AUTOMATED PROPERTY VERIFICATION IN UML MODELS

Ambrosio Toval (atoval@um.es)
Dpto. de Informática y Sistemas
University of Murcia
Campus de Espinardo
Murcia – 30071 – Spain

José Sáez (jsaez@um.es)
Dpto. de Informática y Sistemas
University of Murcia
Campus de Espinardo
Murcia – 30071 – Spain

Francisco Maestre (fmm1@alu.um.es)
Dpto. de Informática y Sistemas
University of Murcia
Campus de Espinardo
Murcia – 30071 – Spain

ABSTRACT

This paper describes the design and implementation of a framework for automated property verification in UML models. The framework consists of a property manager interface, which allows the user browsing and selecting properties to be checked in the model, a set of verifiers, and a hierarchy of properties, each of which knows how to check a number of features or constraints in the model. The framework allows the addition of different verifiers and property sets. In order to illustrate the feasible application of the framework, an instantiation to UML statecharts verification is also presented. The framework can be used to construct applications, using the Java language, aimed at checking the fulfilment of properties in UML models, or integrated in tools with a specific interface for accessing all modelling elements, such as RIVIERA, into which it has been integrated..

KEYWORDS

Software Engineering, UML model verification, UML model testing, property verification, statecharts

1. INTRODUCTION

UML models are increasingly complex and comprehensive, including many different diagram types. New approaches to software development such as MDA [25, 26], that emphasize the use of UML models, stress the importance of ensuring the correctness of these. This is mainly true when dealing with critical systems where human life or expensive systems can be exposed to danger. Defect detection should be done as soon as possible in every phase of the development process, but specially when modelling, because of the high cost of faults in early stages. The finding of design faults should be as automated as possible.

On the other hand, and in spite of the proliferation of UML CASE tools [11], only a few of them are able to produce models which have been validated against user requirements or verified for conformance to particular constraints or properties. Some studies [19, 20] that show that CASE tools are seldom used, claim that many CASE tools are oriented towards particular development approaches, their use tends to be expensive (licenses, training…), they lack measurable benefits and they do not fulfil the user's unrealistic expectations.

Fortunately, this trend is changing, and emphasis on V&V (Validation & Verification) and M&S (Modelling and Simulation) is helping to move forward the next CASE tool generation, where a growing number of tools aim at assuring the quality of models by checking whether some properties are satisfied or not [8, 13, 17].

This paper focuses on the design and implementation issues of a framework useful to support automated property verification for UML models. The Property Verification Framework (PVF from now on) is composed of a generic Property Manager (PM) which shows the set of available properties, classified according to diagram types, and other relevant information for each of the properties, which can be browsed by the user. He or she can then select explicitly which properties should be checked. Then, a verifier -there will be one for each diagram type-, will be the responsible for contrasting the model with respect to each selected property and collecting the results in a final report. Each verifier can use different methods to test whether the model conforms to the property or constraint, such as using a Java function or invoking external tools, such as theorem provers (this will be the subject of future work). The PVF is illustrated by applying it to the verification of statechart properties. This framework can be used to construct applications, using the Java language, aimed at checking the fulfilment of properties in UML models.

The RIVIERA (RIgorous VIrtual Environment for Requirement Analysis) project [12] aims at developing a framework for the construction of a CASE tool which has three main purposes, regarding Software Engineering models: 1) to serve as a simulation environment to execute models, in order to validate user requirements, 2) to support verification of properties on the models, and 3) to provide a transformation environment so that changes can be suggested and made to the model according certain criteria. The RIVIERA approach is based on offering an open and resilient framework within which different components can be included, extending its capabilities. In addition, this should allow the creation of new tools by reusing the main architectural design.

The rest of this paper is organized as follows: section 2 discusses the main sources we have used to identify

interesting properties, and the metainformation which describes each of them. Section 3 deals with the design of the PVF, while section 4 shows how it is applied to statecharts. In section 5, we discuss how some popular CASE tools support the creation of correct models, and section 6 ends with some conclusions and plans for future work.

2. PROPERTY SOURCES AND STRUCTURE

The first step towards the design of the framework consisted of identifying relevant properties for UML models, as well as identifying what metainformation is needed to describe each property.

2.1 Sources

The main source for identifying properties regarding UML models is, not surprisingly, the UML specification [1], where information is highly structured according to every diagram type. Within this specification, properties can be categorized as follows:

a) syntactic properties, described in sections 2 (subsections *.2 Abstract Syntax) and 3 (UML Notation Guide) of [1]. These describe how to create syntactically correct models.
b) semantic properties, described in section 2 of [1], including those contained in subsections *.3 (Well-Formedness Rules) and *.4 (Detailed Semantics). These include both static and dynamic semantics properties.

For particular aspects of UML where the official specification is not enough clear, and even ambiguous, it is practical to use alternative sources of information for identifying properties. Some well known UML books [21, 22, 23, 24] are of great help for this task. In addition, several papers related to metrics for UML models can be very useful too [29, 30, 31, 32, 33].

The matter of deciding "the right semantics" for a given property, regarding a particular kind of UML diagram or model is a tricky issue. UML has been criticized since its appearance (some of the criticism has been accepted by the authors themselves), mainly due to the ambiguity and the lack of a truly formal definition of its semantics [27]. The UML static semantics are described by the semi-formal constraint language OCL (Object Constraint Language) [1], and the UML dynamic semantics is expressed in natural language. The lack of a formal definition of the UML semantics gives rise to different interpretations between members of the development team and the users. This situation hinders the rigorous statement or the precise proving of properties [7] related to the models constructed using this language. In addition, these problems can lead to misunderstandings and errors along the software development process (from UML conceptual models to UML implementation or deployment models), thus considerably increasing the cost of the projects.

For these reasons, the metainformation template associated to each property (see next section) includes information fields regarding the possibility of a formal description of the property along with its source and, even more important, this allows that if a specific verifier uses a particular semantics intended by the modeler, it can be stated precisely in any available formalism.

Currently, a number of properties which affect the quality of a UML statechart have been collected (some of them have been adapted from Harel statecharts, where UML is ambiguous or not clear enough [27]), and a verifier has been developed. As it is reasonable to expect that new properties will be defined in the future, the PVF offers a resilient design which allows the integration of new properties as long as they appear.

The analysis has centered around UML statecharts, but it is being currently extended to other diagram types. From this study, several properties were identified, many of them regarding the structure of models. The experience of our research group relating to the development and study of UML models was also important, for it lead us to identify some interesting properties [7].

Another useful source for interesting statechart properties have been some tools devised to verify specific constraints and requirements on statecharts, such as Statemate Magnum [13] and the statechart-related part of general purpose UML tools (see section 5).

2.2 Property metainformation

In order to describe each property, a *property template* has been developed. This defines the relevant metainformation structure needed for each property. The set of properties is constructed by filling the template for each new property which is inserted into the framework. Not all fields are required for all properties. New metainformation fields can be added as they become necessary, provided that the corresponding verifier can deal with this new pieces of information for the properties considered.

The metainformation for each property contains a numeric code for rapid reference, and it has also a name and a natural language description, so that the user can get an idea of the purpose of the property. In addition to this, the following information is included:

?? **Semantics**. It can be dynamic or static. Some properties are related to the model structure (such as "absence of *livelocks*" in statecharts), and they belong to static semantics[1]. On the other hand, dynamic

[1] In the sense that this kind of properties can be tested on the model structure, not necessarily on the execution of the model

properties cannot be tested until the model is *running*, and thus they correspond to dynamic semantics. An example of dynamic properties are those regarding cardinality constraints in class diagrams.

?? **Priority**. Some properties can be identified as required, so that its absence from the model is considered an error (for example, syntactic errors such as a transition with no target state). Some others can be marked as desirable, but its absence does not mean an error in the specification, and it is signalled as a warning (for example, the existence of a non-final state which only has incoming transitions).

?? **Involved elements**. This identifies to what model elements the property can be applied. For example, some properties apply to classes, some other to transitions in statecharts, and some affect more than one element. Lastly, some properties involve the model as a whole.

?? **Formal expression**. Some properties can be stated in natural language, and others can be declared in a formal language. We can find several examples of these in the UML specification, where many properties include a OCL expression. We could also have properties defined in some other formal language different from OCL. This field is optional.

?? **References**. These link the reader to the documents from which the information concerning the property was extracted or is extended.

?? **Verifiable**. Yes / not. This field identifies properties which are actually implemented.

?? **Type**. Contains the name of the class which actually implements the appropriate test method for this property, so that the verifier will know what particular class it must instantiate when creating the corresponding Property instance.

?? **Category**. This identifies to which category, according the corresponding taxonomy used, this property belongs. This aspect is described in detail in Section 4.

?? **Arguments**. Some properties can have defined parameters needed to apply the tests. For example, we can define a property to test whether a given state in a statechart can be reached from some other state. Both state names are parameters to the property "Reachability", as well as the statemachine name.

As an example, figure 1 shows the template corresponding to the property which checks whether in a model a final state has outgoing transitions (which is an error). This property has no arguments.

COD	P1.1.1
PROPERTY	Final state with no outgoing transitions
DESCRIPTION	A final state cannot have any outgoing transitions
SEMANTICS	static
PRIORITY	error
ELEMENT	FinalState
OCL	self.outgoing->size = 0
REF	UML v1.4 spec. section 2.12.3.2 pag. 213.
VERIFIABLE	yes
TYPE	PropTransition
CATEGORY	Absence

Fig. 1. Metainformation template for the property "Final state with no outgoing transitions"

3. THE PROPERTY VERIFICATION FRAMEWORK

The PVF is composed of the following main components: 1) a Property Manager, which is the responsible for retrieving the information about each property and showing it to the user; 2) a Verifier, for each type of diagram; 3) the set of particular properties for each model element (implemented as Java classes), and 4) the data which fills the metainformation template for each property to be checked.

The PVF construction was addressed following a process model based on that proposed by Larman [5], with two main stages: requirements modeling (use cases and conceptual model) and analysis and design model. First of all, a Software Requirement Specification (SRS) was written, following the IEEE Std 830-1998 [4]. The specification sets the requirements for a property verification manager for UML v1.4 models to be integrated in RIVIERA, so that developers can build on it to further analysis and design.

3.1 The Property Manager

The PM main window is shown in figure 10. The GUIVerifier class (figure 2) serves as a graphical user interface for the available verifiers. This graphical interface allows the user browsing the properties, and shows relevant information about properties when these are selected. Several tabbed pages allow navigating the properties by diagram type.

Each property has a checkbox under the *Verify* column (figure 10). That way, the user can select individual properties to be tested in the model (or he can use the *Select all* option at the bottom). Some properties, (such as local achievability of states) require the specification of some arguments (source and target states, for example), which must be provided at the moment of selecting the property *Verify* checkbox. The collaboration diagram in figure 2 shows how this is done for each property.

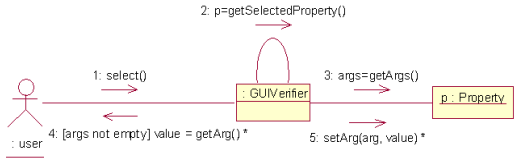

Fig. 2. Selecting properties to test and specification of arguments

The reason for allowing individual selection of properties is due to the fact that sometimes it is not desirable to test the model against all properties. First, if the number of implemented properties is high, this could be a great time-consuming task. A UML modeller perhaps would want to test all properties for the first time, but after subsequent modifications of the model, he probably is interested in testing only those properties which failed initially. Second, maybe that the modeller would want to create a model which explicitly violates some property, but for some reason the model must remain that way. If the broken property were tested unconditionally every time, messages saying that the model does not conform to that property would annoy the modeller.

After having selected some properties, and introduced its corresponding arguments, the user can click the Check button, what starts the checking process. As this process progresses, messages and verification results are written at the bottom text area. These can be saved to a text file for later retrieval.

3.2 Verifiers

Each diagram type will have an associated Verifier. This design is resilient enough to progressively add the ability to check different property sets. A hierarchy has been designed which includes one verifier for each diagram type contained in the model. This hierarchy is shown in figure 3, as well as the relationships between the verifiers and the PM (GUIVerifier class).

Each verifier uses a persistence manager to retrieve the set of all properties. This persistence manager is the responsible for obtaining all the information, including parameters, from the persistence mechanism (this can be a text file, a database or any other). By using this design, the information about each property is not mixed with the actual code. In addition, this avoids coupling between the verifier and the persistent storage mechanism used for the properties.

The verifiers access all information about the model through the IModelInterface, as described in section 3.5. Each verifier can use different methods to test whether the model conforms to the property or constraint. Often, this is done by implementing the test method into a Java function, but there are other possibilities, such as invoking external tools (e. g. theorem provers). In this case, a translator is needed which must translate the UML model to a suitable notation which can be managed by the external tool. RIVIERA can test some properties by using MAUDE [18] as a theorem prover, which can read algebraic terms that represent UML models and diagrams thanks to a formalization of the UML metamodel in the MAUDE algebraic specification language [34].

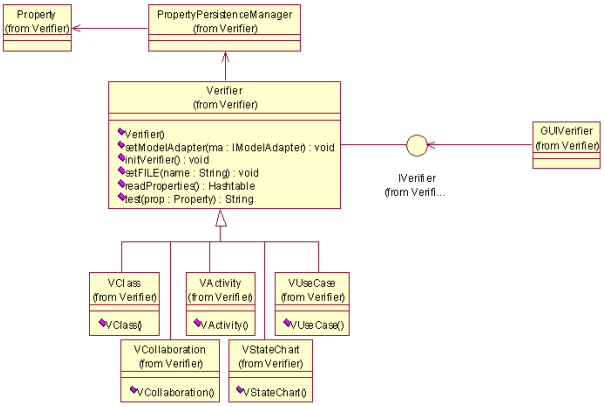

Fig. 3. Global structure of the Verification Framework

3.3 Property Hierarchy

Property structure has been designed by creating a generic hierarchy for all UML diagrams. This hierarchy will be extended as long as new properties are incorporated. Properties are defined by a generic Property class, which acts as the root for a property hierarchy. This class defines a generic test method, which should be overridden in subclasses. A subclass for each type of UML diagram has been defined, as shown in figure 4.

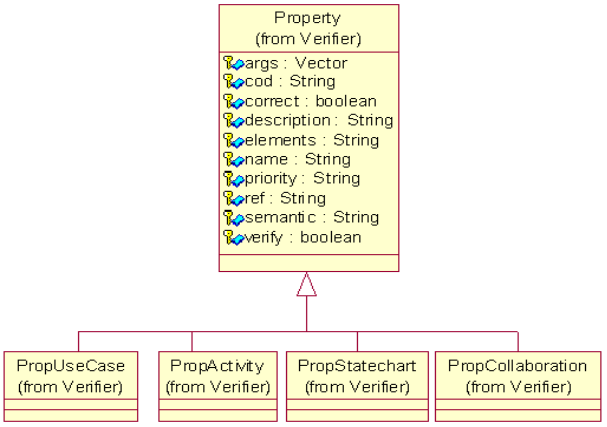

Fig. 4. Property hierarchy

Next, each particular property extends the property for each diagram type and redefines the *test* method. So, we leave the common code in the upper classes in the hierarchy, while each subclass redefines only the changing code. Properties are grouped so that the test method in each class is as simple as possible, paving the way for developers that want to add new properties. For example, the *test(IModelAdapter ma)* method, defined in

class *Property*, is redefined in class *PropStatechart*. Within this class, the method traverses all state machines, and for each of them, it invokes the method *test(String machine, IModelAdapter ma)*. This second *test* method is overridden in every child class, and it is the responsible for testing that the property is satisfied within the model. The rationale behind this is to apply the *Template Method* design pattern [6].

The test method can be implemented in many ways. Some properties could be easily tested executing a simple algorithm, which can be implemented in Java, for example. Others can require the use of external tools, such as a theorem prover, and the translation of the property into the external tool language (such as Maude).

By using arguments for properties, as described in Section 2, the same Property subclass can be used to test for two or more different properties on the model. For example, there exists a class that counts the number of incoming transitions into a state in a statechart, and generates a warning when this number is greater than or less than any given number. We can use this class to test both the property which states that "An initial state cannot have any incoming transitions", and the property which mandates that "A final state should have at least one incoming transition". In the first case, three arguments should indicate that we apply it to "Initial states", condition is "less than" and value is "0". In the second case, the arguments would be "Final states", "greater than" and "0".

Properties which are relevant to more than one diagram type, such as assuring that all classes and messages included in a sequence diagram are defined in the corresponding class diagram can be dealt with in two ways. The first option is to assign it to the most relevant diagram type to which belongs. In the example given, this property could appear under the sequence diagram property hierarchy, or under the class diagram property hierarchy. The alternative option is to define a different abstract class which is the root for those *interdiagram* or global properties. Currently, we choose the first option.

3.4 The Verification Process

Once the user has selected which individual properties must be checked (or the whole set of them), and after pressing the Check button, the following sequence of actions is launched (see figure 5):

The check message is sent to the property manager (step 1)

The property manager invokes in turn the *test* method for the corresponding verifier (step 2). In turn, this method obtains the reference to the model adapter (step 3), which will provide all the information about the model

The verifier iterates through the set of concrete properties, invoking the corresponding test method for each of them (step 4), and collecting all the responses as a unique message string, which will be returned to the property manager (GUIVerifier)

The property manager writes verification results and messages to the main text box (step 5)

The complete sequence of actions can be seen in figure 5.

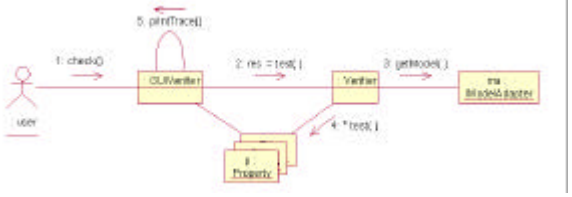

Fig. 5. Collaboration diagram for the *testProp* operation

3.5 Integrating the PVF into a CASE tool

The PVF consists of a unique Java package, which includes all classes related to property verification, ranging from the graphical user interface (GUIVerifier, the PM) to the individual class for each property. The only requirement for the host application tool where the PVF is going to be integrated is that it must provide an IModelAdapter interface, which allows the PVF access all information about the model. This interface must provide services to retrieve the set of all classes in the model, the features of each class (attributes and methods), states and transitions in statemachines, and so on. Including new properties and verifiers is fully restricted to the boundaries of the Verifier package, without affecting the rest of the modules.

As we mentioned in the Introduction section, we have integrated the PVF into a UML CASE tool developed by the authors, RIVIERA [12], which is mainly intended to support UML model V&V, simulation and transformation.

RIVIERA code is structured in several packages. One of them contains all classes related to the graphical user interface; other contains classes related to the internal representation of the model, and so on. Information about the UML model is maintained by using NSUML [9]. This is a freely available implementation of the UML 1.4 metamodel in Java, which includes services to read and write UML models by using the XMI format.

In addition, RIVIERA includes an implementation of the IModelAdapter interface, which provides all the information about the structure of the model in an abstract way, including information about classes, links, states and substates, transitions and so on. This avoids highly coupling between this particular implementation of the UML metamodel (NSUML) and the rest of RIVIERA components, by using the Façade design pattern. That way, future evolution of RIVIERA is not tied to the NSUML package. If some reason forces to remove the NSUML library from RIVIERA, this change can be addressed by only rewriting a new class which implements the IModelAdapter interface.

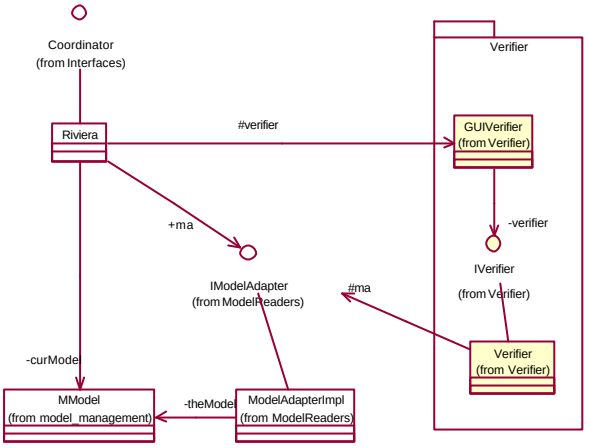

Fig. 6. Integration of the property manager within RIVIERA

Similarly, for the PVF to be included in other existing tools, the unique requirement is that the tool designers must write a class which implements the IModelAdapter interface. This should not be too difficult, because the PM and the Verifiers need few services to access model elements. Additionally, one must implement some way to invoke the PM main window, for example, by adding a new menu item in the application.

Figure 6 shows the relationships between the verifiers, the PM and the rest of the components of RIVIERA.

3.6 Adding new properties

The number of properties to be satisfied by a particular model can be enormous. Therefore, an incremental procedure to include –or modify– properties, both in code as well as the related metainformation, is needed.
Each of the verifiers gets the information about properties from a persistent storage (currently, one text file with .prop extension for each diagram type, as shown in table 1).

Verifier	Property file
VStatecharts	state.prop
Vclass	class.prop
VCollaboration	collaboration.prop
VActivity	activity.prop
VUseCase	usecase.prop

Table 1. File names containing property information

To add new properties, one must insert the corresponding information in the related file, including name, description, arguments, and so on. If there is no property in the hierarchy whose test method is able to make the test, one also must create the required Property subclass, and implement a *test* method for this class. Thus, one should write the code necessary to check the model against this property, returning a boolean value which tells whether the property is satisfied or not. Newly added property classes must inherit from the Property abstract class. A property hierarchy currently exists for statecharts properties, as shown in figure 8, but others will be gradually incorporated into the PVF.

This hierarchical design provides an efficient way of integrating new properties. If we had only a Property class and all the specific properties were instances of it, we should check a big conditional statement (such as a switch or case) for implementing the test method, what would difficult the reading and comprehension of the code.

4. FRAMEWORK INSTANTIATION FOR STATECHARTS

In order to show how the PVF can be applied to a specific set of properties, this section discusses how it has been instantiated for the verification of properties regarding UML statecharts.

4.1 Statechart Property Taxonomy

After having identified a set of relevant properties for statecharts, these were classified to allow a global understanding and handling. Moreover, this provides an additional criterion for the visualization in the PM. The classification scheme by *Dwyer, Avrunin y Corbett* [3] (figure 7) served us as starting point for classifying the identified statechart properties. This scheme groups all properties that can be specified in a particular system according to the type of behaviours they describe, especially those related to the occurrence and absence of a particular sequence of states or events during the execution of the state machine. Though this arrangement is oriented towards dynamic properties, and most of our properties are related with the static structure of the state machine, Dwyers' patterns provide a very useful conceptual framework and they have been easily adapted to establish a systematic arrangement of the properties. Though some of the properties identified for statecharts do not perfectly fit within any of the patterns, we think that it is more useful to have an imperfect classification framework that not having it at all.

This scheme establishes some generic specification patterns of occurrence and order. A specification pattern is a generalized description of a requirement which occurs frequently on the permissible sequence of states / events in a finite state model of a system.

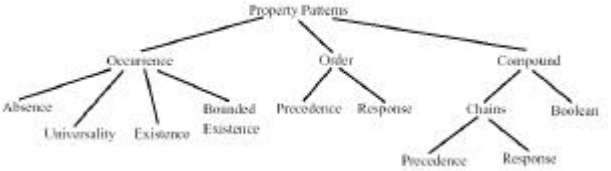

Fig. 7. Classification of statechart properties by Dweyer et al.

Occurrence patterns describe properties related to the occurrence of a given state or event in a particular scope during system execution (for example, we used occurrence patterns to specify the property that at least

one incoming transition to the final state of a statechart must exist). **Order patterns** concern properties which require that certain states or event must occur previous to others during system execution. Lastly, **compound patterns** are combination and generalization of the two above. This classification scheme is reflected in the Category attribute for each property. Table 2 shows the name and description of the occurrence and order patterns used.

Pattern	Description	Example properties
Occurrence		
Absence (Never)	A given state/event does not occur within a particular scope	1.1.1 A final state cannot have outgoing transitions 1.1.3 Top state cannot be the source for a transition
Existence (Eventuality)	A given state/event must occur within a particular scope	1.2.2 A transition must have target state 1.2.3 Composite states must have a default substate
Bounded existence	A given state/event must occur k times within a particular scope	1.3.1 A composite state must have at most one initial state 1.3.4 A composite concurrent state must have at least two composite substates
Universality (Always)	All states in a particular scope must conform to the property	1.4.1 A concurrent state can have only composite substates
		1.4.3 A top state is always composite
Order		
Order-Precedence	Within a scope, a given state / event must be preceded by another state/event	2.1.1 Two sequential substates of a composite state cannot be active at the same time
Order-Response	A certain state/event must be always followed by a given state/event within a defined portion of a system execution	Properties will be defined and added when we include simulation capabilities for statecharts into RIVIERA

Table 2. Property classification according to Dwyer et al. pattern hierarchy

It is important to note the difference between this property taxonomy (at requirements level) and the property hierarchy discussed in the next section (at design level).

4.2 Extending the Property Hierarchy for Statecharts

As shown in figure 8, the PropStatechart class is extended by the corresponding subclasses. These subclasses are defined according to the model elements they check. The PropPseudostate class is the responsible for checking properties related to pseudostates, such as property 1.2.9, which states that "Pseudostates different from initial should have at least one incoming transition". The PropTransition class checks those properties related to transitions, such as property 1.1.5 "A fork segment (outgoing transition from a fork pseudostate) should not have guards or triggers". The PropCompositeState class verifies properties specific to composite states, such as property 1.2.3 "A composite-or state should have a default entry". The class PropState controls properties related to *regular* states, as opposed to pseudostates, such as property 1.2.8 "Reachability (all states have at least one incoming transition, in order to be reachable)". Properties where state machines as a whole or top states are involved are managed by the PropStateMachine subclass, e. g. property 1.4.3 "The top state must be composite". Finally, the PropSubMachine class tests those properties which affect to submachine states, such as property 1.1.4 "Submachine states cannot be concurrent".

Note that some of the properties fit well in more than one category, and could be verified by more than one class. In these cases, a design decision must be made to assign this responsibility to the subclass that seems more appropriate.

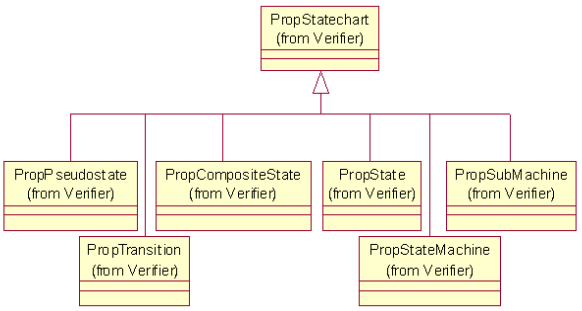

Fig. 8. Statechart property subtree

4.3 Use of the Verifier

Next we will see how a UML modeller can use the verification functionality through an example. The example will be based on the telephone statechart shown in figure 9, adapted from [1]. In this figure, we can see that there are two states, Pinned and Talking, which are connected by two transitions. One problem with this model is that once we enter any of both states, there is no mean to get out of them, what is sometimes referred to as *livelock*. This is not a syntactic error but it is a situation that should be warned to the user, because it represents a potential failure of the system. The verifier should detect this condition and propose a possible solution to it.

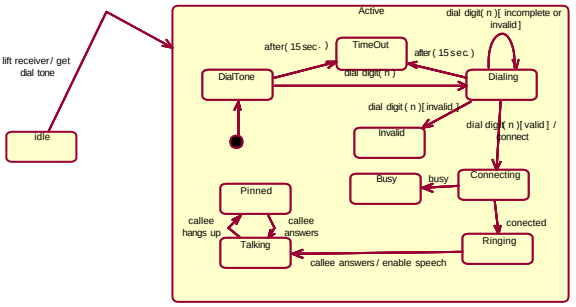

Fig. 9. Sample statechart diagram for a telephone system

Figure 10 shows the PM, containing specific statechart properties. Among these, we can see that the Livelocks property is selected for verification. At the top right text area, a complete description of the property is displayed, as well as part of the metainformation about it.

Once selected the desired properties, the user presses the Check button, what starts the verification process. Results can be seen in the lower text area in figure 10. Property "Livelocks" has been labeled with "Warning", because the model contains a livelock, that is , a loop between the Talking and Pinned states (actually, the verifier detects two livelocks, once starting from the Talking state, and the other starting from the Pinned state, but both are the two sides of the same problem).

The algorithm that carries out the search for livelocks in the model is coded in Java, within class PropTransition (it is a property related both with transitions and states, though it could have been implemented within class PropState as well). Some properties could be tested by translating the model elements and the property to a particular formal language and invoking a external tool on this transformation, as discussed by Fernández and Toval in [7] for the so-called orthogonality property.

In figure 10, we can also see that the property "Transitions from initial or historic PseudoStates should not ascend" is marked as OK, what means that it is satisfied within the model.

Each property selected is marked as OK, WARNING or ERROR, so indicating whether the property holds or not in the model. It also shows a description of the reason for the error or warning, and some advice to correct it. According to the incremental procedure used for the PVF construction, some properties have not been yet implemented, what is reported when these are selected to be checked.

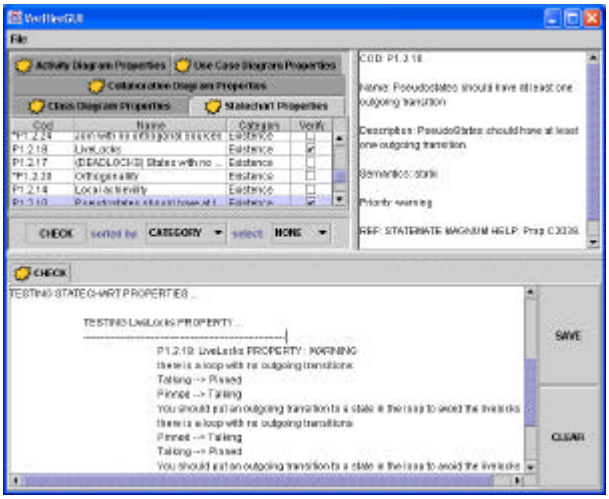

Fig. 10. Property manager main window

5. RELATED TOOLS

The current number of UML CASE tools is huge [16]. Many of these tools offer different levels of support for the creation, edition and management of models, but only a few of them offer a solid support for precise verification of the models under construction. Some of the most popular UML CASE tools have been reviewed to identify what support they offer for creating syntactically and semantically correct models. This revision served us as an additional source for extracting some interesting properties.

Rational Rose [10] has severe limitations regarding the creation of correct UML models. For instance, Rose prevents from some actions which can produce syntactically incorrect statecharts, such as inserting a transition from a final state or creating two initial states in the same statechart. But on the other hand, some actions, such as creating transitions from an initial state with many different stereotypes or guards are not restricted (these are illegal, according to UML specification), and so erroneous models can be easily generated. In addition, it offers no means to specify concurrent states, so there are a lot of properties that cannot be tested on these particular type of statecharts. Rose can export models in XMI format, what allows reading those models in RIVIERA.

Statemate Magnum [13], from I-Logix, is a powerful tool for the modelling of reactive systems, as well as testing completion and correction properties on statecharts. Models can be simulated before being finished, and code and documentation can be generated from the models. Statemate allows the specification of concurrent states. However, it works with Harel statecharts, which are not identical to UML statecharts (though UML statecharts derive from Harel's). Some examples of properties which are checked by Statemate are the existence of or-substates (sequential) without a default entrance, or the existence of compound transitions with no orthogonal sources. On the other hand, Statemate allows triggers in outgoing transitions from junction vertexes, as well as composite states with more than one deep history state. This is in contradiction with the constraints defined in the UML specification (one could argue that Statemate statecharts have not identical semantics than UML statecharts, but in these examples semantics is similar). Statemate cannot export models in XMI format.

ArgoUML [17] is a very popular UML modelling tool, and includes a very interesting system of critics (suggestions) which examines the model in the background, as it is being created, and generates a set of to-do tasks in order to improve the design of the model according particular modelling principles. For example, it suggest adding names to unnamed elements, or adding one initial state to a statechart without it. However, some incorrect actions are allowed in Argo which can produce incorrect models, such as adding two initial states to a statechart or writing a guard expression in an outgoing transition from an initial state (this is forbidden in the UML specification [1]). Argo also saves models using XMI format, and these can be read in RIVIERA.

Rhapsody [14] was also developed by I-Logix, and it allows the creation of UML models. Unlike Rose and

ArgoUML, Rhapsody allows the definition of concurrent regions. In addition, it can check a lot of properties, such as creating isolated states, or creating circles (livelocks). On the other hand, some interesting properties are not verified in Rhapsody. For example, one can create composite states containing more than one deep history state, you cannot specify joint vertexes which have more than one incoming transition, or you can create history states without siblings (all this actions are incorrect according the UML specification [1]). In addition, generated XMI code is not fully compatible with other tools such as Rational Rose and Together, and we have not been able to open models created with Rhapsody in RIVIERA.

Together/J [15] is a powerful development environment which supports the creation of UML models and generation of code, providing integrated reverse engineering facilities. Thus, one can add a class to a diagram and see it instantly in the corresponding code, and one can add an attribute to a class in the code window and see the attribute appear immediately in the diagram window. Created models can be analyzed by using the metrics components included in Together. However, one cannot create concurrent states, and statecharts cannot be attached to a class. In addition, some constraints are not satisfied in the creation of statecharts. For example, one can define more than one initial states for a particular composite state. Or one can add triggers or guards to a fork vertex outgoing transitions or one can add guards or triggers to a join incoming transitions (both actions are forbidden by the UML specification); or you can put more than one initial state in a statechart. Together can export models to XMI format.

The PhD thesis by Porres [8] deals with the specification and analysis of software behavior using UML. Use cases and statecharts are specially taken into account. Model checking is applied to statecharts by translating them to the PROMELA specification language, and using the SPIN model checker. The vUML tool interfaces between the user and the SPIN checker, and it is the responsible of translating the UML model to the PROMELA system. It can also translate SPIN results to UML sequence diagrams, which can serve as counterexamples when an error is found.

The described PVF has a highly modular design which enables to be included in existing CASE tools implemented in the Java programming language. In addition, it is flexible, so that new properties can be added in the future in an easy and homogeneous way (by subclassing the appropriate property class and implementing a test method). When new algorithms are found, the test method of each property can be easily replaced by the new code.

In addition, the user of the PFV is not forced to work with all the existing properties. The PVF allows a high level of customization, and undesired properties can be checked-out so that they are not tested in the current model. Having implemented the PVF in Java permits that it be used in the most important platforms existing today.

6. CONCLUSIONS AND FUTURE WAYS

We have presented a Property Verification Framework which helps to detect errors in UML models, by using a set of Verifiers. The main goal is that the user can check his models against desired quality features. The verifiers use a predefined set of desirable properties which must hold for a model, and also give some suggestions for the modeller in order to change the model to conform to the selected properties.

The PVF has been instantiated to a set of interesting properties for UML statecharts. The definition for these properties have been extracted from several sources, and we have seen how some popular CASE tools sometimes fail to help the user to create models conforming to many of these properties and constraints (even though some are syntactical ones). Some properties have been identified but not actually implemented.

One of the major advantages of the PVF is that it has a modular design which allows that it can be easily incorporated into existing CASE tools which are implemented in the Java programming language. Currently, the PVF has been integrated into RIVIERA.
In addition, the framework can be extended to include new properties as long as these are identified. The developer who adds new properties only has to create the corresponding class and implement one *test* method in it.
By containing all information about properties in a separated persistent mechanism, internationalization an be easily provided so that descriptions and names of properties can be shown in different languages.

The PVF can and will be improved. Future work in this direction will include:

- ?? augment the set of properties that can be verified to include those for other UML elements, such as class and collaboration diagrams.
- ?? identify new properties for statecharts
- ?? implement properties that are not currently implemented
- ?? improve the user interface, so that each verifier can generate reports in HTML format, for example, what would allow us to insert hyperlinks to the documents where the properties are discussed and analyzed
- ?? apply the property verification to actual models which are being used in industrial projects

7. REFERENCES

[1] OMG, "Unified Modeling Language Specification version 1.4". www.omg.org/uml

[2] R Eshuis, R. Wieringa. Requirements-level semantics for UML Statecharts.University of Twente, Dept. Of Computer Science, The Netherlands. pp 121-140, Formal Methods for Open Object-Based Distributed Systems IV (FMOODS), 2000.

[3] M. B. Dwyer, G. S. Avrunin and J. C. Corbett. Property Specification Patterns for Finite-state Verification. 2nd Workshop on Formal Methods in Software Practice, March, 1998. http://www.cis.ksu.edu/santos/spec-patterns

[4] Institute of Electrical and Electronics Engineers, Inc. "IEEE Recommended Practice for Software Requirements Specifications" IEEE Std 830-1998. Template A.4. http://standards.ieee.org/catalog/olis/se.html

[5] C. Larman. Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design. Prentice Hall, 1998.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[7] J. L. Fernández, and A. Toval,, "Can Intuition Become Rigorous? Foundations for UML Model Verification Tools", *Proc. of the International Symposium on Software Reliability Engineering (ISSRE 2000), IEEE Press*, October 8-11, 2000.

[8] Porres, I. (2001). Modeling and Analyzing Software Behavior in UML. Ph. D. thesis. Department of Computer Science. Turku, Finland, Abo Akademi University.

[9] Novosoft metadata framework NSUML: nsuml.sourceforge.net

[10] Rational Rose UML CASE tool. www.rational.com

[11] Object by Design. UML Modeling Tools, www.objectsbydesign.com/tools/umltools_byPlatform.html, 2002.

[12] Sáez, J., A. Toval, et al. (2001). Tool Support for Transforming UML Models to a Formal Language. WTUML 01: International Workshop on Transformations in UML, ETAPS 2001 (European Joint Conferences on Theory and Practice of Software)., Genoa, Italy.

[13] Statemate Magnum CASE tool. http://www.ilogix.com/products/magnum/index.cfm

[14] Rhapsody 4.0 CASE tool. www.ilogix.com

[15] Together 6.0 CASE tool. www.togethersoft.com

[16] (2002). UML Modeling Tools, Object by Design. www.objectsbydesign.com/tools/umltools_byPlatform.html

[17] ArgoUML modeling tool. argouml.tigris.org

[18] Clavel, M., F. Durán, et al. (January 1999). Maude: Specification and Programming in Rewriting Logic, Computer Science Laboratory. SRI International.

[19] D. Lending, N. L. Chervany. "The Use of CASE Tools". Proceedings of the 1998 conference on Computer personnel research, 1998, pp. 49-58

[20] M. E. McMurtrey, J. T. C. Teng, V. Grover and H. V. Kher. "Current utilization of CASE technology: lessons from the field". Industrial Management & Data Systems, 100 (1), 2000, pp. 22-30.

[21] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. 1999: Addison-Wesley Longman, Inc.

[22] Fowler, M., *UML Distilled - 2nd ed.*, ed. A.W. Longman. 2000.

[23] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. 1999: Addison Wesley Longman Inc.

[24] Rumbaugh, J., I. Jacobson, et al. (1998). The Unified Modeling Language Reference Manual, Addison-Wesley Pub Co.

[25] Frankel, D. (2003). Model Dirven Architecture: Applying MDA to Entreprise Computing, John Wiley & Sons.

[26] Mellor, S. J. and M. J. Balcer (2002). Executable UML: A Foundation for Model Driven Architecture, Addison Wesley Professional.

[27] Reggio, G. and R. Wieringa (1999). Thirty one Problems in the Semantics of UML 1.3 Dynamics. Workshop "Rigorous Modeling and Analysis of the UML Challenges and Limitations", OOPSLA'99, Denver, Colorado.

[29] Poels, G. and G. Dedene (1997). "Comments on "Property-based Software engineering measurement": Refining the additivity properties." IEEE Transactions on Software Engineering 23(3): 190-195.

[30] Brito e Abreu F. and Melo W. (1996). *Evaluating the Impact of Object-Oriented Design on Software Quality*. Proceedings of 3rd International Metric Symposium. [32] Chidamber S. and Kemerer C. (1994). *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 20(6), 476-493.

[31] Harrison R., Counsell S. and Nithi R. (1998). *An Evaluation of the MOOD set of Object-Oriented Software Metrics*. IEEE Transactions on Software Engineering, 24(6), 491-496.

[32] Lorenz M. and Kidd J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, Englewood Cliffs, New Jersey.

[33] Marchesi M. (1998). *OOA Metrics for the Unified Modeling Language*. 2nd Euromicro Conference on Software Maintenance and Reengineering, 67-73.

[34] Toval, A. and J. L. Fernández Alemán (2001). Improving System Reliability via Rigorous Software Modeling: The UML Case. IEEE Aerospace Conference (track 10: Software and Computing), Montana, USA, IEEE Computer Society.