

A Dynamic Assertion-based Verification Platform for Validation of UML designs

A. Banerjee[†] S. Ray[#] P. Dasgupta[#] P. P. Chakrabarti[#]
S. Ramesh^{*} P. Vignesh V Ganesan^{*}

[†]Advanced Computing and Microelectronics Unit, Indian Statistical Institute

[#]Dept. of Computer Science & Engineering, IIT Kharagpur

^{*}General Motors India Science Lab

ABSTRACT

Capacity limitations continue to impede widespread adoption of formal property verification in the design validation flow of software and hardware systems. The more popular choice (at least in the hardware domain) has been dynamic property verification (DPV), which is a semi-formal approach where the formal properties are checked over simulation runs. DPV is highly scalable and can support a rich specification language. The main contribution of this paper is to build an integrated DPV platform for validation of UML-based designs. Specifically, we present (a) a language, named Action-LTL (a simple extension of Linear Temporal Logic) for writing assertions over data attributes and events of UML models, and (b) an integrated dynamic assertion-verification platform for verification of UML designs. In view of the capacity limitations of existing formal property verification tools, we believe that the methods presented in this paper are of immediate practical value to the UML design community.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software / Program Verification—*Assertion checkers Formal Methods Validation*

General Terms

Verification

Keywords

Software Verification, Dynamic Property Verification, Assertion Checking, UML

1. INTRODUCTION

In recent years, the software development paradigm for safety-critical software is increasingly moving towards a *model-based* development process, in which the largely textual way of requirement capturing is replaced by executable specification models at different levels of abstraction. For the past few decades, the Unified Modeling Language (UML) [31] has been one of the preferred choices for abstract modeling in the software community for the design of a wide variety of applications, ranging from automotive control to medical instrumentation. 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 and is capable of handling the ever-increasing complexity of application software. The main contribution of this work is to formalize and develop a DPV platform for verifying behavioral properties of software systems described using UML Statecharts.

In the last few decades, formal property verification has established itself as an effective validation methodology, both in the hardware and software-verification community for its ability of

automatic and exhaustive reasoning. Researchers have analyzed several historically significant failures and have shown that the use of formal verification could have guarded against these failures. Verification practitioners have also been able to uncover flaws in the specifications of complex protocols and intricate bugs in live designs.

Unfortunately, the exponential increase in complexity and the increasingly distributed nature of functions as used in application software today renders formal verification infeasible because of its inherent capacity bottleneck in handling large systems. In the last decade, a popular validation methodology (at least in the hardware domain) has been dynamic property verification (DPV). DPV is a semi-formal approach where the formal properties are checked over simulation runs. DPV is highly scalable and can support a rich specification language.

The main idea of DPV is based on validating the truth/falsification of the assertions on the basis of the responses of the system under test (SUT) during simulation. To build a DPV platform for validation of UML designs, we need the following:

- A simulator to simulate system descriptions expressed in UML.
- An assertion-specification language for expressing requirements
- An interface to monitor model responses
- A verification engine running on top of the simulator

This work was developed over Rhapsody, a widely used software in the UML community. Rhapsody supports a rich variety of UML modeling constructs and has support for simulation of UML designs.

Property-specification languages that have been widely used in the verification community are pre-dominantly either state-based or event-based. To describe correctness properties arising in the context of software systems modeled as UML Statecharts, one needs to describe properties over data attributes (*state* information) and events (communication among components). For example, the Local Interconnect Network (LIN) [42] protocol specification has the following requirement: *In slave mode, detection of break/synch frame shall abort the transfer in progress and processing of the new frame shall commence.* Evidently, both states (for describing a transfer in progress) and events (break/synch event) are required to capture the behavior.

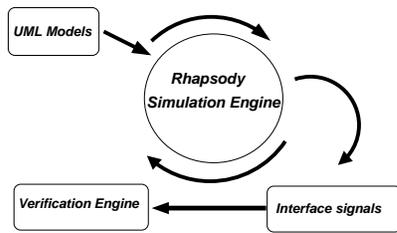


Figure 1: The DPV platform over Rhapsody

To address the above issue, we extended Linear Temporal Logic (LTL) [32] with the ability to express requirements over events and arithmetic and relational queries over data attributes. Our logic is called Action-LTL and is used within our DPV framework for specifying assertions. We call the logic Action-LTL to distinguish it from the standard Propositional LTL used in the hardware-verification community. However, it may be noted that this is *not* a new logic altogether. While the temporal features remain as in LTL, the only major difference is in the definition of events that allow us to capture state-event requirements of protocols, and a support for arithmetic expressions. The additional distinction of Action-LTL lies in its semantics, which is defined in accordance to the simulation semantics of Rhapsody. Figure 1 shows the overall architecture of the DPV platform.

The main idea of DPV is based on examining the behavior of the system under test during simulation. Therefore, one of the primary requirements is to define an *interface* for accessing model variables. The system under test must allow some external hooks to access the model attributes and events visible to the verification engine. Adding hooks inside the source code of a model requires perturbation of source code and may introduce coding and execution overheads. In addition, the granularity of model-attribute sampling is also important: due to the absence of a universal clock (as in the hardware domain) values of data attributes and events read at different points may result in different validation results. To address this issue, we modified the Rhapsody code base and defined an interface with appropriate callbacks to allow us read model responses.

The verification engine plays the most crucial role in the DPV process. Its main task is to read the model-attribute values from the interface at the end of every simulation cycle and evaluate the assertions on the basis of the read values. If any assertion evaluates to true or false, it reports the corresponding status to the user. For assertions that cannot be fully evaluated to success or failure within one cycle, the verification engine prepares the property to be checked during the next simulation cycle and returns control to the simulator.

In our work, the verification engine was designed as an internal Rhapsody routine to allow property checking of Action-LTL assertions. Success/failures are shown as new events on the generated sequence diagram. This facilitates debugging. The verification engine was built on top of Rhapsody. This posed several non-trivial challenges, like creation and integration of assertion monitors, and required us to engineer the internals of Rhapsody to implement our DPV algorithm.

The main contributions of this paper are as follows:

- We have proposed a LTL-based simulation verification approach for UML, using Rhapsody semantics.

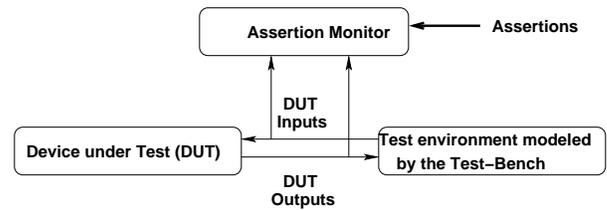


Figure 2: DPV Platform

- We have extended LTL to be able to express requirements over data attributes and events in UML. The extended logic, called Action-LTL has a very similar syntactic structure as LTL. The novelty of the logic lies in the fact that its atomic predicates connect to entities in UML models and the semantics is defined over the Rhapsody semantics of UML.
- We have developed a prototype implementation of the proposed verification platform over Rhapsody.

This paper is organized as follows. Section 2 discusses some background concepts that have been used in this work. Section 3 presents Action-LTL, while Section 4 presents the concept of DPV for Action-LTL. Section 5 presents the implementation of the DPV in the context of Rhapsody while Section 6 presents some experimental results. Section 7 presents related work in this area.

2. BACKGROUND THEORY

In this section, we present some background concepts that form the foundations of this work.

2.1 Dynamic Property Verification

Figure 2 shows the DPV setup. The DUT represents the design description to be verified. The test-bench models the test environment which has the responsibility of driving different test inputs to the DUT and reading the values of the DUT outputs. The assertions represent the correctness properties that the user wishes to verify on the DUT. At each simulation step, the test bench provides a valuation for the DUT inputs, to which the DUT reacts and produces a valuation of the outputs. The assertion monitor reads the input and output valuation produced at each step and evaluates the assertions to determine which ones are true or false.

There are two key features of DPV that explain its growing usage in the verification projects. Firstly, it is built over the traditional simulation framework and requires minimal additional effort from the verification engineer. Secondly, it does not have any major capacity limitations, since the verification is done over the simulation run.

2.1.1 DPV Basics

DPV is a simulation-based approach, in which a set of properties expressing the desired behavior of the system under test are checked against a simulation run. As in simulation, we write a test-bench to drive inputs into the implementation. At the heart of DPV is the concept of monitoring requirements over simulation runs. Requirements are expressed as assertions in some temporal-logic formalism. These assertions are verified broadly in either of the following ways:

1. Each assertion is automatically translated to a monitor state machine with states for acceptance and rejection corresponding to success or failure of the assertion. These assertion

monitors are then cosimulated with the design under test. Success or violations are recorded depending on whether the monitor enters an accepting or rejecting state during simulation. In case the monitor remains at an intermittent state, the success or failure cannot be decided during the course of the simulation run. The translation rules and monitoring methodology are standard [16].

2. Assertions are monitored on-the-fly during simulation using the concept of dynamic property unfolding over simulation. The unfolding rules are standard and quite widely used in the formal verification methods for bounded model checking [16]. In each simulation step, each assertion is evaluated based on the values of the system variables generated in that simulation step. If any assertion evaluates to true or false, it is reported to the user. For assertions that cannot be fully evaluated to success or failure within one cycle, the verification engine prepares the property to be checked during the next simulation cycle and returns control to the simulator. This process continues till all assertions have been evaluated to be true or false or simulation is terminated.

In this work, we adopt the second approach. The translation rules and the overall methodology will be presented in Section 4. The main advantage of this approach over the earlier one is scalability. In the earlier approach, the size of the assertion monitor grows with complexity of the assertion and thereby increases the simulation overhead. In the second approach, the formula size might blow up as well. The main difference between these two approaches lies in the fact that the latter approach expands the formula as and when the need arises, whereas the first approach does the translation independent of the model and hence would always incur the blowup.

The success of DPV depends largely on the ability of the simulation test generator to create good testcases for validating the requirements. Classical approaches to writing test cases are guided by the test engineer's intuition and experience in designing *truly effective* test cases which have a high likelihood in exposing errors. Ideally, the set of testcases should cover all functionality, address boundary conditions, and cover all scenarios. The main criticism of this approach is that the fraction of the design space which can be explored by simulation is miniscule, especially for large designs. Only one state and one input combination of the design under test are visited during each simulation cycle. Moreover the test stimuli are hand crafted by the designer to cover those areas of the design that he wishes to validate. For a large, complex system, it is impossible to test or simulate all possible inputs or sequences of inputs. Furthermore, simulation inputs are usually based on the design specification and are thus only aimed at verifying that the design performs all the primary activities indicated in the specification document. However, it is often the case that complex systems manifest unforeseen behavior for corner case situations. Most often, designers are unaware of behavior that results as a by-product of the interactions among different modules and that was unaccounted for in the specification document. These cases do not get checked, while they may have negative consequences on the overall behavior of the system. DPV is not effective unless we can force the test bench to create the relevant scenarios that can cover the corner cases for which some assertions are written.

2.2 UML Statecharts

UML is a semi-formal language. Though its syntax and semantics (the model elements, their interconnection and well-formedness) are formally defined, the dynamic semantics are specified only informally. This makes the development of a validation framework

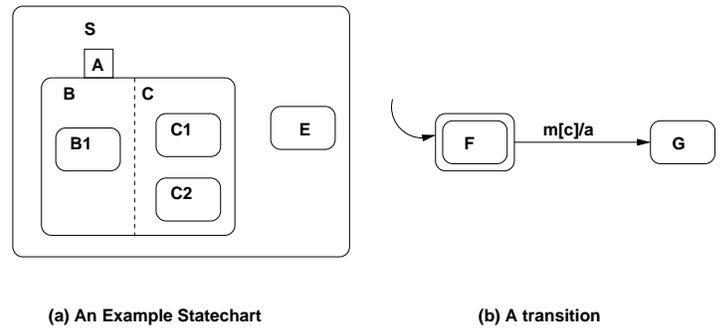


Figure 3: An Example Statechart

more difficult. UML design tools with simulation support typically define their own dynamic semantics. Our validation framework is based on Rhapsody from I-Logix [20]. [20] presents a detailed discussion of the Rhapsody semantics of Statecharts.

UML has a variety of features to facilitate system description. For the purpose of this work, we have used Statecharts as the mode of expressing designs. A design consists of a collection of Statecharts corresponding to the behavior of the components of the subsystems it comprises. A Statechart describes the modal behavior of a subsystem description.

A Statechart consists of 3 types of states, **OR-states**, **AND-states** and **basic states**. The OR states have sub-states related to each other by *exclusive or*, AND-states have orthogonal components that are related by *and*, while basic states have no sub-states, and are the lowest in the state hierarchy. Figure 3(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. If a system is in the OR superstate, it means that its in any of the component OR states. On the contrary, if a system is in a AND superstate, it implies that the system is in all the component states simultaneously. When building a Statechart in Rhapsody, an additional state is created implicitly, called the root state (the highest in the state hierarchy). In the example, the root state has S as a sub-state.

The general syntax of a transition label in a Statechart is $m[c]/a$ (all optional) where m is the message that triggers the transition, c is a condition that prevents 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. Figure 3(b) shows an example transition.

2.3 Rhapsody Semantics

In Rhapsody, events form the central component for communication among Statecharts for different concurrent modules of a system. Events are used to describe asynchronous communication. Each module defines the set of events it can receive. The main motivation for using asynchronous events is that the sender object can continue its work without waiting for the receiver to consume the event. For the purpose of our work, we consider only single-threaded applications and, hence, our verification methodology assumes a single simulation-event queue (FIFO) for the entire system. This ensures that we have a fixed interleaving of events, and events are dispatched in the strict sequential order in which they are generated. Events are sent by applying the GEN method to the destination object: $O \rightarrow GEN(event)$. The sending object should be able to refer to the destination object

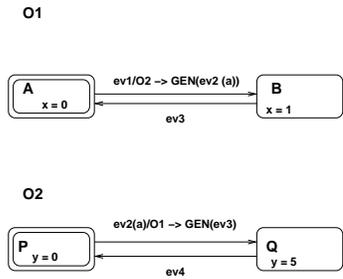


Figure 4: A Sample System

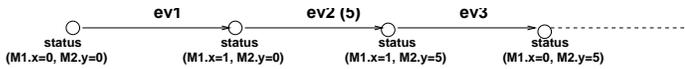


Figure 5: The Rhapsody Run

O. Rhapsody supports the expression of events with parameters. For the purpose of this work, *we restrict ourselves to events without parameters*. The GEN method creates the event instance and queues it in the queue of an object O's thread.

Events in Rhapsody are managed via queues (FIFO). 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. The following example illustrates the simulation of a simple system in Rhapsody. In the subsequent discussion, we will formalize the concepts.

EXAMPLE 1. Consider the system in Figure 4 consisting of objects O1 and O2. x is a data member of O1, y is a data member of O2. $ev1$ is an external event which is at the top of the event queue at the start of simulation. $ev2$, $ev3$, $ev4$ are system events. State A is the initial state of O1 and State P is the initial state of O2. Hence, the initial state of the system consists of (O1.A, O2.P). Consider the following execution of the system: O1 receives event $ev1$ from the user. The transition from state A to state B is fired. This transition involves sending event $ev2$ to object O2 by placing a new event $ev2$ in the event queue as specified by the action $O2 \rightarrow GEN(ev2)$. Once the transition to state B of O1 is completed and the system is stabilized, the state of the system is (O1.B, O2.P). In the next step, event $ev2$ is removed from the event queue, and is dispatched to O2, causing it to take the transition from its state P to state Q. A sample snapshot of the Rhapsody execution trace of the system is shown in Figure 5. \square

In the following subsections, we formalize some concepts underlying simulation in Rhapsody.

2.3.1 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 sequence of detailed *snapshots* of the system's state. 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 6) triggered by the dispatch of an event. Each step is composed of *microsteps* as shown in Fig 6. Microsteps typically capture a set of internal reactions of a system that occur before the system reaches the next status from the current status. The system, being in a certain status and as a response to an event,

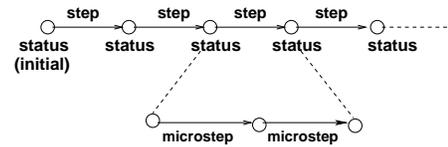


Figure 6: Basic Reaction

undergoes a sequence of microsteps, until it reaches another status, and at which point it is ready for the next event dispatch. *It may be noted that only a single event can be dispatched on a step between two status points since we assume a single blocking event queue in the system*. If the guard condition of an event on top of the queue is not satisfied, it is discarded and not dispatched. Also, the next status corresponding to an event dispatch may be same as the present status if the guard condition is not satisfied causing the system to remain in the same status. [20] presents a detailed discussion on steps and microsteps.

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.

2.3.2 The Rhapsody Simulation-Step Algorithm

In this section we present a schematic description of the algorithm that executes a step. This is of utmost importance to our work, since our DPV mechanism is built in accordance to this semantics. In this discussion, we briefly present the concepts relevant to our work. The work [20] presents a detailed discussion of the Rhapsody simulation semantics.

At each Rhapsody simulation step, if the event queue is non-empty, the first event is taken and dispatched to its destination. For a dispatched event at a particular system status, a step (consisting of a sequence of microsteps) is executed and the system is transferred to a new configuration before the next event is processed. 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). Below, we present a formal definition of a status and a Rhapsody run.

DEFINITION 1 (**(: Status :)**). A status η contains information about the state of a system at a particular instant. Given a system with a set of objects O (each having its own set of data variables) and events, a status contains the following: (a) values of the different data variables at that instant and (b) occurrence / non-occurrence information of each event in the system queue at that instant. \square

DEFINITION 2 (**(: Run :)**). A run π is described as a sequence $\langle \eta_0, ev_0, \eta_1, ev_1, \dots \rangle$, where η_i corresponds to a Rhapsody status (η_0 being the initial status), and each ev_i is an event occurring on the transition between status η_i and η_{i+1} with some object as the target. \square

We denote by π^i , the suffix of a run π starting from the status η_i . Action-LTL properties are expressed over data values at the status points and over events that are dispatched at each step (between two status points).

3. ACTION-LTL

Action-LTL is a simple extension of LTL, and hence enjoys similar syntax and semantics. First, we describe the language elements over which Action-LTL properties are written.

Given a system consisting of objects O_1, O_2, \dots, O_k , each having its own set of data variables D_i , the syntax of Action-LTL is defined over the set of data variables $D = \bigcup D_i$ and the set of events Σ .

The instantiation of a single event at a simulation step is interpreted as a Boolean value indicating that the event is true at that particular step. In case the event is absent, we interpret it as false.

We define the concept of a *data predicate*.

DEFINITION 3. A **data predicate** is an expression of the form $\langle lexp \rangle \bowtie \langle rexp \rangle$ where $\bowtie \in \{\leq, \geq, =, \neq, <, >\}$, $\langle lexp \rangle$ and $\langle rexp \rangle$ are relational expressions involving arithmetic and logical operators over data variables and numeric constants. \square

Given a valuation to the data variables at a particular status on a Rhapsody run, a data predicate evaluates to **true** or **false**.

EXAMPLE 2. Consider the system in Figure 4. Example data predicates over this system are : $O1.x = 0$, $O2.y \leq O1.x + 2$, $O2.y \neq O1.x - 6$, $O1.x = 5$. \square

In the following discussion, $O.p$ refers to a Boolean data variable p of object O , $O.ev$ is an event dispatched to O , d refers to a data predicate. It may be noted that from the interpretation perspective, an event $O.ev$ can be considered as similar to a Boolean data variable having the value true or false depending on its occurrence or non-occurrence respectively. However, there are subtle differences as well, namely (a) the scope of each event has a target object which may vary across simulation steps while each data variable is bound to the same object of which it is a member, and (b) the access and dispatch mechanism for events is different from that of data variables. We will illustrate these issues later in Section 5.

The syntax of Action-LTL is as follows:

- $O.p$ is an Action-LTL formula
- $O.ev$ is an Action-LTL formula
- Each data predicate d is an Action-LTL formula
- If ϕ and ψ are Action-LTL formulas, then so are $\neg\phi$, $\phi \vee \psi$, $X\phi$, $\phi U\psi$.

As usual, we have the abbreviations $\phi \wedge \psi$ for $\neg(\neg\phi \vee \neg\psi)$, *true* for $\phi \vee \neg\phi$, *false* for $\neg true$, $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$, $F\phi$ for $true U\phi$ and $G\phi$ for $\neg F\neg\phi$.

The semantics of Action-LTL is defined with respect to an infinite run $\pi = \langle \eta_0, ev_0, \eta_1, ev_1, \dots \rangle$ of Rhapsody, where η_i corresponds to a Rhapsody status (η_0 being the initial status), ev_i are events, **f** and **g** are arbitrary Action-LTL formulae. Before stating the

semantics of Action-LTL, we first present two conditions that help us determine the truth of a Boolean formula or a data predicate at a state η_k on the run. These are as follows:

- $\eta_k \models O.p$ iff the Boolean member p of object O is true in the status η_k ,
- $\eta_k \models d$ iff the data predicate d is true in the status η_k ,

It may be noted that in conformance to the semantics of LTL, integer type data members of objects cannot be used in an Action-LTL formula as $O.p$ like their Boolean counterparts. Instead, these can be used in Action-LTL data predicates to express conditions that have a Boolean evaluation result.

An Action-LTL formula ϕ holds at a point $i \geq 0$ on an infinite run π , denoted by $\pi^i \models \phi$ if the following hold:

- $\pi^i \models O.p$ iff $\eta_i \models O.p$
- $\pi^i \models d$ iff $\eta_i \models d$
- $\pi^i \models O.ev$ iff $i \geq 1$ and $ev_{i-1} = ev$ and there is a dispatch of event ev to object O between status η_{i-1} and η_i
- $\pi^i \models \neg f$ iff $\pi^i \not\models f$
- $\pi^i \models f \vee g$ iff $\pi^i \models f$ or $\pi^i \models g$
- $\pi^i \models X f$ iff $\pi^{i+1} \models f$
- $\pi^i \models f U g$ iff $\exists k \geq i$ such that $\pi^k \models g$, and $\forall j, i \leq j \leq k-1, \pi^j \models f$.

An infinite run π satisfies an Action-LTL formula ϕ , denoted $\pi \models \phi$, if $\pi^0 \models \phi$. In the remainder of this paper, we assume this semantics. The semantics of Action-LTL is explained through the following example.

EXAMPLE 3. Consider the system in Figure 4. Below, we present some correctness requirements for this system and the corresponding Action-LTL encodings.

- $O1.x = 0$: The property formally expresses the requirement that O1 has $x=0$ in the initial status. In this case, $O1.x = 0$ is a data predicate that evaluates to true if the data variable x has the value 0. This property evaluates to true in our example model.
- $O2.ev1$: The property formally expresses the requirement that the first event that is dispatched is an **ev1** event that is dispatched to $O2$. This property also evaluates to true on our example model.
- $F((O1.x = 0) \wedge (O2.y = 6))$: This formally expresses the requirement that in some future status, the value of the data variables x and y are 0 and 6 respectively.

As the above properties demonstrate, properties involving state variables and events may be expressed using our language. \square

4. DPV WITH ACTION-LTL

In this section, we give the overall verification procedure. For validating a system S consisting of a set of UML objects, each having a Statechart capturing its behavior, the overall approach is:

- Encode the assertions in Action-LTL and develop the procedure for assertion unfolding and monitoring.
- Define the interface to read data values.
- Overload the event dispatcher (explained later) to ensure that at every step, a copy of every dispatched event is also delivered instantaneously to the assertion monitor.
- Run Rhapsody simulation and verify the truth of each assertion on the generated system trace.

The first three steps above are generic, and need to be executed only once at the start of the verification stage, for verifying multiple assertions using the fourth step.

The user-defined Action-LTL specification defines the initial assertion monitor for simulation. The initial values of the design-under-test are substituted and we check whether a success/failure of the specification can be concluded. If so, appropriate message is logged. Otherwise, the Rhapsody simulation step algorithm takes over. This monitoring of the specification is performed at each step to check for assertion violations. This is implemented as an on-the-fly assertion unfolding routine and added as a Statechart inside the design. This routine performs the following action steps:

1. Create a one-step unfolding of the Action-LTL specification to be verified
2. Substitute the values corresponding to the current active configuration (or the initial active configuration) in the unfolded specification and check whether the specification evaluates to true or false. If so, log the appropriate status and return, otherwise proceed with the top event from the event queue as earlier.

To address the first point above, we need to formalize the unfolding rules for an Action-LTL formula that create an instance of the formula to be checked such that the second step can be applied.

For the second point, our goal is to decide on satisfaction / violation of the Action-LTL formula to be checked on the basis of the simulation values obtained by finite-length simulation till the current time point where Step 2 is being executed. This may be possible if the formula under consideration is purely Boolean. However, this conclusion task is not straightforward for formulas consisting of temporal operators and we need to define a finite-trace semantics of satisfaction for LTL operators. For example, the satisfaction of a formula containing the *next* (X) operator depends on the number of cycles for which simulation is run and the temporal depth of the *next* operator. An Action-LTL property $X p$ can only be concluded to be true / false using a minimum of 2 simulation cycles, since this property refers to a requirement at the third time step (assuming the first step is the initial configuration). For formulas containing the *always* (G) operator, it is only possible to conclude a refutation status on the basis of a finite-length trace, and satisfactions cannot be concluded. For formulas

containing the *future* (F) operator, it is only possible to conclude a satisfaction and not a refutation on the basis of a finite-length simulation run. This necessitates a finite-trace semantics of LTL, which we define and use for the purpose of this work.

4.1 Finite Trace Semantics of Action-LTL

The semantics of Action-LTL presented in Sec 3 is defined with respect to an infinite run of the system. To verify the truth of Action-LTL assertions over simulation, we need to formalize the finite-trace semantics of Action-LTL since simulation will be performed for a finite number of steps.

Let $\pi = \langle \eta_0, ev_0, \eta_1, ev_1, \dots, \eta_{k-1}, ev_{k-1}, \eta_k \rangle$ be a finite simulation run obtained by simulating the system for k steps through Rhapsody. As before, η_i corresponds to a Rhapsody status (η_0 being the initial status and η_k being the final status), and ev_i are events. An Action-LTL formula ϕ holds at a point $i \geq 0$ on a finite run π , denoted $\pi^i \models \phi$ if the infinite-trace semantics together with the following modification holds:

- $\pi^i \models X f$ iff $i < k - 1$ and $\pi^{i+1} \models f$
- $\pi^i \models f U g$ iff $\exists m, i \leq m < k$ such that $\pi^m \models g$, and $\forall j, i \leq j < m - 1, \pi^j \models f$.

4.2 Monitoring Action-LTL properties over a finite simulation run

Before defining the methodology of monitoring an Action-LTL property over a finite simulation run, we present a few useful concepts. In the discussion below, we assume g and h are arbitrary Action-LTL formulae.

DEFINITION 4 (Elementary Subformula :). *The set $EL(f)$ of elementary subformulas of an Action-LTL formula f is recursively defined as follows:*

- $EL(p) = \{p\}$ if $p \in \Sigma \cup D$
- $EL(\neg g) = EL(g)$
- $EL(g \text{ op } h) = EL(g) \cup EL(h)$ where *op* stands for Boolean binary operators used in Action-LTL formulae or relational operators used in data predicates.
- $EL(Xg) = \{Xg\} \cup EL(g)$
- $EL(Fg) = \{X Fg\} \cup EL(g)$
- $EL(Gg) = \{X Gg\} \cup EL(g)$
- $EL(g U h) = \{X(g U h)\} \cup EL(g) \cup EL(h)$

The concept of elementary subformula is standard and used extensively for tableau-based LTL model checking [13].

DEFINITION 5 (One-Step unfolding :). *The one-step unfolding of an Action-LTL formula φ , denoted as $\Gamma(\varphi)$ is defined as follows:*

- $\Gamma(p) = p$ if p is a data member ($O.p$) or an event ($O.ev$)
- $\Gamma(\neg g) = \neg \Gamma(g)$

- $\Gamma(g \text{ op } h) = \Gamma(g) \text{ op } \Gamma(h)$ where *op* stands for Boolean binary operators used in Action-LTL formulae or relational operators used in data predicates.
- $\Gamma(Xg) = X \Gamma(g)$
- $\Gamma(g \text{ U } h) = \Gamma(h) \vee [\Gamma(g) \wedge X(g \text{ U } h)]$
- $\Gamma(Fg) = \Gamma(g) \vee [X Fg]$
- $\Gamma(Gg) = \Gamma(g) \wedge [X Gg]$

The unfolding rules for LTL are standard [16]. The unfolding rules applied to φ yields a temporal formula in terms of the elementary subformulae $EL(\varphi)$ of φ . An elementary subformula, as defined above, is a constant, an atomic proposition (data member or event), or a formula starting with **X**. The expanded formula, put in disjunctive normal form (DNF) is an elementary cover of φ . The work [39] presents a detailed discussion on this. Each term of the cover identifies a condition that can contribute to the satisfaction of φ . In our case, the atomic propositions and their negation define the condition to be satisfied at a particular time instant t . The remaining elementary subformulae form the *next* part of the term; they are Action-LTL formulae that identify the obligations that must be fulfilled to obtain an accepting condition of φ at $t + 1$.

Given a system consisting of objects O_1, O_2, \dots, O_k , each having its own set of data variables D_i , we now define the *valuation* of an Action-LTL formula defined over the set of data variables $D = \bigcup D_i$ and the set of events Σ with respect to an assignment of values of elements in $D \cup \Sigma$.

DEFINITION 6 (Assignment :). An **Assignment** Θ over $D \cup \Sigma$ is a valuation that assigns to each member $x \in D \cup \Sigma$ a value from its respective domain. \square

Consider the system shown in Figure 4. An assignment Θ over $\{x, y, ev1, ev2, ev3\}$ is $\{0, 2, true, false, true\}$ that assigns a value to each data member and event from the respective domain.

An **assignment** Θ over $D \cup \Sigma$ at a status point η_i ($i \geq 1$) on a run $\pi = \langle \eta_0, ev_0, \eta_1, ev_1, \dots \rangle$ is constructed by including the values of all members in D at η_i and **true** for the event ev_{i-1} dispatched between status η_{i-1} and η_i and **false** for all events x , $x \in \{\Sigma - ev_{i-1}\}$. For the initial status η_0 , we consider Θ to contain only the values of the members in D since there are no events till that point. Consider the second time point on the run shown in Figure 5. The assignment Θ over $\{x, y, ev1, ev2, ev3\}$ is $\{1, 0, true, false, false\}$.

DEFINITION 7 (X-prefixed formula:). An Action-LTL formula φ is **X-prefixed** if the leftmost term (discarding preceding parentheses) in φ is the **X** operator, where the scope of this **X** extends entirely over the remaining terms of φ . \square

For example, $X(g)$, $X[f \text{ U } g]$, $X(p)$, $X(a \vee b)$, $X(\neg g)$ are all X-prefixed since the leftmost term in each of these is the **X** operator, where the scope of this **X** extends entirely over φ . $(X \varphi)$ is also considered X-prefixed as per Definition 7. However, $X(a) \vee X(b)$ is not X-prefixed since the scope of the leftmost **X** does not extend over the entire formula.

DEFINITION 8 (Remove-X :). The **Remove-X** operation on an X-prefixed Action-LTL formula removes the leftmost **X** from φ . \square

The **Remove-X** operation on $X(f \text{ U } g)$ produces $(f \text{ U } g)$.

DEFINITION 9 (Valuation :). The **Valuation** $\mathcal{V}(\varphi, \Theta)$ of an Action-LTL formula φ , with respect to an assignment Θ is obtained by substituting the values of data members and events specified in Θ in the one-step unfolded form $\Gamma(\varphi)$. \mathcal{V} can be one of the following :

- true**
- false**
- An Action-LTL formula containing only X-prefixed terms in the elementary subformulae of φ . \square

The structure of $\mathcal{V}(\varphi, \Theta)$ follows intuitively from the definition of the one-step unfolded form of φ . Consider the following example:

EXAMPLE 4. Consider the Action-LTL property $\varphi : F(x = 0)$ defined over $\{x, y, ev1, ev2, ev3\}$ for the system shown in Figure 4. Consider the valuation $\Theta : \{0, 2, true, false, true\}$. The one-step unfolded form of φ , namely, $\Gamma(\varphi)$ is : $(x = 0) \vee X(F(x = 0))$. The data predicate $(x = 0)$ has the value **true**, since x has the value 0 in Θ . This gives us the following form for $\Gamma(\varphi)$: $true \vee X(F(x = 0)) = true$. The valuation \mathcal{V} of φ , therefore, has the value **true**. Similarly, for the Action-LTL property $\varphi : G(x = 0)$, the one-step unfolded form, $\Gamma(\varphi)$ is : $(x = 0) \wedge X(\varphi)$. For the same valuation Θ , we have the following form for $\Gamma(\varphi)$: $true \wedge X(\varphi) = X(\varphi) = XG(x = 0)$. \square

Lemma 1 establishes the correspondence between an Action-LTL formula and its valuation on a finite run

$\pi = \langle \eta_0, ev_0, \eta_1, ev_1, \dots, \eta_{k-1}, ev_{k-1}, \eta_k \rangle$.

LEMMA 1. An Action-LTL formula φ holds at a point $i \geq 0$ on a finite run π , denoted by $\pi^i \models \varphi$ iff at least one of the conditions below hold on $\mathcal{V}(\varphi, \Theta)$, where Θ denotes the assignment at s_i :

- Valuation $\mathcal{V}(\varphi, \Theta) = true$,
- $\mathcal{V}(\varphi, \Theta)$ consists only of X-prefixed terms in the elementary subformula of φ and $\pi^{i+1} \models \phi$ where ϕ is obtained after the **Remove - X** operation on each X-prefixed elementary subformula of φ present in $\mathcal{V}(\varphi, \Theta)$.

Proof : The first condition is obvious. For the second condition, we note that Valuation $\mathcal{V}(\varphi, \Theta)$ consists only of X-prefixed terms in the elementary subformula of φ . The Valuation operation operates on $\Gamma(\varphi)$ and each term in $\Gamma(\varphi)$ put in disjunctive normal form constitutes an elementary cover of φ . Each term of the cover identifies a condition that can contribute to the satisfaction of φ . $\mathcal{V}(\varphi, \Theta)$ comprises only of X-prefixed terms, and therefore, constitutes the *next* part of the term which are the obligations that must be fulfilled to obtain an accepting condition of φ at $i + 1$, where each term is X-prefixed. The *Remove - X* operation on $\mathcal{V}(\varphi, \Theta)$ produces the obligation to be satisfied from $i + 1$. Hence, we can conclude $\pi^i \models \varphi$ if $\pi^{i+1} \models \phi$, where ϕ is obtained after

the **Remove** – **X** operation on each X -prefixed elementary subformula of φ present in $\mathcal{V}(\varphi, \Theta)$. The other direction of the proof is similar. \square

The task of monitoring the truth of a given Action-LTL property along a finite simulation run is intuitively simple. If we are required to check a property, φ , from a given time step, t , we have the following actions :

1. We create the one-step unfolded form $\Gamma(\varphi)$ for φ .
2. The assertion monitor reads the valuation Θ of data members and events from the Rhapsody status at time t .
3. The *Valuation* $\mathcal{V}(\varphi, \Theta)$ is computed.
4. If $\mathcal{V}(\varphi, \Theta)$ evaluates to **true** or **false**, we stop, else we continue the same steps as above on the Action-LTL formula $\hat{\varphi}$ obtained after the **Remove** – **X** operation on each X -prefixed elementary subformula of φ present in $\mathcal{V}(\varphi, \Theta)$. $\hat{\varphi}$ constitutes the formula to be checked from time $t + 1$.

The above steps are continued till we reach a success or failure in Step 4 or till the end of simulation where the result of the property under consideration remains inconclusive. We report such properties as *false*.

EXAMPLE 5. Consider the property:

$$\psi = O1.p \cup (O2.q \cup M3.r)$$

at time t , where p, q, r are all Boolean data variables. We rewrite ψ as :

$$\psi = (M3.r \vee (O2.q \wedge X(O2.q \cup M3.r))) \vee (O1.p \wedge X(O1.p \cup (O2.q \cup M3.r)))$$

If the simulation at t gives $p = 0, q = 1, r = 0$, by substituting these values, we obtain the property $X(O2.q \cup M3.r)$. After the **Remove-X** operation, we obtain the Action-LTL formula $O2.q \cup M3.r$ which we need to check from time $t + 1$. We repeat the same methodology on $O2.q \cup M3.r$ at time $t + 1$. \square

4.3 The DPV Algorithm

Algorithm 4.1 presents the DPV procedure for an Action-LTL specification.

ALGO. 4.1. Procedure DPV

DPV(Action-LTL: \mathcal{L})

Step 1: Set \hat{S} = assignment at the initial system status η_0

Step 2: While (not end of simulation) do

- 2.1: Create the one-step unfolded form $\Gamma(\mathcal{L})$
- 2.2: Compute $\hat{L} = \mathcal{V}(\mathcal{L}, \hat{S})$ using $\Gamma(\mathcal{L})$
- 2.3: If \hat{L} evaluates to FALSE, flag failure; exit
- 2.4: If \hat{L} evaluates to TRUE, flag success; exit
- 2.5: Set \mathcal{P} = the Action-LTL formula obtained by the *Remove-X* operation on each X -prefixed elementary subformula of \mathcal{L} present in \hat{L}

2.6: Set $\mathcal{L} = \mathcal{P}$

2.7: Set \hat{S} = the assignment at the next status obtained after one step simulation

2.8: Go to Step 2

EndAlgorithm

The above routine is invoked at every step of Rhapsody simulation. At any step, if a violation / success is detected, simulation is halted with appropriate message to the designer so that he can intervene. If we reach the end of simulation without a conclusive *true* / *false* result for the property under consideration, we conclude that the status of the property is *false* till the number of simulation cycles run in accordance to the finite-trace semantics.

The correctness of Algorithm 4.1 follows the discussion presented in Section 4.2 and Lemma 1. The following example illustrates the concept of unfolding and dynamic monitoring in greater detail.

EXAMPLE 6. Consider the specification

$$\mathcal{L} = G(O2.ev2 \Rightarrow X(O2.y=4))$$

on the system shown in Example 1. Let us assume we are simulating for 5 iterations. The initial assignment \hat{S} for $\{x, y, ev1, ev2, ev3\}$ is $\{0, 0, true, false, false\}$ since $O1.x = 0, O2.y = 0$, and $ev1$ is the top event in the event queue. The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$\begin{aligned} & \Gamma(\neg O2.ev2 \vee X(O2.y=4)) \wedge (X \mathcal{L}) \\ &= [\Gamma(\neg O2.ev2) \vee \Gamma(X(O2.y=4))] \wedge (X \mathcal{L}) \\ &= [\neg O2.ev2 \vee X \Gamma(O2.y=4)] \wedge (X \mathcal{L}) \\ &= [\neg O2.ev2 \vee X(O2.y=4)] \wedge (X \mathcal{L}) \\ &= (O2.ev2 \Rightarrow X(O2.y=4)) \wedge (X \mathcal{L}) \end{aligned}$$

The valuation $\mathcal{V}(L, \hat{S})$ produces \hat{L} as

$$\begin{aligned} \hat{L} &= (false \Rightarrow X(O2.y=4)) \wedge (X \mathcal{L}) \\ &= true \wedge (X \mathcal{L}) \\ &= (X \mathcal{L}) \\ &= X G(O2.ev2 \Rightarrow X(O2.y=4)) \end{aligned}$$

\hat{L} evaluates to neither true nor false. Step 2.5 on \hat{L} produces $G(O2.ev2 \Rightarrow X(O2.y=4))$. On consumption of the event $ev1$, the model (as shown by the status in Fig 5) changes its active configuration. At this step, $O1.x=1, O2.y=0$ and $ev2$ is the top event in the event queue. Hence, $\hat{S} = \{1, 0, false, true, false\}$. At the start of the next step,

$$\mathcal{L} = G(O2.ev2 \Rightarrow X(O2.y=4))$$

The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$(O2.ev2 \Rightarrow X(O2.y=4)) \wedge (X \mathcal{L})$$

The valuation $\mathcal{V}(\mathcal{L}, \hat{S})$ produces \hat{L} as

$$\begin{aligned} \hat{L} &= (true \Rightarrow X(O2.y=4)) \wedge (X \mathcal{L}) \\ &= X(O2.y=4) \wedge (X \mathcal{L}) \\ &= X(O2.y=4) \wedge X G(O2.ev2 \Rightarrow X(O2.y=4)) \end{aligned}$$

\widehat{L} evaluates to neither true nor false. Step 2.5 on \widehat{L} produces $(O2.y=4) \wedge G(O2.ev2 \Rightarrow X(O2.y=4))$. The event $ev2$ is dispatched and the next active configuration is recorded. \widehat{S} now is $\{1, 5, false, false, true\}$ At the start of the next step,

$$\mathcal{L} = (O2.y=4) \wedge G(O2.ev2 \Rightarrow X(O2.y=4))$$

The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$(O2.y=4) \wedge (O2.ev2 \Rightarrow X(O2.y=4)) \wedge (X\mathcal{L})$$

Since $O2.y$ has the value 5, $O2.y=4$ evaluates to false. The valuation $\mathcal{V}(\mathcal{L}, \widehat{S})$ produces \widehat{L} as

$$\widehat{L} = false \wedge (O2.ev2 \Rightarrow X(O2.y=4)) \wedge (X\mathcal{L}) = false$$

Therefore, we have a failure for \mathcal{L} . \square

EXAMPLE 7. We now consider another specification to illustrate the working of our DPV algorithm 4.1. Let us assume we are simulating for 5 iterations as in Example 6.

$$\mathcal{L} = F(O1.x=1 \cup O2.y=5)$$

For $\{x, y, ev1, ev2, ev3\}$, we have $\{0, 0, true, false, false\}$ as the initial assignment, since $O1.x = 0$, $O2.y = 0$, and $ev1$ is the top event in the event queue. The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$\begin{aligned} \Gamma(O1.x=1 \cup O2.y=5) \vee X\mathcal{L} \\ = [O2.y=5 \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee X\mathcal{L} \end{aligned}$$

Since $O1.x=0$ and $O2.y=0$, the valuation $\mathcal{V}(\mathcal{L}, \widehat{S})$ produces \widehat{L} as

$$\begin{aligned} \widehat{L} &= [false \vee (false \wedge X(O1.x=1 \cup O2.y=5))] \vee X\mathcal{L} \\ &= (X\mathcal{L}) \\ &= X F(O1.x=1 \cup O2.y=5) \end{aligned}$$

\widehat{L} evaluates to neither true nor false. Step 2.5 on \widehat{L} produces $F(O1.x=1 \cup O2.y=5)$. On consumption of the event $ev1$, the model (as shown by the status in Fig 5) changes its active configuration. At this step, $O1.x=1$, $O2.y=0$ and $ev2$ is the top event in the event queue. Hence, $\widehat{S} = \{1, 0, false, true, false\}$. At the start of the next step,

$$\mathcal{L} = F(O1.x=1 \cup O2.y=5)$$

The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$[O2.y=5 \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee X\mathcal{L}$$

Since $O1.x=1$ and $O2.y=0$, the valuation $\mathcal{V}(\mathcal{L}, \widehat{S})$ produces \widehat{L} as

$$\begin{aligned} \widehat{L} &= [false \vee (true \wedge X(O1.x=1 \cup O2.y=5))] \vee X\mathcal{L} \\ &= X(O1.x=1 \cup O2.y=5) \vee X\mathcal{L} \\ &= X(O1.x=1 \cup O2.y=5) \vee X F(O1.x=1 \cup O2.y=5) \end{aligned}$$

\widehat{L} evaluates to neither true nor false. Step 2.5 on \widehat{L} produces $(O1.x=1 \cup O2.y=5) \vee F(O1.x=1 \cup O2.y=5)$. The event $ev2$ is dispatched and the next active configuration is recorded. \widehat{S} now is $\{1, 5, false, false, true\}$ At the start of the next step,

$$\mathcal{L} = (O1.x=1 \cup O2.y=5) \vee F(O1.x=1 \cup O2.y=5)$$

The one-step unfolded form, $\Gamma(\mathcal{L})$ is :

$$\begin{aligned} \Gamma(O1.x=1 \cup O2.y=5) \vee \Gamma(F(O1.x=1 \cup O2.y=5)) \\ = [O2.y=5 \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee \\ [O2.y=5 \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee \\ X F(O1.x=1 \cup O2.y=5) \end{aligned}$$

Since $O2.y$ has the value 5, $O2.y=5$ evaluates to true. The valuation $\mathcal{V}(\mathcal{L}, \widehat{S})$ produces \widehat{L} as

$$\begin{aligned} \widehat{L} &= [true \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee \\ & [O2.y=5 \vee (O1.x=1 \wedge X(O1.x=1 \cup O2.y=5))] \vee \\ & X F(O1.x=1 \cup O2.y=5) \\ &= true \end{aligned}$$

Therefore, we have a success for \mathcal{L} . \square

5. IMPLEMENTATION DETAILS

Assertions in Action-LTL are written over the model variables, specifically, the data members and events. Hence, to determine satisfaction/refutation of an assertion at a simulation step due to the model behavior, it is necessary to be able to read values of the model variables. Below, we briefly explain some issues faced by us in developing the DPV platform over Rhapsody and subsequently, the approach we adopted. The issues are as follows:

1. With the interception of every event, the assertion monitor must check if the model configuration satisfies/refutes the desired specification. The assertion monitor must remember the context of verification; it must remember what has to be checked after dispatch of each event to certify correctness of the model at that point of time.
2. The assertion monitor must be able to sense the values of data variables and all events dispatched in the model at every status. An interface is necessary through which model-data variables can be read by the assertion monitor.
3. The UML model must be minimally perturbed in course of building the assertion monitors. Last but not the least, the assertion monitors must be seamlessly useable into an already existing design environment over Rhapsody.

We now briefly discuss the strategy adopted by us in addressing the above issues.

5.1 Assertion Monitor Architecture

The assertion monitor has been implemented as a Statechart with an embedded C routine. At every simulation step, on notification of an event dispatch, the assertion monitor reads the data variable values and checks for satisfaction/refutations. The C routine, based on the one step unfolding of the temporal properties as discussed in the previous section, has been developed and a control logic has been designed that calls that routine appropriately.

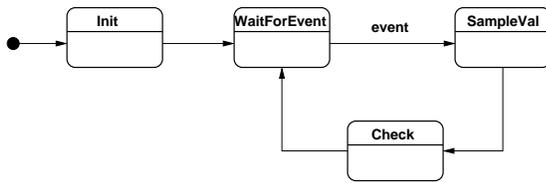


Figure 7: The Statechart for obsrvEventReceiver

The assertion monitor is a component that monitors the simulation of the UML model for any property violation. At the heart of the monitor lies an object named `obsrvEventReceiver`. This object performs all the verification tasks including the reception of the copies of events dispatched in the model. The dynamic behavior of this object is shown in the Statechart of Figure 7.

Tasks of the four states of the object `obsrvEventReceiver` are explained below. The transition, labeled with `event`, from state `WaitForEvent` to state `SampleVal` of Figure 7 signifies that control would jump from `WaitForEvent` to `SampleVal` when a copy of any event dispatched in the model will reach the object `obsrvEventReceiver`.

The tasks of different states of `obsrvEventReceiver` are explained below.

- **Init:** Initialization of data-structures for internal house-keeping
- **WaitForEvent:** Control waits for any event dispatch in the model
- **SampleVal:** Immediately after dispatch of any event in the model, control reads values of data variables necessary for verification
- **Check:** Control checks for any specification violation using the values read in `SampleVal` state. The salient steps are
 1. Retrieves the formula to be verified for the current step
 2. Reads values of the data variables relevant to the present status
 3. Substitutes the relevant values for the variables to check for any violation. If any violation is detected, it reports and exits. Otherwise, it prepares and saves the context of verification for the next step.

5.2 Visibility of data members and events

The addition of the assertion monitor as a Statechart inside the design hierarchy is not enough to be able to evaluate assertions on the Rhapsody run. The values of data members and event occurrence information is necessary to be able to evaluate the assertions. Unfortunately, we could not devise a mechanism to extract the data structure of the Rhapsody status and no Rhapsody routines are available for this purpose. We adopted a different approach to solve this issue.

To be able to read the values of the data variables of each object at each simulation step, the assertion-monitor object is integrated at the appropriate class hierarchy that reveals the basic building components of the assertion monitor and the relationships that the individual components share among themselves and with the

model elements. These relationships explain how the monitor can access the model-data variables during simulation.

Events in Rhapsody are dispatched with a particular target object. The occurrence information of any event is only available to the object that is its target and not to any other object in the system. For our work, we need the assertion monitor to have information about every event dispatched. This is done by modifying the event-dispatch mechanism inside Rhapsody. A Rhapsody macro, named `RiCGEN` that handles all event transactions inside Rhapsody has been modified so that a copy of every event dispatched in the original model is instantaneously delivered to the assertion monitor.

`RiCGEN` is a macro that is the core of the event transfer mechanism of Rhapsody. The sender element (actor, class etc.) of an event calls this macro with the name of the target element and the event. By modifying this macro, we ensure that whenever an event is thrown targeting any model element, a copy of the same event with same parameters, if any, would go to the *assertion monitor*. The overloaded `RiCGEN` routine is given below.

```

#define RiCGEN(INSTANCE,EVENT)
{
if ((INSTANCE) != NULL)
{
RiCReactive * reactive = &((INSTANCE)->ric_reactive);
RiCEvent * event = &(RiC_Create_##EVENT->ric_event);
RiCEvent * eventCopy = &(RiC_Create_##EVENT->ric_event);
RiCReactive_gen(reactive, event, RiCFALSE);
RiCReactive_gen( &((&obsrvEventReceiver)->ric_reactive),
eventCopy, RiCFALSE);
}
}
  
```

The element `obsrvEventReceiver` is a component object of the assertion monitor. Two important objectives have been achieved in this implementation of the overloading:

- Whenever this overloaded macro is called (by some model element to throw an event to some other), a copy of the same event is thrown to the object `obsrvEventReceiver`. This makes the assertion monitor aware of the interception of the corresponding event.
- Once this macro is called, two instances of the same event are pushed to the event queue consecutively. This is important, since, as mentioned previously, Rhapsody assumes a RTC semantics and at every step, a single event is dispatched. Whenever a dispatched event causes the model to move from one stable configuration to another, the next event notifies the monitor to perform the desired task.

5.3 Assertion monitor integration

For integrating the assertion monitor with the Rhapsody model under test as per our approach, one needs to have access permission inside the class hierarchy to be able to insert the assertion-monitor object. No modification of the original model is needed.

6. RESULTS

We deployed the DPV framework described above for the verification of two case studies, namely (a) an industrial implementation

of the Local Interconnect Network (LIN) protocol, and (b) an implementation of an access control mechanism in a web-based conference management system. Below, we describe the experimental results obtained on each of these.

6.1 Verifying an implementation of the LIN Protocol

Local Interconnect Network (LIN) [42] is a standard communication protocol used in automotive control subsystems. We developed a formal property suite of the LIN protocol in Action-LTL. The properties were extracted from the LIN 2.0 protocol specification document. The property suite consists of 30 properties expressing behavioral requirements on different components of the LIN subsystem. The DPV platform developed by us was tested on an industrial Rhapsody-based LIN implementation written in UML. Below, we present a few representative properties expressed in Action-LTL and verified by us using the DPV platform. The variables used in our properties are those used in the LIN implementation. The variables that begin with the prefix ‘ev’ represent events.

- P1: A slave task will always be able to detect the break/synch symbols sequence, even if it expects a byte field.
- P2: In slave mode, detection of break/synch sequence shall abort the transfer in progress and processing of the new frame shall commence.
- P3: Unconditional frames always carry signals and their identifiers are in the range 0 to 59(0x3b).
- P4: Response_Error shall always be set whenever a frame received by the node or a frame transmitted by the node contains an error in response field.
- P5: Successful_transfer shall be set when a frame has been successfully transferred by the node.
- P6: Both status bits are cleared after reading.

In Action-LTL, these can be expressed as:

```
P1: G[S1v.evBrkSync ⇒ X(S1v.state = RcvPID)]
P2: G[(S1v.state = Active) and S1v.evBrkSync
      ⇒ (S1v.state = RcvPID)]
P3: G[(S1v.FrmType = UNCONDITIONAL)
      ⇒ (S1v.PID ≥ 0x00 ∧ S1v.PID ≤ 0x3b)]
P4: G[S1v.evTrnsfrErr ⇒ (S1v.flg1 = 1)]
P5: G[S1v.evTrnsfrSucc ⇒ (S1v.flg2 = 1)]
P6: G[S1v.evRdStatus ⇒ (S1v.flg3 = 1)]
```

Table 1 shows the results obtained by our tool on a Pentium-4 with 1 GB RAM. The second column shows the number of properties used for our experimentation. In Column 3, we present the time required by the simulation platform to simulate the model without the assertion-checking feature, while in Column 4, we show the respective times required when the assertion monitor is inserted inside the simulation platform for verifying properties. Column 5 shows the number of properties verified using DPV. All these properties evaluated to true on the respective implementation models. The simulations were run for 1000 cycles (1000 iterations of the *while* loop in Algorithm 4.1). The choice of 1000 was arbitrary. For properties involving the always (future) operator, satisfaction (refutation) was verified as per the finite trace semantics.

Since most of the properties in our LIN property suite involved the always operator, our primary objective was to uncover refutation scenarios by simulating for a significantly large period and observing if there is a substantial increase in time requirements with and without the assertion-checking feature in our simulation framework. Results show that not much simulation overhead is incurred in terms of simulation time due to the presence of the assertion-checking engine inside the simulation platform.

6.2 Verifying access control policies in a conference management system

Our next case study was a web-based conference management system (CMS). In a typical web-based conference management system, authors are allowed to submit a paper online, and reviewers are allowed to review papers and submit their reports online as well. Access control policies are important to maintain the sanity of a conference management system. These access rights vary from one role to another (they are different for an author, a reviewer, the conference chair) and may also vary dynamically over time as the system changes. A few representative requirements are:

- P1: The same person should not be assigned as a reviewer for the same paper more than once.
- P2: No reviewer is allowed to modify his review report once he submits.
- P3: A reviewer is not allowed to see other review reports.
- P4: Submission of a paper by an author automatically disqualifies him from acting as the reviewer of the same paper in future.

Requirements such as those above are enforced by defining access rights to the different entities present in a CMS. Our goal was to verify whether a given CMS implementation respects the access control policies. The CMS implementation was an in-house one, developed in UML in the Rhapsody framework. We used the Action-LTL language to specify properties and our DPV framework to verify those properties.

7. DISCUSSION ON RELATED WORK

In this section, we discuss the prior work done in state/event based temporal logics in the context of concurrent programs, and the relevant validation efforts done in verification of UML-based designs.

7.1 State/Event Temporal Logics

The idea of combining state-based and event-based formalisms is not new. De Nicola and Vaandrager [27], for instance, introduced *doubly labeled transition systems*. Kindler and Vesper [22] proposed a state/event-based temporal logic for Petri nets. They motivated their approach by arguing, as we do, that pure state-based or event-based formalisms lack expressiveness in important respects. Huth et al. [21] also proposed a state/event framework, and defined rich notions of abstraction and refinement. In addition, they provided *may* and *must* modalities for transitions, and showed how to perform efficient three-valued verification on such structures. Giannakopoulou and Magee [18] defined *fluent* propositions in the context of a labeled transition system to express action-based linear-time properties. A fluent proposition is a property that holds after it is initiated by an action and ceases to hold when terminated by another action. Their work

Table 1: Results of DPV

Module	No. of Properties	Sim. Time (without DPV)	Sim. time (with DPV)	No. of Properties Checked
LIN	25	12.9 mins	15.4 mins	30
CMS	11	7 mins	10.2 mins	20

exploited partial-order reduction techniques and is implemented in the LTSA tool [26]. In a comparatively early paper, De Nicola et al. [28] proposed a process algebraic framework with an action-based version of CTL as specification formalism. In this approach, verification proceeds by first translating the underlying labeled transition systems (LTSs) of processes into Kripke structures and the action-based CTL specifications into equivalent state-based CTL formulas. At that point, a model checker is used to establish or refute the property. Dill [44] defined *trace structures* as algebraic objects to model both hardware circuits and their specifications. Trace structures can handle equally well states or events, although usually not both at the same time. Dill's approach to verification is based on abstractions and compositional reasoning. In general, events (input signals) in circuits can be encoded via changes in state variables. Browne made use of this idea in [8], which features a CTL* specification formalism. Browne's framework also features abstractions and compositional reasoning, in a manner similar to Dill's. In [10, 11], Chaki et al have proposed a framework in which both state-based and event-based properties can be expressed, combined and verified. The modeling framework consists of labeled Kripke structures (LKS), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with events. The specification logic, called SE-LTL is a state-event derivative of LTL. EAGLE [14] is a specification logic, popularly used in the Java verification community.

7.2 Verification of UML Statecharts

A number of different approaches for the verification of UML Statecharts have been developed. One of the most prominent approaches in this direction is ROOM [36]. Currently, the UML 2.0 proposal of the main tool vendors includes the basic ROOM concepts using the notation UML/RT [37]. However, there is only rudimentary support for non-functional system properties with respect to dynamic behavior in UML/RT. To overcome this deficiency, temporal extensions of OCL (Object Constraint Language) have been proposed. OCL has been primarily developed to express invariants attached to classes and pre-conditions and post-conditions of operations. By introducing additional temporal logic operators in OCL (eventually, always, never), verification engineers are able to specify the required behavior by means of temporal restrictions among actions and events [6]. Temporal extensions of OCL that consider real-time issues have been proposed for events in OCL/RT [9] and for states in RT-OCL [17]. Researchers [19] have also proposed a behavioral real-time model for timed UML Statecharts and a compositional verification approach.

While the work discussed above is mainly targeted towards enriching the expressiveness of UML constructs, thereby allowing specification of temporal constraints, the primary validation mechanism is dependent on the specific tool used for simulating the UML models. In recent times, there has been some research on employing formal verification tools for validation of UML Statecharts. The work [3] presents a detailed discussion on this. The main

principle behind this approach is to translate the Statecharts into some format that is 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 success or falsify the specification on the model. In [24], a simple (branching time) model-checking approach to the formal verification of UML Statecharts is presented using the AMC model checker available in JACK. The work of [25] provides a predefined set of checks of invariants, e.g. absence of deadlocks, queue overflows, and unreachability of invalid states. They use SPIN as the underlying model checker. The approach of HUGO [35] is tailored to the use-case of "drive to collaboration", that is, it is checked for a given set of objects whether the objects are able to adhere to a communication sequence given by a collaboration diagram. This work also takes timing annotations on state-machine transitions into account. HUGO [35] takes the approach of translating the model into the Promela language using SPIN [41] as the underlying verification engine. Another group of researchers present similar work [34], which has a verification environment for UML models, based on the model checker VIS [45]. In this approach, the UML Statecharts are first translated to XMI format and then converted to the input format of VIS. The requirements to be verified are specified using predefined temporal patterns on the graphical specification formalism Life Sequence Charts (LSC) [15]. The LSC are also translated to CTL queries and the model checker VIS is then invoked to check for specification violations. Similar approaches based on the specification language of the underlying model checker, are also reported [35, 38].

Graf et al. [29, 30] perform a verification of UML models by extending an existing automata-based validation framework, called IF [5], to support concurrency and communication aspects. IF supports all major UML constructs. IF itself has connections to explicit-state model checkers. On the specification side, they use UML observers that are similar to Statecharts. Another research in this direction is based on model checking of xUML [46].

A number of researchers have pursued the idea of requirements-driven validation in different contexts. The fundamental idea behind many of the approaches is to compile the requirements as monitor automata [12] for co-simulation with the design to be verified. In an earlier work [2], we have presented an approach for compiling assertions into Statechart monitors and co-simulating these monitors with the design under test. In this work, we adopt an on-the-fly approach for assertion monitoring based on property unfolding. Trace based monitoring and event-based runtime verification have been popularly used in a number of approaches [4, 12, 14, 40].

The main features that distinguish our work from those proposed in literature are as follows:

- The use of Action-LTL as a formalism for expressing prop-

erties over UML-based designs: Action-LTL allows expressing requirements involving arithmetic expressions over data members. The novelty of the logic lies in the fact that its atomic predicates connect to entities in UML models.

- The semantics of Action-LTL is defined in accordance to the simulation semantics of Rhapsody.
- The verification platform proposed in this paper is based on DPV. In [1], we have presented a brief architectural overview of the DPV framework.

8. CONCLUSION AND FUTURE WORK

The main focus of this work was to develop an assertion-based verification framework for verifying behavioral properties over UML models in accordance to Rhapsody simulation semantics. The expression language is a simple derivative of LTL suitable for capturing data attributes (Boolean and integer) and events of UML entities and the simulation framework is a simple and straightforward adoption of the standard property verification paradigm popularly used in hardware simulation-based verification in the Rhapsody context.

Our current research focus is in making this DPV platform more effective by the following enrichments:

- The simple syntactic structure of Action-LTL limits our expressive power to temporal properties over data members and simple events. To make Action-LTL more expressive, we are currently working on supporting events with payloads and more complex arithmetic expressions inside Action-LTL properties. This will enable us capture a wider range of properties.
- The assumption of the single event queue workflow model of Rhapsody adopted by us here limits the degree of concurrency that we can exploit in our verification framework. We are currently working on exploiting the scheduling semantics of Rhapsody to support the general concurrency model of communicating concurrent UML entities.
- Last but not the least, any simulation-based validation framework has the inherent issue of coverage as detailed in Section 2.1.1. The presence of an assertion guarding against an exception condition does not necessarily ensure that the assertion is checked unless simulation generates an event sequence in which the exception condition is generated and the assertion is violated. Random simulators fail to generate complex exception conditions in reasonable time. We are currently working on supporting coverage-driven simulation inside our DPV framework. The idea is to generate simulation events by analyzing the assertions so that the assertions are triggered during simulation.

The motivation of this work was to study the efficacy of a simulation-based verification paradigm in the UML context on top of Rhapsody. With the growing complexity of software and hardware systems and the focus towards UML-based high level design approaches, a validation platform is becoming indispensable. We believe that our verification framework is a suitable candidate for this purpose.

9. REFERENCES

- [1] Banerjee, A., Ray, S., Dasgupta, P., Chakrabarti P. P., Ramesh S. and Ganesan, P.V.V., A Dynamic Assertion-based Verification Platform for Validation of UML designs, In ATVA 2008
- [2] Banerjee, A., Ray, S., Dasgupta, P., Chakrabarti P. P., Ramesh S. and Ganesan, P.V.V., A Dynamic Assertion-based Verification Platform for UML Statecharts over Rhapsody, In IEEE TENCON 2008
- [3] Bhaduri, P. and Ramesh, S., Model Checking of Statechart Models: Survey and Recent Directions, 2004,
- [4] Bodden, E., "J-LO A tool for runtime-checking temporal assertions," Diploma Thesis , 2005.
- [5] Bozga, M., Fernandez, C., Ghirvu, L., Graf, S., Krimm, J., and Mounier, L., IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In Proceedings of FM 1999 (Toulouse, France), volume 1708 of LNCS, pages 307-327. Springer-Verlag, September 1999.
- [6] Bradfield, J., Kuester, F., and Stevens, P., Enriching OCL Using Observational mu-calculus. In R.D. Kutsche and H. Weber, editors, Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France, volume 2306 of LNCS. Springer, April 2002.
- [7] Bradfield, J., and Stirling, C., Modal Logics and Mu-Calculi: An Introduction, pages 293-330. Handbook of Process Algebra. Elsevier, 2001.
- [8] Browne, M.C., Automatic verification of finite state machines using temporal logic. PhD thesis, Carnegie Mellon University, 1989. Technical report no. CMU-CS-89-117.
- [9] Cengarle, M.V., and Knapp, A., Towards OCL/RT, In International Symposium of Formal Methods Europe, Copenhagen, Denmark, volume 2391 of LNCS, pages 389-408, Springer, 2002.
- [10] Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., and Sinha, N., Concurrent software verification with states, events, and deadlocks, Formal Aspects of Computing 17(4), 2005.
- [11] Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., and Sinha, N., State/event software based model checking, Fourth International Conference on Integrated Formal Methods (IFM) 2004, LNCS 2999, pages 128-147,
- [12] Chen, F. and Rosu, G. 2007. Mop: an efficient and generic runtime verification framework. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 569-588.
- [13] Clarke, E.M., Grumberg, O., and Peled, D.A., *Model Checking*, MIT Press, 2000.
- [14] d'Amorim, M. and Havelund, K. 2005. Event-based runtime verification of java programs. In Proceedings of the Third international Workshop on Dynamic Analysis (St. Louis, Missouri, May 17 - 17, 2005). WODA '05. ACM, New York, NY, 1-7.
- [15] Damm, W., and Harel, D., LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design, 19(1), pages 45-80, July 2001.
- [16] Dasgupta, P., A Roadmap for Formal Property Verification, Springer 2006.
- [17] Flake, S., and Mueller, W., An OCL Extension for Real-Time Constraints. In Object modeling with the OCL: The Rationale behind the Object Constraint Language, volume 2263 of LNCS, pages 389-408, Springer, 2002.
- [18] Giannakopoulou, D., and Magee, J., Fluent model checking

- for event-based systems. In Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), pages 257-266. ACM Press, 2003.
- [19] Giese, H. et al, Towards the Compositional Verification of Real-Time UML Designs, In ESEC/FSE 2003.
- [20] Harel, D. and Kugler, H., The Rhapsody semantics of Statecharts (or, On the Executable Core of the UML), LNCS vol. 3147, Springer-Verlag, pages 325-354, 2004.
- [21] Huth, R., Jagadeesan, and Schmidt, D., Modal transition systems: A foundation for three-valued program analysis. In Proceedings of the 10th European Symposium on Programming (ESOP), volume 2028 of Lecture Notes in Computer Science, pages 137-154. Springer, 2001.
- [22] Kindler, E., and Vesper, T., ESTL: A temporal logic for events and states. In Proceedings of the 19th International Conference on the Application and Theory of Petri Nets (ICATPN), volume 1420 of Lecture Notes in Computer Science, pages 365-383. Springer, 1998
- [23] Kozen, D., Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333-354, 1983.
- [24] Latella, D., Majzik, I., and Massink, M., Automatic Verification of a Behavioral Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637-664, 1999.
- [25] Lilius, J., and Porres, I., vUML: a tool for In Proceedings of the Automatic Software Engineering Conference (ASE), IEEE Computer Society, 1999.
- [26] LTSA website.
<http://www-dse.doc.ic.ac.uk/concurrency/ltsa/LTSA.html>
- [27] Nicola, R.D., and Vaandrager, F., Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458-487, 1995.
- [28] Nicola, R.D., Fantechi, A., Gnesi, S., and Ristori, G., An action-based framework for verifying logical and behavioral properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761-778, 1993.
- [29] Ober, I., Graf, S., and Ober, I., Validating timed UML models by simulation and verification. In Workshop SVERTS on Specification and Validation of UML models for Real Time and Embedded Systems, a satellite event of UML 2003, San Francisco, October 2003.
- [30] Ober, I., Graf, S., and Ober, I., Model Checking of UML Models via a Mapping to Communicating Extended Timed Automata. In 11th International SPIN Workshop on Model Checking of Software, 2004, volume 2989 of LNCS, 2004.
- [31] Object Management Group, *Unified Modeling Language Specification*, Version 1.4, Draft, OMG(2001), <http://cgi.omg.org/cgi-bin/docad/018214>.
- [32] Pnueli, A., The Temporal Logic of Programs. In *Proc. of Foundations of Computer Science*, pages 46-57, 1977.
- [33] Pnueli, A., Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, volume 224 of Lecture Notes in Computer Science, pages 510-584. Springer, 1986.
- [34] Schinz, I., Toben, T., Mrugulla, C. and Westphal, B., The Rhapsody UML Verification Environment
- [35] Schäfer, T., Knapp, A., and Merz, S., Model checking UML state machines and collaborations. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55(3), Elsevier, 2001.
- [36] Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.
- [37] Selic, B., Rumbaugh, J., *Using UML for modeling complex real-time systems*. Techreport, ObjectTime Limited, 1998.
- [38] Shen, W., Compton, K., and Huggins, J.K., A toolset for supporting UML static and dynamic model checking. In Proc. 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA, pages 315-318. IEEE Computer Society, 2001.
- [39] Somenzi, F., and Bloem, R., Efficient Büchi Automata from LTL Formulae, CAV 2000: 248-263
- [40] Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Barringer, H., Finkbeiner, B., Gurevich, Y., Sipma, H. (eds.) ISSAC 1982 and EUROCAM 1982. ENTCS, vol. 144, Elsevier, Amsterdam (2005)
- [41] The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, ISBN 0-321-22862-6.
- [42] The Local Interconnect Network Protocol, <http://www.lin-subbus.org/>
- [43] The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, ISBN 0-321-22862-6.
- [44] Trace theory for automatic hierarchical verification of speed-independent circuits. PhD thesis, Carnegie Mellon University, 1988. Technical report no. CMU-CS-88-119.
- [45] VIS: A system for Verification and Synthesis, The VIS Group, In the Proceedings of the 8th International Conference on Computer Aided Verification, p428-432, Springer Lecture Notes in Computer Science, 1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996.
- [46] Xie, F., Levin, V., and Browne, J.C., Model Checking for an Executable Subset of UML. In M. Feather and M. Goedicke, editors, *Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering*. IEEE CS Press, 2001.