A Framework for the Verification of UML Models. Examples using Petri Nets

Esther Guerra and Juan de Lara *

Escuela Politécnica Superior, Ingeniería Informática
Universidad Autónoma de Madrid
Esther.Guerra_Sanchez@ii.uam.es, Juan.Lara@ii.uam.es

Abstract. This paper presents a framework for the verification of UML models. Our approach is to build meta-models for the different UML diagrams and translate them into formalisms (whose syntax is also specified with a meta-model) in which properties of interest can be proved. The translation (denotational semantics) as well as the formalisms operational semantics are formally described by means of graph grammars. We show the implementation of these concepts in the Multi-Paradigm tool AToM^3 together with an example in which we translate a UML design (composed of Statecharts, Class and Sequence diagrams) into Petri nets for subsequent verification using model checking.

Keywords: Graph Rewriting, Meta-Modelling, UML, Petri Nets, Model Checking.

1 Introduction

UML is the *de-facto* standard for the analysis and design of object-oriented systems. It offers a number of diagrams that can be used for the modelling of different aspects of the application. Ever more frequently, these diagrams are not a mere documentation of the system's features, but are used for code generation, simulation, or verification. The current trend in hiding the verification process behind well-known notations – such as UML – has the potential to reduce some of the discouraging aspects of classical formal methods. Although they offer significant benefits in terms of improved quality, some of the reasons for which they are not broadly used are the need of expert personnel in a certain formal method. This expert knowledge is seldom found between the average software engineers.

In this paper, we propose hiding the verification process by letting the developers specify the system in a well-known modelling language (UML) and then automatically translating these models into a semantic domain for further analysis. Once this analysis is performed, feedback can be given back to the developers. The process is implemented in an open, formal framework: both source and target formalisms are meta-modelled, and the transformation between them is formally specified using graph grammars [10]. This is possible as

* Partially sponsored by the Spanish Ministry of Science and Technology (TIC2002-01948)

(meta-)*models can be represented as attributed typed graphs, and thus can be manipulated using graph grammars.

AToM^3 [1] [4] is a tool for Multi-Paradigm modelling which we have developed in collaboration with the MSDL Lab at McGill University. This tool allows the user defining formalisms by building models of its (possibly graphical) syntax by means of meta-modelling. In AToM^3 models are stored as graphs and their manipulation is expressed by means of models in the graph grammar formalism [10]. Typical manipulations include defining operational semantics (specifying a simulator), denotational semantics (translating the model into another formalism preserving behaviour, and code generation for another tool) and optimising the model (for example, reducing its complexity). We have used the tool to define some of the diagrams of the UML standard, and defined graph grammars to perform the translation of these diagrams into Petri nets. Once the diagrams have been translated into a single Petri net, we can use simulation, or analysis techniques (such as the ones that use the reachability graph) to check properties of the whole system (horizontal coherence between the UML diagrams, deadlock, state reachability, etc.) In particular we have implemented Model-Checking techniques [3] that can be used on the resulting reachability graph.

2 A Framework for the Verification of Software Models

The approach we propose in this paper is an adaptation of the ideas of *Multi-Paradigm* modelling [12] (for physical systems) to the verification of software models. In Multi-Paradigm modelling each component of the system can be described using the most appropriate formalism. For the processing of such a system (understanding it as a whole), one has to translate each component into a common formalism. This translation must preserve the properties under investigation. The target formalism must provide the necessary tools to answer the question we are interested in the original system.

In an analogy to the formalism transformation graph for physical systems [12], one can imagine a *"formalism space"* whose elements are the different possible formalisms for software systems (a very small fragment of it is depicted in Figure 1). Some of the formalisms are expressive, high-level, and appropriate for modelling. Some others are lower-level, but good for analysis (due to existing mathematical analysis methods and tools) or simulation. Behaviour preserving transformations can be found between some of these formalisms (shown as thin arrows). Sometimes when transforming a model, information can be lost (abstracted), but this is acceptable if we do not need this information for the question we are trying to answer. In our framework, we formally specify the formalisms by means of meta-modelling.

The thicker, dotted vertical arrows in the figure denote a transformation of the model to the *"traces"* formalism (its execution). Whereas the *transformation arrows* can be thought as expressing the *denotational semantics* of a formalism in terms of another, the *execution* transformations correspond to the *operational*

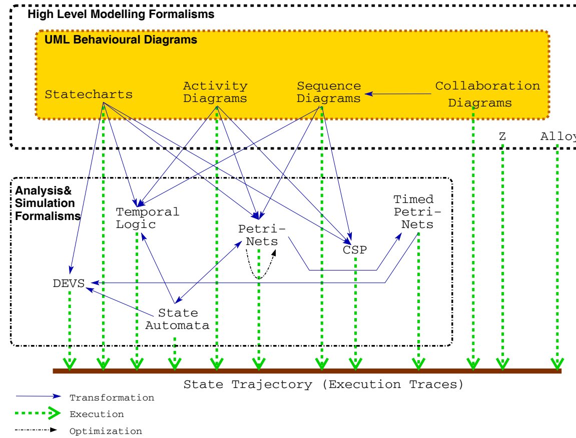

Fig. 1. Formalism Transformation Graph for Software Systems.

semantics of the formalism. The dark arrow that departs and arrives to the Petri net formalism means the existence of an "optimising" transformation. For example, we have implemented the complexity reduction transformations proposed in [8], which reduce the size of the Petri net model, preserving liveness, safeness and boundedness. We usually apply this transformation before calculating the reachability graph ro reduce the search space. In our framework, we formally specify the three types of transformations (optimising, operational and denotational semantics) as graph grammars.

When describing software systems (in UML for example), one usually has a number of views describing different aspects of the system, which may overlap. The *"connection"* of the different views of the system becomes a non-trivial problem: some parts of the system may have been specified several times (in several ways, in different views), while others may have not been specified at all. In any case, we have to verify that the union of all the views results in a consistent description of the system. We tackle these problems in the paper.

3 Meta-Modelling Sequence Diagrams

This section shows the definition of a meta-model for sequence diagrams using AToM^3 [1]. This tool allows defining formalisms using Entity-Relationship or UML class diagrams. In both formalisms we can use constraints (OCL or Python) to deal with dynamic restrictions ("well-formedness rules"). To define a formalism, the user has to include *abstract syntax* (entities, attributes, relationships and constraints) as well as *graphical information*. The latter includes the appearance of entities and relationships and the possible graphical constraints, that is, *concrete syntax*. Once the formalism is defined, AToM^3 generates some (Python) files that can be loaded again on top of the tool (bootstrapping) to

change its behaviour, in such a way that it accepts valid models according to the formalism definition. The advantage of this approach is that it dramatically enhances the productivity in creating domain-specific tools.

Sequence diagrams represent the flow of messages between some of the objects of the system. Objects are usually arranged horizontally, whereas the time flows in the vertical direction. Figure 2 shows to the left the meta-model we have defined using AToM^3. The iconic buttons to the left (*"Entity"* and *"Rel"*) are used to create the meta-model in the canvas, and are specific to the Entity-Relationship formalism. The user interface shows other buttons depending on the loaded formalism. This part of the user interface is indeed a model, which the user can change, and that is interpreted when the formalism is loaded.

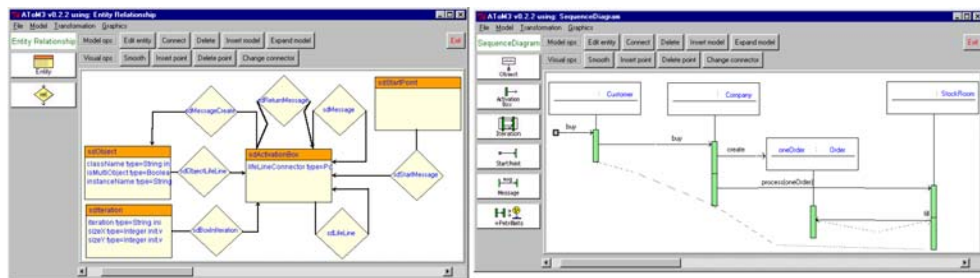

Fig. 2. A Meta-Model for Sequence Diagrams (left) and the Generated Tool (right).

The entities included in the meta-model are Objects (entity *sdObject*), Activation Boxes (*sdActivationBox* in the figure), Iterations (*sdIteration* in the figure, these allow the iterative execution of several messages), and *sdStartPoint* (representing the starting point of the diagram). Messages between objects have been implemented by means of the *sdMessage* relationship. More details about this meta-model can be found at [2].

Figure 2 shows on its right a sequence diagram built with the previously described meta-model. The user interface buttons have changed (to the left) with respect to the previous formalism, and they allow the user to create the entities defined in the meta-model. Additionally, we have added a button that executes a graph grammar to convert the sequence diagram into a Petri net. Graph grammars and their use in AToM^3 are explained in the following section.

4 Graph Grammars

Graph grammars [10] are a generalization of Chomsky grammars for graphs. They are composed of rules; each having graphs in their left and right hand sides (LHS and RHS). Rules are evaluated against an input graph (called host graph). If a matching is found between the LHS of a rule and a zone in the host graph, then the rule can be applied and the matching subgraph of the host graph

is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to decide which rule should be checked next. In $AToM^3$ rules are ordered according to a user-assigned priority, and are checked from higher to lower priority. If a rule makes a match, after its application the system starts trying again the higher priority rule in the list. The execution ends when no rule can be applied.

On the one hand, graph grammars have some advantages over specifying the computation to be done in the graph using a regular programming language. Graph grammars are a natural, formal, visual, declarative and high-level representation of the computation. Computations thus become high-level models, expressed in the graph grammar formalism. The theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation. On the other hand, the use of graph grammars is constrained by efficiency as subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node and edge types and attributes can greatly reduce the search space. This is the case with the vast majority of formalisms.

In the following subsection, due to space constraints, we briefly sketch a transformation from sequence diagrams (whose meta-model was described in the previous section) into (black and white) Petri nets.

5 Transforming Sequence Diagrams into Petri Nets

The tool we obtained in section 3 by means of (static) Meta-Modelling only allows the user to create, load and save models, verifying that they are correct. In this section, we improve these capabilities by means of graph rewriting. In particular, we present a graph grammar that translates a sequence diagram into an equivalent Petri net for further analysis. During the graph grammar execution, the model is indeed a blend of sequence diagrams and Petri nets, but when the graph grammar execution finishes, the resulting model is entirely expressed in the Petri nets formalism. The main idea of the transformation is to replace each message between two activation boxes by three places, which represent the point in which the message has not been sent, the sending of the message, and the reception of the message and its subsequent processing.

Figure 3 shows one of the rules for transforming messages into places. Other rules to transform messages are related to the transformation of initial, creation and destroy messages, and they are very similar to the one we show. Due to space constraints we only show this rule, the interested reader can consult [2] for a complete description of the transformation, together with another grah grammar to transform Activity Diagrams into Petri nets. The rule is applied for each message of type *"expression"* in the diagram and replaces the message by three places connected by transitions. Node and connection attributes in the LHS must be provided with values that will be checked with the node and connection attributes of the host graph in the matching process. These attributes can be set

to *any* or have specific values. In Figure 3, the text shown as *"⟨ANY⟩"* specifies that any name of the message will make a match. Nodes and connections have numbers. If the number appears in both LHS and RHS it means that the node (or connection) is preserved if the rule is applied (although its attribute values may change, and this is specified in the RHS). If the number appears in the LHS but not in the RHS, then the node (or connection) is deleted if the rule is applied. Finally, nodes (or connections) that are newly created by the application of the rule only appear in the RHS, thus, its numbers do not appear in the LHS. In the RHS, we can specify a change in the value of the attributes of the nodes that also appear in the LHS, and certainly we have to specify the attribute values of the newly created nodes or connections. In $AToM^3$ we can either copy the value of the attributes of the LHS, specify a new value, or associate a piece of Python code to calculate the attribute value (possibly using other nodes' attributes).

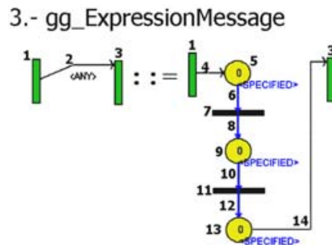

Fig. 3. One of the Rules to Convert Messages into Places.

The second block of rules transforms the activation boxes into Petri net transitions. When none of the previous rules can be applied, then the next step is to eliminate the *return messages* (which signal the invoking method the termination of the invoked operation), linking appropriatelly the Petri net transitions obtained so far. Finally, the graph grammar is completed with a couple of rules for removing all the remaining life lines and auxiliary transitions.

The application of this graph grammar to the example shown in Figure 2 results in a sequential connection of thirteen places terminated in a transition, in which the first place has a token, as there are no conditional messages or iterations. Next section shows how to link the Petri net model obtained from the transformation of a sequence diagram with the Petri net models obtained from the transformation of other UML diagrams.

6 An Example

As an example, we translate a toy UML design composed of Statecharts, Class and Sequence diagrams into Petri nets. We reproduce a very typical situation with UML modelling: some parts of the system are overspecified, while others are not specified enough. That is, some classes have an associated Statechart, while

others do not and in addition, some of the information in the Statechart (messages sent in response to other messages) are repeated in some of the sequence diagrams. The example is shown in Figure 4 and represents a *Sales Company* with a number of *Customers*, which can buy goods, resulting in the creation of an *Order*. This *Order* is sent to one of the *Stockrooms* for its processing. If there are enough goods in the stockroom, the *Order* can be served, and later the *Customer* has to pay for it. Note how some of the classes have an associated Statechart specifying its behaviour, while some others do not. The diagrams for this example should be complemented with the sequence diagram presented in Figure 2, which presents a scenario in which the customer buys an item, and the the stockroom fills the order.

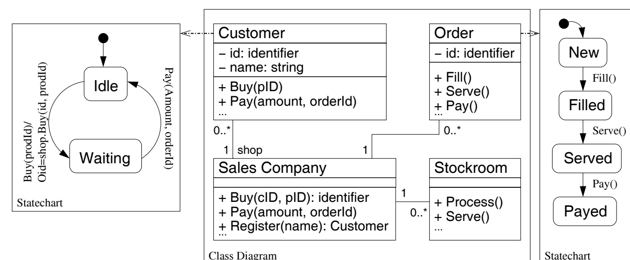

Fig. 4. A Part of the UML Design to be Verified.

The first step is to convert the Statecharts associated with each class into Petri nets. We have already shown how to automate this process using graph grammars in [5]. Each state in the Statechart is converted into a place. The methods defined in the associated class are also converted into places. These places are used as an *"interface"* for the Petri net. When an entity wants to make a method invocation, it puts a token in the corresponding place. The Statecharts of the different classes can be coupled if they send methods to each other. It may be possible for a certain class to not have an associated Statechart. In this case, we add a simple Petri net with a place for each method in the class and a single place representing the only object state. Thus, we are modelling the fact that we do not have any information about the class behaviour, and therefore we assume that the class accepts any method invocation at any moment. When more information becomes available (if the user specifies a Statechart for the class), a new refined Petri net for the behaviour can replace this simple net. The converted Statecharts are shown on top of Figure 5. The sub-model on the left represents the Statechart of the *Order* class; the sub-model on its right represents the Statechart for the *Customer*; the next one, the Statechart for the *Sales Company* and the right-most sub-model, the Statechart for the *Stockroom*. As the *Sales Company* and the *Stockroom* do not have an associated Statechart, they are given two simple sub-models with only one state.

The next step is to convert the sequence diagram and join the resulting model with the Petri net that we have obtained so far. The automation of this process was shown in section 5 (some rules for linking the Petri nets resulting from the

Statecharts and the sequence diagrams have been added). The resulting model is shown in the lower *"trail"* of transitions and places in Figure 5. The place named *"(Customer)proc.buy"* (in the lower- left corner) is the only one that can initiate the Petri net execution, and it corresponds with the initial message in the sequence diagram. That is, the net execution is driven by the scenario specified in the sequence diagram. Note how, if we have several sequence diagram, they should be verified one by one with the Statecharts using this process.

There are two ways of synchronizing the sub-model representing the sequence diagram and the sub-models representing the classes' Statecharts. The first case (*"under-specification"*) arises when the Statechart does not give all the details about sending methods in the transitions. In this case, the sequence diagram sub-model has to put a token in the corresponding interface place. We have four cases of this in Figure 5: the incoming arc to the *interface places "Buy"*, *"new"*, *"fill"* and *"Process"*. Other form of *"under-specification"* arises when there are method invocations in the Statecharts that are not reflected in the sequence diagrams. In this case, the simulation traces or the reachability graph of the resulting Petri net shows the user the methods that are really invoked.

The second situation (*"over-specification"*) arises when a method invocation has been specified both in the sequence diagram and in the Statecharts. We must ensure that the sub-model representing the sequence diagram and the sub-models representing the Statecharts are synchronized at that moment. This is performed by introducing an extra place connected to both the sequence diagram sub-model and to the Statechart sub-model. This occurs once in the example, as in the Statechart associated with the customer we have specified that message *"Buy"* (*salesc.Buy* place) must be sent to a *Sales Company* object when the *Customer* receives a *Buy* message, and this is also specified in the sequence diagram. In the Petri net this is specified by adding the place *"seqDiagram buy"* connected to both the sequence diagram and the *Sales Company* sub-model.

Once we have the complete Petri net, we can simplify it using the simplification grammar mentioned in section 2 and calculate the reachability graph (another formalism transformation). This transformation is only possible if the net is simple (in our case the reachability graph has 13 states), due to the state explosion problem. For more complex nets, other symbolic techniques should be used. Additionally, we have implemented an explicit model checker [3] which allows verifying properties of a model (in our case a reachability graph). The properties to be checked are expressed using Computational Tree Logic (CTL) formulas. The basic predicates are named like the places in the Petri net, and they become true in the graph if a token is present in the corresponding place. In the example, we may for example ask whether an object reaches a certain state (for example, if the order always gets filled *A(True U filled)*), if a certain method is always invoked (for example, the *Process* method, *A(True U Process)*), if the execution always terminates (*A(True U Deadlock)*) and whether an object leaves a state (for example, if the customer can leave the *waiting state E(True U (waiting AND AX NOT waiting))*). All of them, but the last one evaluate to true in the given scenario. We are able to check method invokations as we

represent them as *places*. This checking is useful as a certain method invokation could have not been modelled in some of the diagrams (*"under-specification"*).

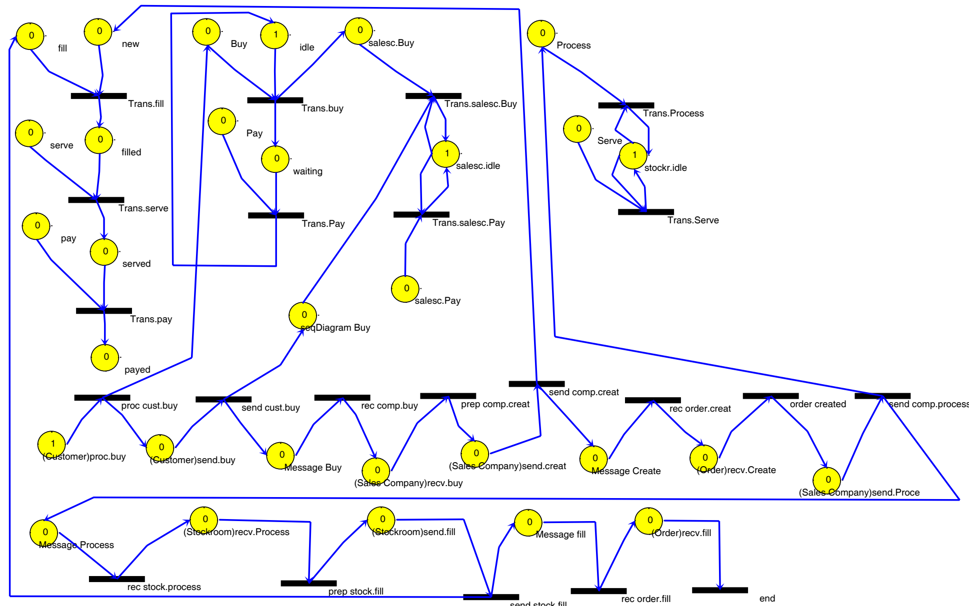

Fig. 5. The Petri Net before simplification.

7 Related Work

Most analysis tools available to the software designer are embedded in a CASE tool in a closed way. The disadvantage of this approach is that one cannot change either the source formalisms or the semantic domain into which these formalisms are transformed. This is necessary, on the one hand, as the source formalism may change (as UML has been changing for the last few years). On the other hand, one would like to use different semantic domains to verify different kinds of properties. On the contrary AToM^3 is open: both source and target formalisms can be defined by the user by means of meta-modelling.

Other approaches also use graph transformations for translating models between formalisms [7]. They express the *denotational semantics* of *single* UML diagrams (single Statecharts). Other approaches aim to directly verify UML models with graph transformations [6], employing graph grammars to specify the *operational semantics* of UML. In our approach, we use graph rewriting to specify the *denotational semantics* of *combinations* of different UML diagrams, translating the UML models into appropriate semantic domains. Other approaches [9] modify the UML notations to incorporate Petri net concepts for the modelling of the dynamic parts of the system. In [11] the OASIS notation is transformed into Object Petri nets for simulation. Our approach has the advantage of using the standard UML notation, and then automatically transforming

these models into a semantic domain, Petri nets in this paper. To our knowledge, this is the first time that this combination of UML diagrams considering over- and under-specification have been automatically translated into B&W Petri nets for analysis.

8 Conclusions and Future Work

In this paper we have presented a general framework based on *Meta-Modelling* and *Formalism transformation* (formally implemented using graph grammars) for the analysis of software systems. As an example we have shown how to transform a UML design composed of class, Statecharts and sequence diagrams into Petri nets for further analysis using Model-Checking. Another important issue that distinguishes our work is that these UML models do not need to be fully specified (some classes may not have an associated Statechart, or the diagrams may not reflect the full details about the method invocations) in order to perform this initial verification.

We are currently working in completely hiding the verification process to the user. In this way, he will not be aware that in order to verify some property, a transformation into a certain semantic domain is performed. In this direction, we are working in back-annotating the model resulting from the transformation, so the result of the analysis in the target formalism can be translated back and given to the user in the context of the source formalism. We plan to continue working on some of the formalisms and transformations depicted in Figure 1, and to include a module in AToM^3 to reason about the best transformation to be performed given the question to be answered.

References

1. AToM^3 home page: `http://atom3.cs.mcgill.ca`.
2. AToM^3 for Verification: `www.ii.uam.es/~jlara/investigacion/UMLandATOM3.html`
3. Clarke, E. M., Grumberg, O., Peled, D. A. 1999. *Model Checking*. MIT Press.
4. de Lara, J., Vangheluwe, H. 2002 *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In ETAPS/FASE'02, LNCS 2306, pp.: 174 - 188. Springer.
5. de Lara, J., Vangheluwe, H. 2002 *Computer Aided Multi-Paradigm Modelling to process Petri-Nets and Statecharts.* ICGT'2002. LNCS 2505. Pp.: 239-253.
6. Gogolla, M., Ziemann, P., Kuske, S. 2002. *Towards an Integrated Graph Based Semantics for UML.* Electronic Notes in Theoretical Computer Science Vol 72(3).
7. Heckel, R., Küster, J., Taentzer, G. 2002. *Towards the Automatic Translation of UML Models into Semantic Domains* AGT'02/ETAPS'02. Pp.:12-22.
8. Murata, T. 1989. *Petri Nets: Properties, Analysis amd Applications.* Proceedings of the IEEE, Vol 77(4). pp.: 541-579.
9. Philippi, S. 2000. *Seamless Object-Oriented Software Development on a Formal Base.* Proc. of the Workshop on Petri-Nets and Software Engineering. pp.: 75–94.
10. Rozenberg, G. (ed.) 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1* World Scientific.
11. Sánchez Palma, P. *Animación de Especificaciones OASIS mediante redes de Petri orientadas al Objeto.* PhD Thesis at Univ.Polit. Valencia.
12. Vangheluwe, H., de Lara, J., Mosterman, P. 2002. *An Introduction to Multi-Paradigm Modelling and Simulation.* Proc. AIS2002. Pp.: 9-20. SCS International.