VeriAgent: an Approach to Integrating UML and Formal Verification Tools[*]

Edjard Mota[1] **, Edmund Clarke**[2] **, Alex Groce**[2] **,**
Waleska Oliveira[1] **, Marcia Falcão**[1] **, Jorge Kanda**[1]

[1]Universidade Federal do Amazonas – Departamento de Ciência da Computação
Av. Gen. Rodrigo Octávio Jordão Ramos, 3000 – 69077-000 Manaus, AM

[2]Carnegie Mellon University – Computer Science Department
5000 Forbes Avenue – Pittsburgh PA 15213-3891

{edjard,woliveira,jkanda,marcia}@dcc.fua.br, {emc,agroce}@cs.cmu.edu

Abstract. *The mathematical notations of Formal Verification Tools (FVTs) do not prevent us from wrongly defining the behavior of systems, any more than UML-based CASE tools do. Both technologies have their advantages: respectively, precise and well defined semantics or high-level graphical notation. Unfortunately, these technologies are not fully integrated and usable in a single tool. With software rapidly growing in size and complexity, graphical specifications in languages like UML need to be formally verified, before the implementation phase, in order to guarantee the development of more reliable systems. While the enterprise of integrating CASE and FVTs has had reasonable success with the translation of simple diagrams to model checkers' notations, integration at the level of returning verification results to users was lacking. In this work we present a protocol interface for joining both technologies as a reliable solution to bridging this gap.*

1. Introduction

The mathematical notations of formal verification tools do not prevent us from wrongly defining the behavior of systems, any more than UML-based CASE tools do[1]. However, the languages used to build models in the former have precise semantics which allow us to find errors by means of automatic verification algorithms, e.g. model checking, which is not the case in the latter. CASE tools are considered to be user-friendly because of their graphical representation and their "capability" to provide free textual information to describe constraints not captured either by diagrams or constraint languages (for instance OCL). Thus, CASE tools lack the necessary apparatus to perform (semi-)automatic processing over models.

[*]Work partially supported by the Brazilian Research Council (CNPq), EsVIA Projet grant, nọ 550730/01-0.

[1]In this text we use the term CASE to mean the modern Computer Aided Software Engineering tools based on UML graphical notation, although it originally was used to describe tools to support the development of structured programming methods

With software rapidly growing in size and complexity, graphical specifications in languages like UML need to be formally verified, before the implementation phase, in order to guarantee the development of more reliable systems. A few years ago, the formal verification community began investigating mechanisms to integrate such graphical specifications with verification tools. While this approach achieved reasonable success on the translation of simple diagrams to model checkers' input notations, the results of verification were not well integrated back into the CASE tools' process. Interpreting the results of verification is still highly human-dependent, and the intensive use of these tools in software development is yet to be achieved.

We claim that a protocol interface joining both technologies can be a more reliable solution to bridging this gap. First, it would avoid the introduction of further notational overhead on either side. Second, we would be able to implement algorithms for reasoning about the relationships between high level specifications and verification results. Finally, such a protocol could also be used for guiding the verification result's explanation with an AI agent "flavor".

This way of bridging the gap allows a convenient marriage between the operations that map to models on both sides. This representation allows a "pre-processing" of system development models in such a way that when they are translated into formal specifications we can retain their original semantics. This means that the results of verification can be better mapped into causal explanations on the model.

The rest of this paper is organized as follows. In Section 2 we outline the fundamental ideas that motivated our research, and also the features a protocol interface should have in terms of language for intermediate representation and functionalities. Section 3 presents an implementation of such a protocol at the representation level. Section 4 describes an experimental result of an actual implementation of our approach for SMV translation, and and make a brief analysis pointing out the way to turn back the verification result in terms of error analysis. In Section 5 we describe our efforts in relation to other research in this area. Finally, in Section 6 we address some remaining open questions.

2. An AI Perspective for Integration

In this section we present the motivation of this work and the Artificial Intelligence (AI) perspective to integrating Formal Methods (FMs) and UML tools for system development.

2.1. The Reasons for an "Intelligent" Intermediate Level

One of the main purposes of Software Engineering (SE) is to enable developers to build systems that operate reliably despite their complexity. The formal methods community has developed many tools to help achieve this goal. The mathematical nature of such tools has not contributed to their adoption in the daily activity of software development. This may be because, in the past, software applications did not require rigorous definitions, as they were mostly focused on information systems. In order to handle complex information CASE tools were developed, and they evolved to the point of being semi-formal as is the case with many object-oriented tools such as those based on the Unified Modeling Language (UML).

UML CASE tools provide functionality for obtaining better abstraction with encapsulation capabilities, and offer a variety of mechanisms for defining the structure and behavior of systems. These mechanisms are not, however, sufficient for precise analysis as required for new market demands with respect to reliability. Such tools could aid in the development of safety critical systems if they were combined with formal methods to achieve this precision.

According to Clarke et al. [Clarke et al., 1996], in order to join these approaches in an attractive way some fundamental concepts should be developed and new tools should satisfy some criteria. These criteria are not necessarily related to formal methods. The current state of CASE technology does not embody three of these criteria, namely

composition: the combination of methods, specifications, models, theories, and proofs.

abstraction: the identification of levels of abstraction (possibly), taking the features of application domains into account.

reuse of models and theories: the use of parameterized models and theories, avoiding the need to start from scratch each time a new application is tackled.

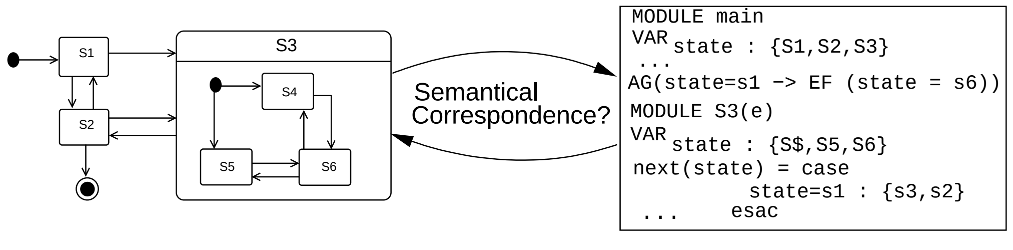
One might argue that these concepts are used in some way at both the semi-formal specification and formal verification levels. Unfortunately, the notations and therefore the embodiments of the concepts are very different. Even worse, if the notations do "match" it does not imply the combination is correct, as Clarke also pointed out [Clarke et al., 1996].

Scientific advances and technological developments in both areas have reached a point that trying to embody the features of one (FMs) into another (CASE tools), would be similar to building an "all purpose development tool." In order to join these disparate worlds it is necessary to introduce, on each side, translation mechanisms and interpretation methods in order to deal with the feedback from verification tools. This can be problematic for the following (not exhaustive) reasons.

1. UML itself is a miscellany of graphical language notations, and it seems unproductive to make a verification tool support all diagrams. However, if any one is chosen, there could be application domains where the others would be more relevant.
2. Direct mapping of UML diagrams into a formal notation may bring semantical correspondence problems (depicted in Figure 1), such as:

   the loss of the original UML model at the verification level

   the difficulty to mapping back the result of verification onto the original model.



```
MODULE main
VAR state : {S1,S2,S3}
 ...
AG(state=s1 -> EF (state = s6))
MODULE S3(e)
VAR state : {S$,S5,S6}
next(state) = case
         state=s1 : {s3,s2}
 ...    esac
```

Figure 1: Problems with semantical correspondence between UML and formal notation.

3. Large scale software development deals with huge flows of information across model diagrams, team notes, etc. To manually map all of this to a verification model would be unproductive, as the time a product has to reach the market is small compared to the time necessary for training dozens of developers in using formal verification tools.

A single solution to all of these problems seems to be a nearly impossible target. The concepts associated with their union, however, point to the crystallization of a computational entity suitable to mediate the integration of both CASE and formal verification tools. The concept of a protocol has allowed better management of huge flows of information across computer networks and the Internet. In the same way, an interface (or protocol) layer between CASE and formal verifications tools would be more likely to succeed in scaling than either one alone.

The methodology we used to analyse the requirements for such a layer is based on the concept of intelligent agent from Artificial Intelligence [S. et al., 1995], and we claim it is justified for the following reasons.

- Automatic translation, alone, from models in UML to models for verification is useless without tools for managing this translation and applying it to handle complex systems. For instance, cause analysis of errors in such systems is still basically done by specialists with no automatic support at all.
- There is lot of knowledge in between both worlds not associated to algorithmic verification or high level modeling *per se*. For example, synthesis is not meant to be only simple translation or cross-execution. It could also be used to search for a similar model (or parts of one), which has already been verified and simply give back previous verification results.
- The "computational skill", or in AI jargon *reasoning mechanism*, necessary to find common patterns between a new model of a system and old ones requires a representation approach able to capture common features using structural language elements as much general as possible. In this way, we may reify dynamic behaviour into "static relations" or snapshots of the system's behaviour.
- For very large systems the more a development tool is able to find out patterns previously verified, the stronger will be confidence in the composition of them.

These features are usually named as synthesis, compositional reasoning, and error explanation, and are all important aspects associated to verification presently lacking in automatic verification tools. The emphasis of our solution is that a layer to join both worlds must have some way of knowing, out of many possible interpretations (executions or inferences) of a model it could draw, which ones it should actually draw. In other words, which ones are rationally better (computationally less expensive) than others.

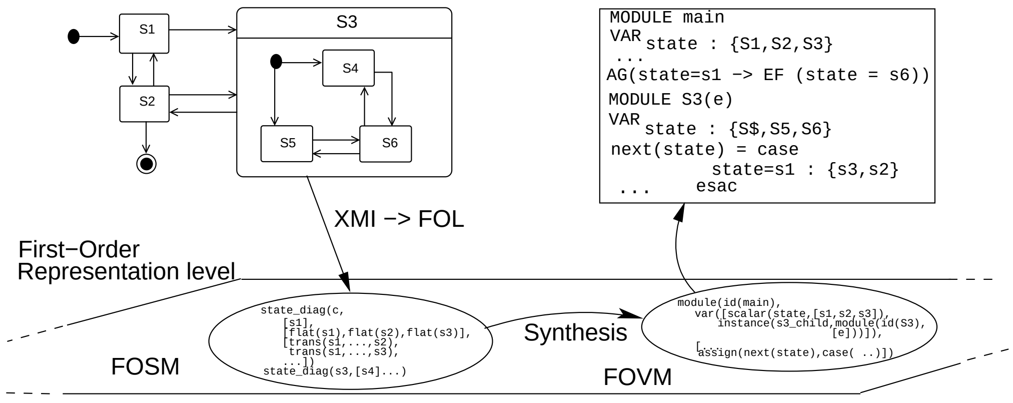
2.2. A First-Order Language Layer

The first aspect of an interface protocol for data exchange is the language for representing the information it is supposed to process. As we are dealing with two different kinds of notation this common language must be as general as possible in terms of structural representation of information. First-Order Logic (FOL) is most suitable for the following reasons.

- FOL is "computationally universal" in that any problem with a computational solution can be described in it. Moreover, such descriptions can be reduced to Horn clauses for logic programs: a set of axioms and rules defining relationships among objects.
- There are many efficient inference engines for handling first-order expressions or logic programs. These expressions are, by definition, well-structured elements allowing us to create abstractions to represent any computational model.
- FOL can be used to *fiber* other logical representation languages [Gabbay, 1999], and so we may perform, among other tasks, synthesis across UML models and formal verification notation.
- The strong relation between syntax and semantics of FOL representation may produce a side-effect on the discipline of using graphical tools. The reason is that while we may keep the original functionalities of a graphical tool, we may warn developers not to "abuse" the graphical freedom of design. Provided the discipline is followed, we should be able to better identify patterns of specification and their relations to patterns of verification models.

Another interesting advantage of using FOL is that the recent demands for model exchange across enterprise applications brought about the XML Metadata Interchange (XMI) [OMG, 2002]. Any valid XML description is associated to a Domain Object Model (DOM). As DOM descriptions are easily mapped into first-order expressions, all modern UML-based CASE tools which export to XMI can be used in our architecture (see Section 4 for an example).

Figure 2 depicts the idea of the sort of integration we are proposing. We map a system model UML statechart represented, for example, in XMI, into its first-order representation [Falcão et al., 2002].



```
MODULE main
VAR state : {S1,S2,S3}
 ...
AG(state=s1 -> EF (state = s6))
MODULE S3(e)
VAR state : {S$,S5,S6}
next(state) = case
          state=s1 : {s3,s2}
 ...    esac
```

```
state_diag(c,
   [s1],
   [flat(s1),flat(s2),flat(s3)],
   [trans(s1,...,s2),
    trans(s1,...,s3),
   ...])
 state_diag(s3,[s4]...)
```

```
module(id(main),
   var([scalar(state,[s1,s2,s3]),
       instance(s3_child,module(id(S3),
                          [e]))]),
   [...
   assign(next(state),case( ..)])
```

Figure 2: The interface layer to join UML tools and formal verification.

A First-Order Specification Model (FOSM) representation can be used as a knowledge base for reasoning about structural properties of the model and consistency checking. Other applications are possible including synthesis (as depicted at the flat level of Figure 2). This is used, for example, in transforming a FOSM description into a First-Order Verification Model (FOVM) which represents the target verification notation, which is then easy to translate directly into the formal verification tool's input notation.

A FOVM is derived from a FOSM by means of transformation rules when requested. At first glance it might seem strange to T-encode computational models of verification methods when they offer suitable notation to represent reactive (embedded) systems, and also T-encode inference mechanisms for proving properties about them. In most of these problems a finite state machine (FSM) [2] is sufficient to represent the system behavior.

Along with FOSM and FOVM, the reprerentation level should also have ways of representing the results of verification in first-order notation. This information we call First-Order Verification Result (FOVR). To this triple we name *pattern of verification*. Our proposition is that a **VeriAgent** needs a representation for patterns of verification in order to perform meta-reasoning about problem descriptions and represent knowledge about verification. These patterns need not be represented in the original computational model notation of formal methods.

2.3. The Inferences or Functionalities Layer

This layer is an open set of reasoning mechanisms to actually manipulate FOSM, FOVM and FOVR. Such mechanisms or functionalities are not directly associated with formal verification methods. It is open because we allow the introduction of new mechanisms to extend the functionalities of a **VeriAgent**. Some of these functionalities have been developed in the last ten years or so, and others have never been realized. Our non-exhaustive list includes:

Synthesis to transform XMI notation into FOSM, and also to map (reduce, compose, fiber, etc.) first-order representations into one another.

Property extraction to generate interesting properties, candidates for verification, as logical formulas.

Formal Code Generation to allow the translation of FOSM notation into formal language notation. For instance, NuSMV for Model Checking.

Choice of tools to allow developers and experts to pick a suitable verification tool given a new system specification model. This task certainly needs to interact with synthesis.

Verification Results to reason about patterns of specification and their relation to patterns of verification associated with the history of solutions to errors previously found.

Error Explanation to choose the appropriate means for explaining the errors found, and to aid in debugging the model and specification.

Compositional Reasoning to exploit the organization of systems as interacting components and to isolate components already verified from those with potential errors.

Historical Reasoning to keep track of which aspects of system development have been formalized and verified. This is actually the third layer to store patterns of reasoning about the verification of certain models (possibly associated with specific application domains). This knowledge base depends on the first-order representation.

To carry out the tasks listed above we propose a *Verification Agent* (**VeriAgent** henceforth) as a computational component used to combine Formal Methods and modern

[2] FSM or one of its well known extensions, e.g.Fair Transition Systems [Manna and Pnueli, 1991].

modeling languages, e.g. UML. This agent is composed of three layers or components as depicted in Figure 3. The system's developer interacts with the Graphical Services interface, for example UML. After defining a new model she/he aks the **VeriAgent** to check for systems' correctness. The model is then translated into a FOSM representation, and the user is requested to choose, from a list of generated properties, some he/she wants to check. The agent may perform synthesis and generate a FOVM model. The agent searches for a pattern of system's verification that matches these two models, by using some kind of semantical unification as done in [Mota, 2000] for temporal reasoning. If some some is found, then the agent simply retrieves it and give back to the developer the corresponding FOVR, otherwise it breakes the original model and apply the same procedure to each submodel. If it fails then it will call the verification tool passing as arguments the property and the FOVM submodels into the appropriate notation for the chosen tool.

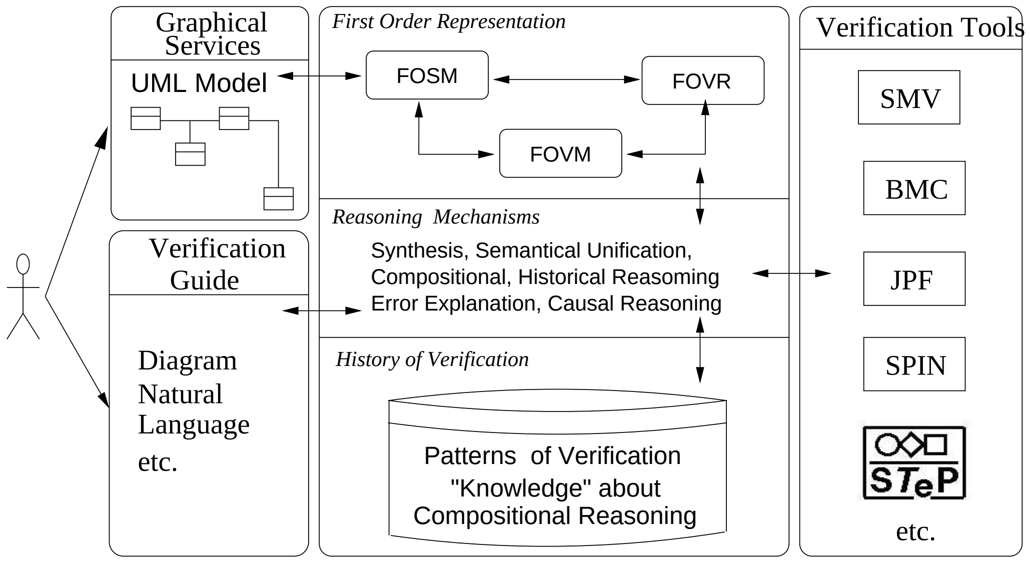


Figure 3: The VeriAgent architecture.

The module named Verification Guide is not in the core of the **VeriAgent**, because it is a kind of interface which can be present in any of the approaches listed in the figure. This module presents the outcomes of the reasoning module (mainly for error explanation and causal analysis) to the developer in a suitably helpful way. This can often probably be accomplished by extending the graphical tool interface with the necessary features for that purpose.

3. A VeriAgent Implementation

3.1. LogUML: a FOSM Based on Logic Programming

The purpose of FORL is not to execute a FOSM, FOVM or FOVR in standard computational logic fashion, but rather to use well known logical inference engines to reason about patterns of both worlds and also about the results of verification. Thus, this layer is composed of three model theories represented as sets of well-formed structured expressions in first-order logic (FOL). At this level we use the T-encode Hilbert style of representation [Schumann, 2001] to deal with those relations that, in their original computational model, would represent relations, concurrent processes, etc.

In our case we are dealing with UML diagrams—in this work just statecharts. The logical form we use as an example is a subset of the Logic Programming (LP) [Lloyd, 1993] language. Apart from having a strong relation between syntax and semantics, LP has a powerful meta-language mechanism for prototyping of any other notation we want to propose or extend LP itself. We shall use Prolog despite the fact that it is based on standard first order logic language, and so we have no structure for representing concurrency in a suitable way. However, we are still able to map "states of concurrency" into structured terms of our object language notation. This means that relations or even processes specification can be reified into terms.

For the purpose of this paper we shall concentrate only on the representational aspects of UML statecharts and its counterpart in a formal verification framework. For lack of space, not all of these concepts will be shown here in this work. In what follows we define a *Logic for Unifying Modeling Language*, or simply *LogUML*.

3.2. Vocabulary

The idea is basically to extend the usual set of symbols as follows.

variables $\mathcal{L}_v$ is extended with the finite set $\mathcal{V}_{uml}$ of LogUML variables.

constants $\mathcal{L}_c$ is extended with the finite sets $\mathcal{C}_{uml}$ for names of classes, $\mathcal{S}_{names}$ for names of states, $\mathcal{D}_{names} \subseteq (\mathcal{C}_{uml} \cup \mathcal{S}_{names})$ for names of diagrams, and $\mathcal{S}_{=}$, $\mathcal{S}_{\textcircled{=}}$, $\mathcal{S}_{\mathrm{A}}$, $\mathcal{S}_{\textcircled{A}}$ for flat, super flat, advanced and superadvanced states, respectively. $\mathcal{N}_{par}$ for names of parameters, and $\mathcal{N}_{att}$ for attributes names, $\mathcal{N}_{oper}$ for names of operations and $\mathcal{N}_{\tau}$ for names of valid LogUML set types.

$\mathcal{E}_{uml}$ is the finite set $\mathcal{A}_{adv} \cup \mathcal{T}_{uml} \cup \{operations/1, param/1, attributes/1, flat/1, advanced/2, super/1, impl/2\} \cup \mathcal{P}_{\tau}$ of special functions for representing elements of the LogUML language, where $\mathcal{A}_{adv}$ is the set $\{entry/1, exit/1, newtarget/1, do/1, defer/1\}$ $\mathcal{T}_{uml}$ is the set $\{trans/5, event/1, cond/1, action/1\}$ $\mathcal{P}_{\tau}$ is the set of unary *function-types* $\{t_1(x_1), \ldots, t_n(x_n)\}$, and each $x_i \in \mathcal{N}_{par} \cup \mathcal{N}_{att}$, and $t_i \in \mathcal{N}_{\tau}$.

We add the special predicates $class/3$ and $state\_diag/4$ into the set of predicates $\mathcal{L}_P$.

Logical Symbols are the usual ones in standard Logic Programming.

3.3. Classes of Expressions

The classes of expressions we want to represent should reflect the elements of an UML specification into the elements of LogUML language. For this work the definition of a LogUML description of a system constitutes the specification of its structure and of its behaviour. A structure of a class represents its interface with the environment and so it should embody the class name, its attributes and operations.

The behavior is captured through a diagram which embodies states and transitions. In what follows we shall formalize these concepts and get into a more detailed definition of each one.

attribute is a term of the form $attrib(X)$ and $X \in \mathcal{P}_{\tau}$

parameter is a term of the form $param(X)$ and $X \in \mathcal{P}_\tau$

operation is set of terms in the form $t_i(T, Pars)$, where $t_i \in \mathcal{N}_{oper}$, $T \in \mathcal{P}_\tau$, and $Pars \subseteq \mathcal{N}_{par}$;

state is an element of the set $\mathcal{D}_s = \mathcal{S}_= \cup \mathcal{S}_{\textcircled{=}} \cup \mathcal{S}_{\mathrm{A}} \cup \mathcal{S}_{\textcircled{A}}$, where elements of
$\mathcal{S}_=$ are of the form $flat(s)$,
$\mathcal{S}_{\textcircled{=}}$ are of the form $superflat(s)$,
$\mathcal{S}_{\mathrm{A}}$ are of the form $advanced(s, la)$,
$\mathcal{S}_{\textcircled{A}}$ are of the form $superadvanced(s, la)$,
and $s \in \mathcal{S}_{names}$, and $la \subseteq \mathcal{A}_{adv}$.

transition is a term of the form $trans(s_1, event(E), cond(C), action(A), s_2)$, where $s_1, s_2 \in \mathcal{S}$, E is a list of events, C is a "boolean condition" and A is a list of actions. Actions can have conditions to be activated, for which we use T-encode [Schumann, 2001], to encode the implication (X $\rightarrow$ Y) with the function $impl(X, Y)$.

class is an association of a class name C, attributes A and operations O, written $class(C, attributes(A), operations(O))$.

state diagram is an association of Diagram name, a set of initial states, a set of states, and a set of transitions, written as
$state\_diag(D, I_s, S, T)$,
where $D \in \mathcal{D}_{names}, I_s \subset S_{names}, S \subseteq \mathcal{D}_s, T \subseteq \mathcal{T}_{uml}$

LogUML class is a term of the form $class(C, A, O)$, where C is the name of the class, A is a set of attributes and O is a set of operations.

3.4. An Example of Mapping UML into LogUML

Figure 4 shows a simple UML model for a thermostat. This model was defined using the **ArgoUML** tool which exports diagrams to XMI by using a plugin added into ArgoUML called PLogAr, generating a LogUML code.

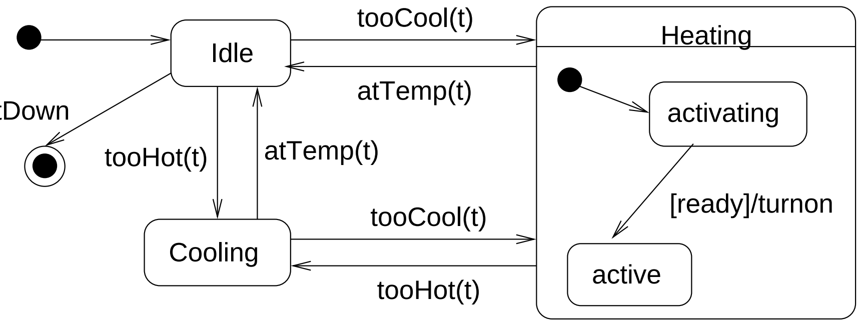


Figure 4: Thermostat Statechart Model

The translation from UML statechart into LogUML is not so difficult and for this example is actually very simple, since we have defined the UML elements as recursive first-order terms with a finite computation on the translation. This example above is written in LogUML as follows.

For the static pat of the model we shall interpret the thermostat as a single unbreakable class, but it could be different. So we have the following class model definition.

```
class(thermostat,
      attributes([float(t)]),
      operations[ready(boolean()),
       shutDown(boolean()),
       atTemp(boolean(),float(t)),tooHot(boolean()),
       tooCool(boolean()),turnOn(boolean())]).
```

The dynamic part of the thermostat system model is declaratively written as follows.

```
state_diag(thermostat,initialstates([idle]),
       [flat(idle),flat(cooling),superflat(heating)],
   [trans(initial,event(_),cond(_),action(_),idle),
    trans(idle,event(shutDown),cond(_),action(_),final),
    trans(idle,event(tooHot),cond(_),action(_), cooling),
    trans(idle,event(tooCool),cond(_),action(_),heating),
    trans(cooling,event(tooCool),cond(_),action(_),heating),
    trans(cooling,event(atTemp),cond(_),action(_),idle),
    trans(heating,event(tooHot),cond(_),action(_),cooling),
    trans(heating,event(atTemp),cond(_),action(_),idle)]).
state_diag(heating,initialstates([activating]),
   [flat(activating),flat(active)],
   [trans(activating,event(ready),cond(_),action(turnOn),active),
    trans(initial,event(tooCool),cond(_),action(_),activating)]).
```

4. Mapping UML to SMV

In this section we describe the experimental result of an actual implementation of a **VeriAgent** tool. A detailed description of the tool, using **ArgoUML** [3] is available [Falcão et al., 2002]. The purpose here is to present part of the behaviour of an implementation of **VeriAgent**.

Note that during the translation from XMI to **LogUML** the pseudo-states `initial` and `final`, are introduced for completeness reasons. This notation does not need to be shown to the developer as it is used as intermediate data for other purposes (see Section 3). Synthesis is called within the **ArgoUML** environment, and is performed by a Logic Program which implements the translation chosen by the modeler. Such translations are seen, from the **VeriAgent** perspective, as first-order interpretations. Here, we map UML to Nu-SMV's input notation using the following translation assumption.

4.1. An SMV Interpretation of UML Statecharts

From the (object-oriented) programming level perspective, event, action and condition have the same semantics, i.e. they are just methods. What each one does and returns as a result will differentiate one from another. As there is no meaning for a transition with only a (guard) condition [Dennis et al., 2002] and no event associated we do not consider

[3]ArgoUML is a Copyright of Tigris Open Source Community which promotes Open Source Software Engineering.

that case. Thus we simply add scalar variables to each state to represent the states of the statechart. Due to space restrictions this paper only considers events and states. There is also one type of module variable (with possibly more than one instance). We shall use only ASSIGN expressions in SMV to represent the transitions to change these variables. This gives us the following mapping:

Every statechart predicate `state_diag(D,I,S,T)`, where `S` is the set `s1,...,sk` of states, `T` is the set of transitions, and `E` is the set {`e1,...,en`} of events which trigger transitions in `T`, is translated intto

```
MODULE D;
VAR state :  S;
     event :  E;
si_child :  Si(event);
INIT
   init(state) := initial
   init(event) := default
ASSIGN
    next(event) := E; next(state):=
    next(state) := case ...
                     state = s & e :  s_j; ...
                   esac;
    MODULE Si(e); ...
```

where every line of a `next(state)` case expression is associated with a transition in the state diagram of the form `trans(r,event(e),cond(c),action(a),s)`. Note that here we ignore conditions and actions as explained above.

Our mapping observes some basic principles. First, statecharts at level zero are mapped to “`MODULE main`”, and superstates are modules called by the module that it sees events rise from. In the translation schema above `D` can call `main`, and all its sub-machines are modules. The hierarchy is derived by simply creating a child variable in every module which has a superstate.

4.2. The Thermostat Example

Using the rules established in the previous section the thermostat diagram is translated from **LogUML** into the following SMV code:

```
MODULE main
VAR
   state  : {heating,cooling,idle,final,initial};
   event  : {default, atTemp, tooCool, tooHot, shutDown};
   heating_child : Heating(event);
ASSIGN
init(state) := initial;  init(event) := default;
next(state) := case
            state = initial & event = default : idle;
            state = heating & event = tooHot  : cooling;
            state = heating & event = atTemp  : idle;
            state = cooling & event = tooCool : heating;
            state = cooling & event = atTemp  : idle;
            state = idle & event = shutDown   : final;
            state = idle & event = tooHot     : cooling;
            state = idle & event = tooCool    : heating;
            1                                 : state;
          esac;
next(event) := {default, atTemp, tooCool, tooHot, shutDown};
MODULE Heating(event)
VAR
   state  : {initial,active,activating};
ASSIGN
init(state) := initial;
next(state) := case
                  state = initial & event = tooCool : activating;
                  state = activating                : active;
                  1                                 : state;
                  esac;
```

Note that we do not derive properties to be verified. In our tool the user still has to enter the Computational Tree Logic (CTL) formula or Linear Temporal Logic formula to be checked. Suppose we want to prove the following safety property "the thermostat will never stay in the active state when the temperature is too high." This is represented in CTL as follows:

```
SPEC AG ((state = heating & event = tooHot)
     -> AX (! heating_child.state = active))
```

Verification and Analysis

After translating the original model into LogUML and then into SMV, we are able to model check the translation (which has the same name as the model with the extension ".smv"). We used the **NuSMV** [Cimatti et al., 2002] model checker and ran the verification as follows, where the option `int` runs an interactive mode of NuSMV, and the command `check_spec` verify all specifications given:

```
[mozart@smv-examples]nusmv -int thermostat2.smv
*** This is NuSMV 2.1.2 (compiled 2002-11-22 12:00:00)
NuSMV > go
NuSMV > check_spec
-- specification AG ((state = heating & event = tooHot)
   -> AX (!heating_child.state = active)) is false
-- as demonstrated by the following execution sequence
-> State 1.1 <-
    state = initial
    event = default
    heating_child.state = initial
-> State 1.2 <-
```

```
    state = idle
    event = tooCool
-> State 1.3 <-
    state = heating
    event = tooHot
    heating_child.state = activating
-> State 1.4 <-
    state = cooling
    event = atTemp
    heating_child.state = active
```

The result says that the CTL formula given is false, which means that the the safety property "the thermostat will never stay in the active state when the temperature is too high" is false because the value of the `heating_child.state` is always active once it is reached. Examining Figure 4, we can see that, in fact, the statechart does not have a transition leaving the `active` state.

In general, developers who use UML would say that, because of UML semantics "it is assumed that the events `tooHot(t)` or `atTemp(t)` should change the state." We think this is a misleading assumption as the heating could be an independent component plugged into the model, and we could never say that it works properly unless it has been modeled properly, or assumptions have been explicitly stated.

Note that the **VeriAgent** still does not generate this sort of analysis, but as we mentioned in Section 3.1, explanation of errors is a goal of future work [Groce and Visser, 2003].

After a hand translation of the model to ANSI C, we applied our current error explanation algorithm, which is based on distance metrics for executions of a system, to the example. In this case, of course, the most useful explanation of the error is at the level of assumptions, but the automated explanation is still instructive.

The basic explanation is in terms of the most similar execution that does not result in an error. In this case, changing the event in state 1.3 from `tooHot` to `default` and leaving all other values unchanged results in a different transition being taken (the state at 1.4 is thus `heating` rather than `cooling`). Generating causes (by a method analogous to David Lewis' counterfactual analysis [Lewis, 1973] shows that the error is causally dependent on the second event being `tooCool` and the third event being `tooHot`.

5. Related Work

Clarke [Clarke et al., 1996] pointed out that the possibility of "the role of formal methods in the entire system development process increases, especially as the tools and methods successful in one domain carry over to others." Formal methods should complement less formal methods that are used in the overall system development process. The goal is not to replace these methods, but to work with them to improve requirements analysis, refinement, and testing.

The basic need is for understanding of how to compose methods, specifications, models, theories and proofs, means to identify different kinds of abstractions of the model, and techniques for reuse and parameterize models and theories.

Some work, like pUML [Evans et al., 1999], is concerned with giving more formalism to UML using a meta-model defined in terms of the abstract syntax (natural language description), well-formedness rules (like OCL), and modeling language (natural language). The trend is towards UML as a well-defined modeling language, with a more precise semantics, in order to reason about models and verify the correctness of designs.

Others have translated UML models into the notations of verification tools. One interesting example [Clarke and Heinle, 2000] translates statecharts from the STATEMATE tool into the notation of the SMV model checker.

ASM UML [Compton et al., 2000] formalizes UML using Abstract State Machines (ASM) to give a semantic model to UML, and then uses the ASM model in a verification tool for UML.

The vUML of TUCS Research group [Lilius and Paltor, 1999], is a tool that automatically verifies UML models where the behavior of the objects is described using UML Statecharts diagrams. It uses the SPIN model checker to perform the verification, but substate machines are not handled. Moreover, the authors also agree that direct translation alone cannot cope with the loss of semantics because the state machine is flattened and all hierarchical information is lost.

These works are focused on the use of model checking. A common level of specification allows use of both theorem proving and model checking as the verification agent will be able to decide the best formalism to verify a given problem specification. The **VeriAgent** also allows the use of other tools, e.g. **STeP**.

The Prosper project [Holt, 1999] advocates the use of toolkits which allow existing verification tools to be adapted to a more flexible formalism so that they may be treated as components.

6. Conclusions and Further Work

In this work we propose an approach for bridging the gap between software specification tools based on UML and formal verification tools that relies on an interface layer joining both kind of tools. By using a common level of representation based on FOL we are able to build suitable mechanisms for keeping track of the aspects of system development which are verified. The advantages of this are that:

- formal methods do not need to be adapted to meet the user-friendliness demands of the current development tools market;
- specification tools may have a kind of *verification plug-in*, where the agent verifier may help to choose a suitable verification method for specific application domains based on a library of patterns of verified systems;
- current efforts at translating UML specifications can use the intermediate level to improve translations.

We plan to investigate the following topics in future work:

- Automatic property extraction from UML diagrams. This will allow the modeler to chose the kind of property to prove (safety, reachability, etc.)

- Full integration of inference engines to perform reasoning tasks. In the current state of the implementation the modeler still has to call a Prolog program from the command line.
- Better translation mechanisms to deal with events local to sub-state machines, conditions and actions. This will aid the mechanisms for causal analysis of errors to detect which actions (or even state activities) are relevant to an error.
- Integration of error-explanation techniques based on distance metrics between executions.
- Transformation of counterexample analysis [Groce and Visser, 2003] results into a natural language style. The current output is quite difficult to understand if a large system is being verified.

7. References

References

Cimatti, A., Clarke, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tachella, A. (2002). NuSMV 2: An open source tool for symbolic model checking. In *Computer Aided Verification*.

Clarke, E. and Heinle, W. (2000). Modular translation of statecharts to smv. Technical report, Carnegie Mellon University.

Clarke, E. M., Wing, J. M., and et. al. (1996). Formal methods: State of art and future directions. *ACM Computing Surveys*, 28(4):626–643.

Compton, K., Gurevich, Y., Huggins, J., and Shen, W. (2000). An automatic verification tool for UML. CSE 423-00, Michigan University.

Dennis, A., Wixon, B. H., and Tegarden, D. (2002). *Systems Analysis & Design: An Object Oriented Approach with UML*. John Wiley & Sons, INc.

Evans, A., France, R., Lano, K., and Rumpe, B. (1999). Meta-modeling semantics of uml. http://www.cs.york.ac.uk/puml/papers/pumluml99.pdf. Computer Science Department - University of York.

Falcão, M., Leite, J., Gonzales, D., Souza, M., and Lima, D. (2002). Uma extensão do ArgoUML para verificação formal. Technical report, Universidade Federal do Amazonas. EsVIA/CNPq Grant Nọ 550730/01-0.

Gabbay, D. (1999). Fibring Logics. Oxford University Press.

Groce, A. and Visser, W. (2003). What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135.

Holt, A. (1999). Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, 24:253–257.

Lewis, D. (1973). Causation. *Journal of Philosophy*, 70:556–567.

Lilius, J. and Paltor, I. P. (1999). vuml: a tool for verifying uml models. Technical report, Abo Akademi University.

Lloyd, J. (1993). *Foundations of Logic Programming*. Springer Verlag.

Manna, Z. and Pnueli, A. (1991). *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag.

Mota, E. (2000). Cyclical and granular time theories as subsets of the herbrand universe. In *Principles of Knowldege Representation and Reasoning*, pages 366–377, Breckenridge, Colorado - USA. Morgan Kaufmann.

OMG (2002). Omg xml metadata exchange specification. URL:www.omg.org. Acessed on October 2002.

S., Russel, and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Schumann, J. (2001). *Automated Theorem Proving in Software Engineering*. Springer Verlag.