers for next \( f \); always \( f \)](image)

We give a recursive algorithm that takes as input an Action-LTL formula and generates the corresponding Statechart by decomposing the formula into basic formulas and using the respective procedures described above. Algorithm 1 outlines the top-level algorithm.

**Algorithm 1: Procedure GenerateStatechart**

\[
\text{GenerateStatechart(Action-LTL property: } \mathcal{L})
\]

\[
\begin{align*}
\text{case } \mathcal{L} \text{ of the form:} \\
&O_1, p: \text{return Statechart of } p \quad \text{/*Fig 6.1*/} \\
&O_1, \eta : \text{return Statechart of } O_1, \eta \quad \text{/*Fig 6.2*/} \\
g \text{ until } h: \text{ build from Statecharts of } g \text{ and } h \quad \text{/*Fig 7*/} \\
&\quad \text{if any one of } g \text{ or } h \text{ involves any event,} \\
&\quad \text{keep the label within angular brackets on the self-loop,} \\
&\quad \text{else make it a default transition.} \\
\text{next } g: \text{ build from Statechart of } g \quad \text{/*Fig 8.1*/} \\
&\text{always } g: \text{ build from Statechart of } g \quad \text{/*Fig 8.2*/}
\end{align*}
\]
if \( g \) involves any event, 
keep the label within angular brackets on the self-loops, 
else make it at a default transition.

\( g \) and \( h \): build from Statecharts of \( g \) and \( h \) /*Fig 9.1*/
\( \neg g \): build from Statechart of \( g \) /*Fig 9.2*/
\( \text{future} \; g \): build from Statechart of \( \neg \) always \( \neg \) \( g \)
end

The working of our algorithm is explained in the following example.

**Example 3:** We show an example of Statechart generation from an Action-LTL formula. Consider a formula \((d \land e) \land \neg m\) where \(d\) and \(m\) are data predicates and \(e\) is an event. The formula can be rewritten as: \(\neg ((d \land e) \land \neg m)\). In the first iteration, the observers for \(d\), \(e\), and \(m\) are built. In the next iteration, the observers for \((d \land e)\), \(\neg m\) are built. This is followed by creation of the observer for \(((d \land e) \land \neg m)\) and finally, \(\neg ((d \land e) \land \neg m)\). Figure 10 shows the Statechart produced as the output of our algorithm. (The node marked T denotes the standard Rhapsody termination construct.)

**REFERENCES**