Experiences in Model Driven Verification of Behavior with UML*

Fabrice Kordon and Yann Thierry-Mieg

LIP6, Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
Fabrice.Kordon@lip6.fr,
yann.thierry-mieg@lip6.fr

Abstract. Model Driven Development (MDD) focuses on the intensive use of models during software development. In this paradigm, models are the central development artifact: transformations are used to derive executable programs, or tests for a given platform. This makes building quality models a cost-effective approach, as the models can be reused for many analysis or generation goals, and not just document a design. However, high quality models are needed for the approach to be successful. Hence the goal of performing analysis of high-level behavioral specifications such as UML, to enhance their quality and detect defects or ensure desired behavior.
High-level specifications provide many facilities to handle large specifications (such as hierarchical structuring mechanisms) and provide sophisticated features to handle programming language's rich semantics. However, the price of these features is that these specifications are difficult to analyse, the semantics are not necessarily formally defined, and the complexity of the language features usually limits analysis to manual inspection, or in the best cases simulation.
On the other hand, formal specifications have been developed specifically with analysis purposes in mind. In particular, model checking is an automatic approach suitable to analyse formally defined behaviors. However, formal specifications languages such as CSP, PROMELA, Petri nets, etc. have a steep learning curve, and are not cost effective since they are not directly linked to code.
In this paper, we explore an approach to integrate formal methods with high-level notations, by translating high-level specifications to formal ones to enable analysis. We are thus bringing Model Driven Engineering to Verification Driven Engineering. We show how this approach was put in practice with UML within the context of the ModelPlex project.

1 Introduction

Industry has always faced a major challenge in the design and implementation of complex systems: how to ensure that they behave appropriately. To do so,

* This work has been partially supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems)

the emerging consensus emphasizes the need for models. We use "models" in the broad sense, meaning tool manipulable descriptions of software design and artifacts. Although simulation and testing are the most common tools used to improve quality assurance, they only increase user's confidence in a piece of software, since by nature they are not exhaustive techniques.

Furthermore, in distributed systems that are becoming the norm, the inherent asynchronous behavior produces non-determinism that testing may have trouble detecting correctly. Only formal methods are capable of proving properties for such systems.

Current practice of software development strongly gravitate around Model Driven Engineering (MDE) that provides guidelines on how to elaborate models at each step of the development process. However, with the notable exception of hardware design, formal specifications are not widely accepted because they are difficult to operate:

- Their application to real-size systems most often requires heavy use of abstractions and tricky encodings of the specification,
- They must be operated using an appropriate formal notation and the related tools. Each one has its strengths and weaknesses, and all are relatively complex to acquire for an engineer,
- The models produced for verification of a design are usually not reusable for other goals such as code generation. Thus a rupture exists between the models and reality, and the investment in the models is considered too expensive for the quality assurance results provided.

This process usually require highly skilled engineers in both their application domain and various formal methods. These people are difficult to find. Thus, adoption by industry remains limited to niche applications, partly because the formalisms used for model checking purposes are considered too difficult to be used by average developers.

In contrast, Model-Driven Engineering (MDE) techniques are gaining attention since machine-readable specifications (models) are more precise, less error prone, can be processed by automated tools, and can allow by code generation to be less dependent on fast evolving technologies [19].

So, the use of formal methods suffer from methodological and technical issues and decrease the benefits of MDE. In [24], we proposed to define *Verification Driven Engineering* (VDE) as an an addition to the use of models all over the software development cycle. Since there is no "silver bullet", the idea is to provide enough information to help picking up the correct technique and tools. By providing model transformations and self configuration mechanisms to choose the adapted formal verification techniques and tools, we can increase their use by industry.

Models and related notations are crucial. When a community uses a notation, and links from that notation to formal methods can be operated, then it is possible to exploit tools and help designers to build safer systems while requiring limited knowledge of underlying techniques.

The objective of this paper is to show how VDE can be operated to provide such help to engineers in an automated way, to answer simple questions such as deadlock detection, bounds of resources, etc. To do so, we propose a general schema for relating high-level notations to formal methods (section 2). Then, this schema is instantiated for UML (section 3) and we present an implementation on some useful properties (section 4) before a conclusion.

2 Connecting high-level notations with formal methods

It is now widely accepted that models should be throughout the design and implementation procedure. However, our specific goal here is to make the best possible usage of these models, for example to check if expected behavioral properties are verified or not.

To perform verification through model-checking, a process like the one described in figure 1 is required. First, it is important to have specifications and associated properties (for example, invariants modeling safety properties). These can be expressed using standard high-level notations of an application domain such as UML.

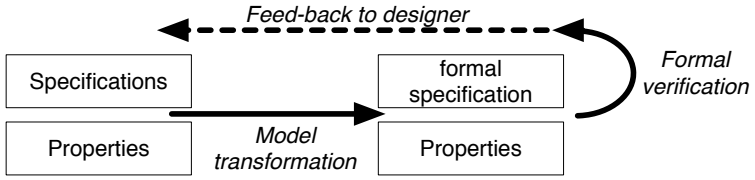

Fig. 1. Development process involving models and verification.

Most high-level specifications are not directly suitable for the computation of properties. Therefore, they must be transformed into formal specifications on which properties can be verified. When the procedure succeeds a positive answer or an error diagnostic is provided. The verification procedure however may fail due to time or memory constraints. This error diagnostic feedback allows system architects to improve their design.

In an iterative development cycle, model validation is introduced in every iteration. It thus is applied on specifications that are successively refined. As suggested in [24], the MDE (Model Driven Engineering) process come to a VDE (Verification Driven Engineering) process. Figure 2 shows the overall development process that is helicoidal. Each step corresponds to a version of the system specifications that are verified and then, potentially corrected or enriched.

We first briefly investigate several types of verification techniques and then identify the main issues to be solved when using formal methods in a VDE approach. Section 3 shows how such an approach can be instantiated for UML.

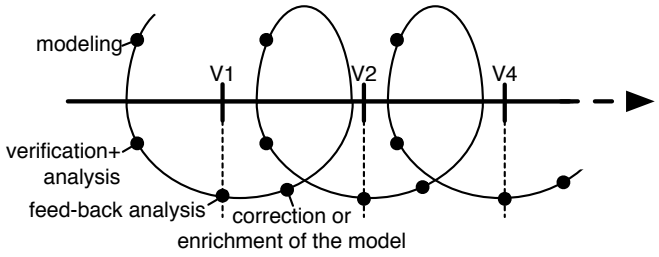

Fig. 2. Helicoidal development life cycle.

UML is by far the most popular and tool-supported language used for industrial specifications [14]. For the formal verification community, this offers the possibility of defining model checking based verification as one of many services on industrial UML models. It thus helps adoption of these formal techniques into mainstream software development methods.

2.1 Choosing a class of verification techniques

There are several classes of formal verification techniques that allow one to reason on a model based on formal grounds. Through different theories (sets, automata, stochastic, etc.), there is a large panel of methods. Let us list two of them.

Algebraic approaches and axiomatic logic such as Z [23] or B [1] allow to describe a system using axioms and then, prove properties of this specification as a theorem to be demonstrated from these axioms. For example, these methods allow one to check for the consistency of interfaces through a complete type checking mechanism, or even to go further and prove theorems (lemmas, invariants) on a system.

These are of particular interest because the proof is parametric and abstract ; for instance a property can hold for a number of entities taken in the natural range. However, theorem provers that help elaborate the proof are difficult to use and still require highly skilled and experienced engineers.

Model checking [10] is an active research domain [11]. It consists in the exhaustive investigation of a system state space. A designer expresses a property to be tested on a model, using a temporal logic formula expressing (un)desired behaviors of the system. This formula is compared with all the paths in the system's state space. If there is a path that does not satisfy the property, then the property does not hold and the returned path (or execution trace) exhibits a counter-example for the property.

The main advantage of this technique is that it is fully automated given a system and a property. However, results obtained are rarely parametric, they answer for a particular set of resources (e.g. N threads). Besides, model-checking

is limited by the combinatorial explosion and can mainly address finite systems. However, many efficient techniques exist such as symbolic decision diagram (BDD) [5], partial order and abstraction methods [12], or symmetry based techniques [7]. They allow to scale up to more complex systems. Some recent studies also investigate model checking of infinite-state systems [25]. Other extensions allow the verification of time-related or probabilistic properties on a model.

The complexity of current software systems and their parallel, distributed and heterogeneous nature raises many challenges when trying to ensure their correctness. In this respect, *model checking* [22] is a formal verification technique which promises to automatically check whether a system satisfies some stated properties. Thus, it seems that model checking is currently a better solution for building push-button tools implementing VDE.

2.2 Raised Issues

However, several important issues must be solved in order to operate such an approach:

- specification issues
- transformation issues
- complexity issues

Specification Issues. To be suitable for verification purpose, the specification must contain precise information about the behavior of its components. If the description is structured, an appropriate way of combining behaviors of components must be provided.

For instance, in a notation like UML, information is consistent in diagrams but lacks semantics when diagrams are connected. For example, state-machines describe the behavior of a *Classifier* (*Class* or *Component*). But how this behavior is connected to a description based on sequence diagrams is rather unclear. Therefore, if verification can be performed on isolated diagrams like in [32] for state-machines or in [13] for sequence diagrams, it is difficult to ensure consistency of the entire system specification.

To overcome this first problem, profiling can be considered. The missing semantic information can be stated by means of annotations that are exploited to fix identified "semantic variation points" and other places where the semantic is imprecisely defined.

Another important point concerns the specification of properties. Invariant specification is currently used in industry, for example to specify the profile of unexpected events in the system. OCL is the standard notation to express invariants in UML specifications. However, temporal logic is more difficult to use and is not fully standardized.

To overcome this second problem, sequence diagrams can be used as a more friendly way of specifying temporal logic formulae. These diagrams provide a way to help designers to describe expected causalities between events (since

sequence diagrams express a partial order over events). These descriptions of expected behavior can be used for verification.

Let us note that a notation like AADL [29] solves these problems by providing a unified way to relate component to annotations[1]. AADL components describe a piece of the system while annotations define either component features or component properties. Thus, both the system, its behavior (in an annotation) and its expected properties are defined together. This is helpful when transforming the full specification of a system into a formal notation.

Transformation Issues. High-level concepts must be mapped to formal specifications. When elaborating the transformation rules, it is important to consider:

- The granularity of atomic events: it must be tuned appropriately to limit complexity and while remaining precise in the specification.
- The traceability of transformation to provide comprehensive feed-back to engineers in terms of the high-level specification.
- The consistency between the transformation of the specification and its associated properties, particularly if different modeling notations are used.

Since no formal verification technique fits all the needs it is necessary to choose the most appropriate one. Therefore, several transformations from the high-level notation to various formal models must be considered. Each transformation is dedicated to the verification of a given property using a given technique.

Complexity Issues. Combinatorial explosion in the verification process is another reason to consider several transformation. It is important to select, for a given formal notation, the most efficient verification technique. Such a technique depends on the formal notation but also on the property to be verified. Therefore, there can be several transformation from the high-level notation to a given formal notation, each one being optimizing some configuration.

As an example, let us consider deadlock detection in Petri nets. It can be achieved by means of model checking but also trough structural analysis for certain types of Petri Nets like in [2, 3]. The structural approach usually scales much better, though it is less general than full model-checking. Customizing a transformation for deadlock detection is thus of interest.

2.3 An approach to enable VDE

The elaboration of a VDE approach requires a methodology: one must cope with the issues and elaborate its transformation and verification tools. The approach we propose has four steps:

[1] in AADL, annotations are called "properties".

1. Selection of the semantic scope to be handled in the high-level specification. This is necessary to bound the transformations rules. It is possible to extend the scope when a first set of transformation rules are validated.
2. Selection of the appropriate formal notation according to the type of property to be verified. It is also necessary to identify the technique to be used for verification.
3. Identification of the abstraction level to be considered in the high-level specification. This corresponds to the selection of the elements to be considered in the high-level notation for the transformation to be elaborated.
4. Mapping of the selected elements to the formal notation. It is important to identify the elements that correspond to the "glue" in the system such as hierarchical composition or communication mechanisms. The transformation rules handling these elements are of particular importance because they orchestrate others.

Through these phases, it is important to preserve traceability information so that results can be expressed in the original notation.

3 From UML to formal methods

Model-Driven Engineering (MDE) development methods are gaining increasing attention from industry. In MDE, the model is the primary artifact and serves several goals among which code generation, requirements traceability, and model-based testing. MDE thus enables cost-effective building of models vs. direct coding of an application. In this context, model-based formal verification of behavioral consistency is desirable as it helps improve model quality.

In this section we present a translation based approach using (parts of) UML [28] as source and Instantiable Petri Nets (IPN) as target, to enable formal verification. IPN is a hierarchical formalism defined expressly for this purpose; it is not meant to be used as a front-end modeling language but rather as a powerful back-end verification formalism. The semantics and concepts are thus kept simple, they are described in detail in [30]. We show that IPN are adequate to support the introduction of model checking in an MDE process. The approach is implemented in a prototype tool called BCC: Behavioral Consistency Checker.

3.1 Verification of UML

UML is a standard defined using a meta-modeling approach to establish the concepts of the various diagrams, and plain English to describe dynamic aspects of the specification. UML2 has introduced an important refactoring of the description of actions and behaviors, which allows to describe the behavioral diagrams (activity, state machine and interaction diagrams) using a common base. The semantics of a UML2 model is thus more precise than in UML 1.x. But so-called semantic variation points are deliberately left in the standard, to help UML fit all possible application fields of software engineering. The semantics of UML thus remains imprecise and subject to interpretation.

The idea of providing formal semantics to (at least parts) of the UML by translation to a more formal description is widely used [4]; it allows to use the UML as a (relatively) user friendly graphical environment and exploit the existing formal verification techniques and tools without redeveloping them specifically for the UML. It also is compatible with the ideas of Model-Driven Engineering (MDE) in which the model is the central artifact, and translations from the model to other artifacts (code, tests...) with specific goals in mind is the norm.

One of the challenges when defining such a translation is to handle the composition of parts of a behavior, defined using different types of diagrams. This touches on the problem of consistency among UML diagrams of a given system: although structural consistency rules are defined (using OCL) and can be checked mostly syntactically, the multiple viewpoints offered by the UML induce the possibility of defining incompatible behaviors quite easily.

Given the wealth of work on translation of UML models for analysis purposes, we focus here on a few propositions which are the most similar to this work, particularly those targeting dialects of Petri nets in the translation.

Merseguer et al. have done an important work ([6] gives an overview) on translation of the UML to labeled generalized stochastic Petri nets (LGSPN) for performance analysis purposes. Tool support is provided through the ArgoSPE tool set and the model checker GreatSPN. This work is perhaps the closest to ours because the semantics used to compose the LGSPN is based on synchronizations, like in our IPN. However, that work is centered on performance evaluation rather than consistency checking. Moreover LGSPN do not allow hierarchical compositions.

The work of Eshuis et. al [17] also gives a formal semantics to activity diagrams through translation to workflow nets, and more recently to NuSMV. However they concentrate mainly on a subset of activity diagrams compatible with UML 1.5, and do not handle hierarchy of the description.

Shatz et al. have also done a large body of work [21, 31] on formalization of UML semantics, mostly centered on state charts, that uses Petri nets as translation target. The focus of that work is in correctly capturing the full run to completion semantics of UML state machines. It thus does not address the problem of inter diagram consistency checking.

Engels et al. [16] have elaborated a formalization framework based on CSP as semantic domain. This work is mainly focused on protocol state charts of UML 1.4. It does not handle hierarchy as to enable scalability in the specification.

Overall the main originality of this work is in the composition mechanisms used and the fact we natively support hierarchy instead of requiring a flattening of the representation, which leads to scalability issues. We also present an implementation in section 4.

3.2 Instantiable Petri Nets

In this work we have used Instantiable Petri Nets (IPN) as a target in the transformation. This formalism is presented in detail in [30], we give an informal presentation here sufficient to explain the subsequent transformation.

IPN explicitly support the concepts of *type and instance* that allow to adequately match the structure of high-level models designed with UML. They also allow to reuse parts of a model in various scenarios, and offer good scale up properties to handle large and complex specifications. Finally, since instances of a type share a common definition, they allow to capture internal regularity or symmetries of a system, which can be exploited for model checking.

IPN explicitly support a compositional definition based on the notion of *synchronizations*. A wide corpus of theoretical [18] and empirical [8] results (introduced by process algebra such as CSP [26], up to applicability to probabilistic systems [15]) show that this mode of composition is favorable to better compositional verification algorithms.

Their definition is split in two parts.

First, we define Elementary Petri Nets, which are essentially standard Petri nets in which transitions bear a *visibility* that may be private or public. A public transition is part of the *interface* of the net; it can only be fired if it is sollicited through an external synchronization. Transitions private to a net can be fired according to usual Petri net semantics.

We then define a Composite type, to hold instances of elementary nets, or, recursively, instances of a composite type. A Composite may contain synchronizations, again labeled with a visibility. A synchronization forces synchronous firing of its *parts*, that is transitions belonging to the interface of the contained instances.

This hierarchical formalism uses a semantic largely inspired by process calculi such as CSP. It also borrows from component based formalisms (such as Corba component model (CCM), or Fractal) the notions of components defined as hierarchical composition of simpler bricks.

Examples of IPN will be provided in the next sections, a formal definition of IPN can be found in [30].

3.3 How to translate

The issues relating to transformation of UML to a formal notation are explained here through an example.

We use in this paper UML activity diagrams. A similar approach can be applied to other UML diagrams that represent a *Behavior*. The essential characteristic of a behavior is that it begins with an occurrence of a start event and ends with a termination event occurrence (UML Superstructure, section 13, p.419 [28]).

The transformation is based on a set of patterns of transformation. The principle consists in building one IPN type per diagram of the UML specification.

These types will then be instantiated and assembled according to various verification scenarios. Hence for each activity diagram we build an elementary IPN type.

We first apply the translation rules for the various types of *nodes* of the UML activity diagram, using the patterns described in figure 3.

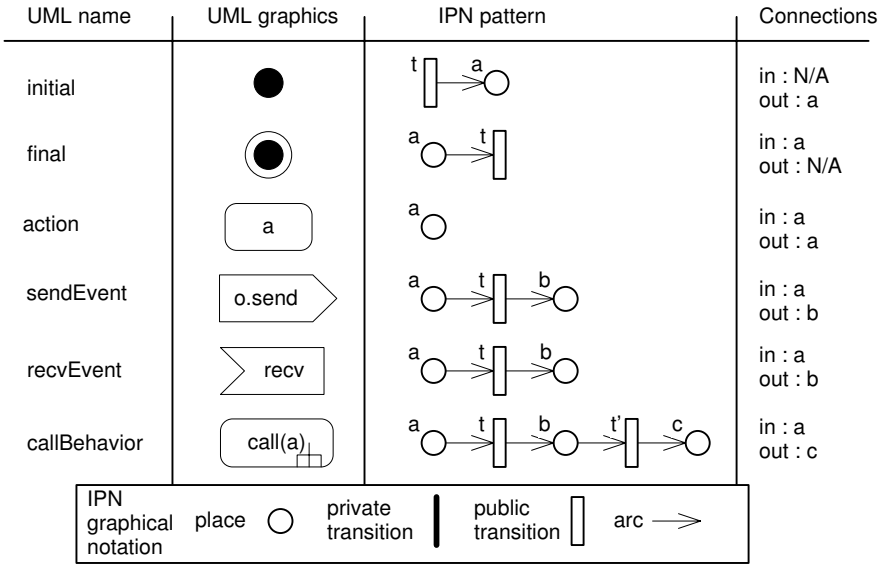

Fig. 3. Translation rules for nodes of UML2 activity diagrams. Patterns are expressed here in a simplified form: each generated public transition is also associated to sufficient information to enable appropriate connections in the linking phase (see section 3.4).

Places: Each node gives rise to one or more places and zero or more public transitions. The most complex case is the *callBehavior* pattern where the place b represents a state where we are waiting for the called behavior to complete. We keep track during the application of this transformation of the *in* and *out* places generated for each UML object. After this first pass all places of the resulting IPN have been produced; each control flow has a source (the "out" place of the source activity) or a destination (the "in" place of the target activity) or both. An additional case not represented on this figure (but used in the example see fig. 6) arises for control flows that link two control nodes (e.g. merge to fork): we produce an additional place for these edges that acts as both as source and destination in the translation.

Interface: The public transitions produced are meant to be synchronized with other diagrams: the transition labeled t of the *initial* pattern is meant to be synchronized with the transition t of the *callBehavior* pattern. Similarly, the transition t of the *final* pattern is meant to be synchronized with the transition

t' of the *callBehavior* pattern. Transition t of the *sendEvent* pattern is meant to be synchronized with the appropriate corresponding transition t of the *recvEvent* pattern of the receiving object.

States: We additionally define two labeled states for each diagram, *active* which assigns one token to the place corresponding to the UML initial node, and *passive* (the default) in which all places are initially empty.

Transitions: We then translate the control flows and control nodes of the UML activity diagram, using the patterns described in figure 4. In these patterns, the activities noted a, b and c are just placeholders for the actual nodes that the edges connect to: the translation pattern is centered on a control node, and queries the control flows that link to it to obtain the appropriate source or target place that was defined in the first translation step.

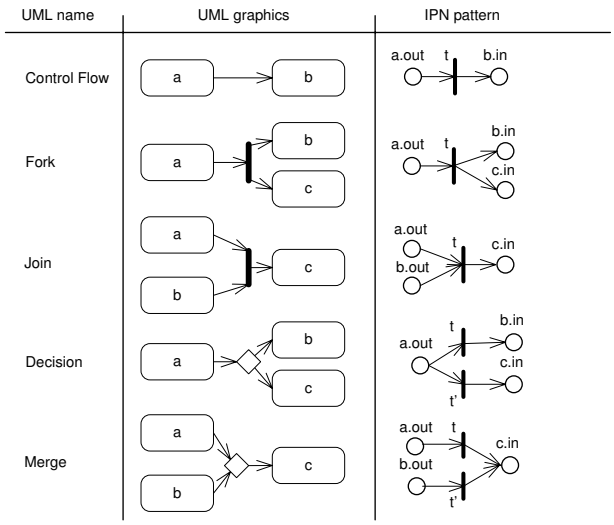

Fig. 4. Translation rules for edges of UML2 activity diagrams.

These patterns are mostly straightforward Petri net interpretations of UML semantics, so we hope a reader familiar with both UML and Petri nets will understand these translation patterns without further ado.

All the transitions produced in this phase have private visibility: interaction with other diagrams happens within nodes in activity diagrams. This is to be contrasted with the reactive semantics of UML state-machines, in which edges are the main point of interaction with other diagrams.

Application to an example Figure 5 presents a small example that describes an order and shipping process. It contains many of the UML features our translation supports. The action HandlePayment of the "Order" activity is a *CallBehavior* action that refers to the behavior described by the "HandlePayment"

activity. The translation yields one elementary net for each activity, graphically depicted in figure 6. Names and graphical layout have been added to the figure to help the reader track the transformation.

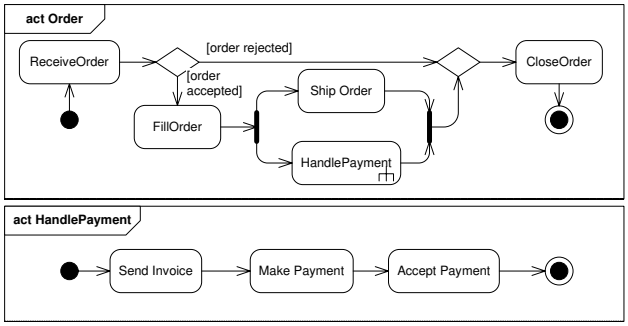

Fig. 5. An example adapted from the UML standard, p.357.

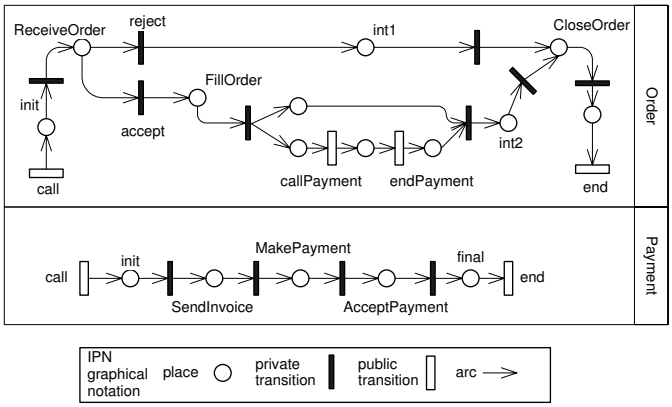

Fig. 6. IPN elementary net types obtained by application of the translation rules.

3.4 Composing Diagrams

The first translation step has allowed to build an IPN type for each diagram of the original specification. Additionally we have built a trace that gives for each UML behavior of the original specification the name of the IPN type produced and the set of links to other behaviors that need to be resolved.

Inter-diagram links: The information contained in a link depends on its nature :

– For send event type links, we have the target UML object instance and the name of the transition that sends this event (i.e. noted t in Fig.3).
– For receive event type links, we have the event and the name of the reception transition (i.e. noted t in Fig.3)
– For CallBehavior actions, we have the target UML Behavior and the name of the transitions corresponding to the call and behavior end (i.e. t and t' in Fig.3)
– Finally for CallOperation actions, we have the target operation name and the target UML object instance as well as the names of the call/return transitions of the IPN.

We then incrementally build more complex scenarios by using different linking strategies.

Diagrams in isolation: The most basic strategy consists in building "isolated" behaviors from each behavior (see Fig.7 top). For a net type n representing a behavior b, we build $isolated(b)$as a composite type containing a single instance i of type n, two public transitions *call* and *end* that synchronize to $i.call$ and $i.end$ respectively, and one private transition synchronizing to each transition of i mentioned in the links. The public link transitions corresponding to interactions with the environment are made private in the isolated type, thus enabled at will provided local conditions allow it.The states active and passive are also defined, as associating the corresponding state to i. The *isolated* construction thus allows to control the consistency of a diagram in an *uncontrolled environment* setting.

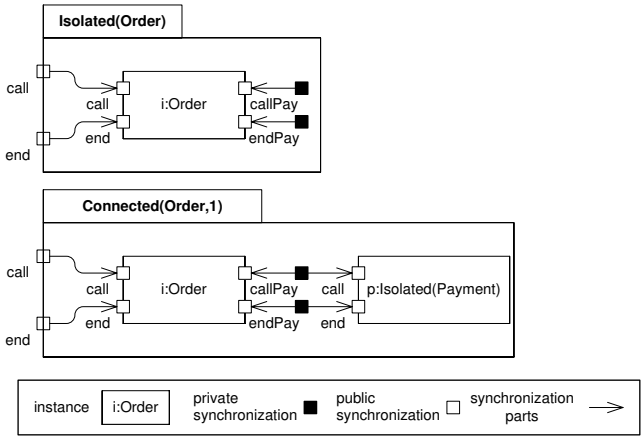

Fig. 7. Composite IPN types produced to analyze composed behaviors

Composing diagrams: Inductively, we can then build types $connected(b, k)$ (see Fig.7 bottom) corresponding to connections resolved up to depth k, where $isolated(b)$ is equivalent to $connected(b, 0)$. The type $connected(b, k+1)$ contains one instance i of type n (representing b) and for each behavior b' mentioned as

a link target, an instance of the $connected(b', k)$ net type. The *call* and *end* transitions are again exported with public visibility, and the links are actually resolved by synchronizing the link transitions of n with the *call* and *receive* of the depth k behavior instances.

UML behaviors can be associated to classifiers or operations. The main use case we have considered is a model where operations may be defined by activity diagrams. Class diagrams are exploited in the following way: when operations of the class are associated with activity diagrams, we construct an IPN type that contains instances of the IPN types corresponding to the activities and "export" the public transitions of the nested activity diagrams. In particular the transitions that allow to start (initial node pattern) and detect the end (final node pattern) of an operation are made visible to connect to CallBehavior of other objects.

When the class contains objects by composition, the corresponding IPN types are instantiated. This gives us a context necessary to determine the target of a CallBehavior or SendEvent action.

3.5 Defining Consistency Checks

One of the main issues when defining properties to verify is that the user (at the UML level) should not explicitly manipulate the formalisms used to specify properties, typically LTL or CTL formulae. This limits the scope of what we can verify, but as we show here, many useful checks can already be defined without user intervention.

Verifying Properties A model-independent property can be defined regardless of the model instance considered. Typical examples include absence of deadlocks or livelocks, boundedness... These are the easiest to handle as they naturally do not require user input.

We first run a structural bounds computation tool (based on [27]), that produces for each place its structural min/max bounds. This tool scales very well as the check is *structural* so the complexity is linked to the number of places and transitions of the specification, and not to the state space size. This tool allows to detect:

- dead code: [0..0] bounded places indicate unreachable model elements. The interpretation here is that the model element associated to the place is either unreachable (logical design error) or disconnected from the rest of the diagram (modeling bug). In practice, the second interpretation is often correct, as a common error, due to the GUI of UML modeling tools, is to delete elements from a diagram but not from the model;
- unbounded behavior: if a place marking is bounded by $+\infty$ this usually indicates a serious misuse of fork/join constructs. This error was actually raised several times in the models provided by industrial partners of ModelPlex. Neither testing, nor simulation tasks which were working with the same models correctly identified this issue. Simulation actually detected a

choke point during performance evaluation and suggested attributing more resources to this faulty activity. Testing would need an infinite test sequence to correctly tag this problem.

When the previous check does not detect unbounded behavior, we can proceed to use model checking based tools for more advanced checks. At this stage we check for absence of deadlocks in the specification. Existence of deadlocks is tagged as an error.

We also check for activity diagrams that their final state is reachable when the initial state is set to "active" (see section 3.3). This is tagged as an error if it is not verified.

The list of verification goals is extensible as we think of new checks interesting for the end-user.

3.6 Conclusions on the Transformation

We thus support the BasicActivities, IntermediateActivities and StructuredActivities packages of the standard, which means our tool handles UML compliance level 2 (see UML2 superstructure [28] for definition of compliance levels).

The weakness of this translation is in the non-determinism of decision nodes, and the absence of any data manipulation. While this may seem a severe restriction, in practice the diagrams we studied (within ModelPlex) were mostly annotated in plain text. They correspond to early phases of design where the logic of the control flow is the focus. Other diagrams we studied were obtained by translation from BPEL, a language for describing the workflow of a business process, where again data manipulation is not the focus. In any case, supporting code level annotations requires extending the standard with a tool specific profile, as the UML standard does not define any data types other than String, Boolean and UnlimitedNatural, or any concrete syntax for actions (e.g. arithmetic).

To overcome this weakness, the transformation could be refined to take into account data, possibly using a CEGAR like approach [9] to limit the induced complexity. This is a perspective at this stage.

The strength of the approach is that it preserves the modularity of the UML specification. Thanks to the concepts of IPN, a UML translation is quite easy to set up, and only one IPN type is produced per UML diagram. When considering several scenarios and a system composed of interacting objects, the fundamental notion of instantiation from object-orientation which is preserved in IPN allows to adequately reuse a model defined in parts. The semantics of IPN are sufficient to capture the concepts of UML behavioral diagrams (sequence, alternative, fork join and parallel behavior, synchronization on events, multiple instantiation...).

Backward correspondence between generated IPN and the original specification is easy to trace since the transformation preserves the structure of UML model. Thanks to this, namespaces are preserved and propagated to IPN objects, thus allowing to trace each transition or place to the corresponding UML model element. THus, when errors are detected in the IPN, it is possible to reconstruct and display a trace in the original UML notation (see figure fig:cs:sap in section 4 where the problems reported refer directly to the UML specification).

4 Behavioral Consistency Checker

We have implemented the approach described in section 3 as a proof of concept in the BCC (Behavioral Consistency Checker) tool.

BCC is designed on top of Eclipse to transparently use formal model checking to provide behavioral consistency checks on UML specifications. Requirements on the tool include that it should need minimal user training, and that the underlying verification technology has to be transparent. To reach these aims, BCC uses UML diagrams as input, and produces easily understandable compiler-style errors and warnings when consistency rules are violated.

This tool was developed as a contribution to the European integrated project ModelPlex[2] (21 industrial and academic partners, 20 M€, 36 months). Within the project, models produced by industrial use case providers are used for several other goals than just verification, such as code generation, requirements traceability and test generation. Thus the investment in the modeling effort is amortized, and any check we can implement that improves the quality of the models is valuable, as the approach is model-centric, and model quality is a goal in itself in this setting. BCC will be integrated in a "model-based simulation, verification and testing (SVT) workbench" that is one of the project deliverables.

The tool is implemented in Java using the EMF framework to parse standard XMI UML models. The transformation is written entirely in Java, rather than using a model transformation engine such as ATL: we had non trivial traceability issues otherwise. Model-checking tools (written in C or C++) are run transparently on the CPN-AMI model-checking platform [20]. Post-interpretation of verification results is again written in Java, using the transformation traces intensively.

A prototype of the tool is available at `http://move.lip6.fr/software/BCC/` and uses services of our CPN-AMI Petri net Framework as a back-end to compute some properties. It currently handles activity diagrams and a subset of state-machine diagrams. It will be completed by the end of ModelPlex (March 2010). We are working on improved class and component diagram handling. Component diagrams allow to have a specific connection topology as the *parts* are class instances. We also work on a sequence diagram translation to IPN. It will allow to control that the sequences describing an expected behavior are indeed executable. Thus it is a means for the user of specifying model-specific properties to check.

Case Study BCC was used on the case study models from industrial partners in the project. We briefly present here some verification results obtained using BCC.

BCC can be run directly on a XMI UML file, but it is integrated as an Eclipse plugin relying on the EMF/GMF validation service and Eclipse UML2 tools. The user simply sets the scope by selecting one or more UML packages (folders) to be analyzed, and selects the "Validate" action. The model is explored

[2] `http://www.modelplex.org`

to detect diagrams to be checked, then transformations Preconfigured properties are selected by the designer and then it really works as a push-button tool that can be operated without any knowledge of the underlying techniques.

One study provides a good illustration about the use of such tools. The SAP case study where a small portion is represented in figure 8. This model specifies the activity diagram of a web store system. When payment is to be done, the system must both check credit card and stock. If one condition fails, then it restart the payment procedure (back to *ask client* state).

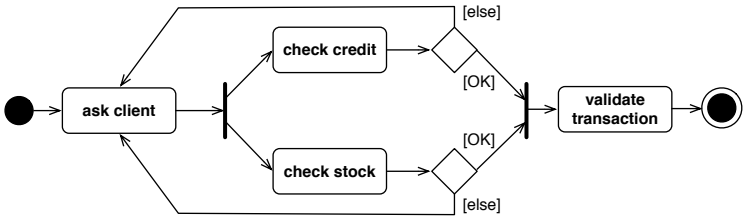

Fig. 8. Portion of the web store system specification.

BCC automatically performs all implemented checks and shows that the specification contained structurally unbounded parts, meaning that, whatever the number of resources provided to execute *check stock* and *check credit*, these transition will still constitute a bottleneck.

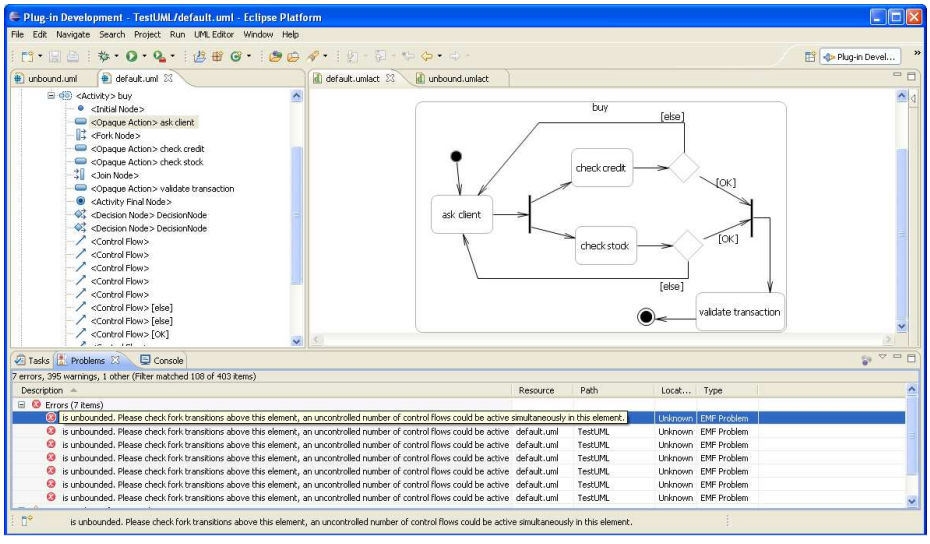

Fig. 9. Window showing bad results for the web store system.

Figure 9 show the BCC user interface in Eclipse. The tool is invoked through the contextual menu "validate model" in the model subwindow. Then, the detected problems are reported in the standard "problem view" (below the main window) which usually contains compilation errors. Messages refer to UML objects, double clicking on an error outline the problematic model element in the model browser window.

In parallel with BCC, model-based simulations and test generation for the same system have been performed. Performance oriented simulation detected potential bottlenecks in both *check stock* and *check credit* procedures. However, no explanation of this phenomena was provided. Testing may not be able to capture this bad property of the system, as the bad behavior is exhibited only in infinite traces. Thus model-based verification is a good complement to other model-based validation techniques.

Such model-independent properties (no dead code or boundedness) are of interest to compute on any type of specification. Therefore, a tool like BCC is valuable to help engineers to improve systems early in the software life cycle (in our case, at design level), which is typically a goal of VDE.

5 Conclusion

This paper proposes an approach to extend MDE (Model Driven Engineering) to VDE (Verification Driven Engineering). VDE is MDE together with an intensive use of formal verification techniques to compute properties directly from models, as soon as possible in the software life cycle.

We exemplify this by providing a way to verify automatically some basic properties on UML model without any intervention from an engineer. This example has been implemented in a tool BCC (Behavioral Consistency Checker) ; its experimentation in a project shows that it produces interesting information to engineers to debug their models.

A main point of this paper is the presentation of an original translation mechanism based on Instantiable Petri Nets (ITS), as a basis to express concepts of a higher level modeling language (such as UML). ITS offers a hierarchical way to assemble subsystems or elementary Petri Net components by means of a well-suited set of synchronization mechanisms. ITS also enable efficient verification techniques suitable to tackle large specifications.

So, ITS appear to be an pivot language suitable to:

- Define an operational semantics for a high-level modeling language (here, it was experimented for UML collaboration diagrams and state charts),
- Use this semantics to perform formal verification on the system by means of appropriate techniques such as structural analysis or model checking.

This paper also provides a step-by-step method to elaborate a verification schema based on transformation techniques and patterns associated to the concepts provided in a high-level modeling language. One could use this method, to implement VDE for other languages as soon as its behavioral semantics can be captured by means of appropriate patterns.

References

1. J.-R. Abrial. *The B book - Assigning Programs to meanings*. Cambridge Univ. Press, 1996.
2. K. Barkaoui and I. Abdallah. Deadlock avoidance in FMS based on structural theory of Petri nets. In *International Conference on Technologies and Factory Automation (ETFA)*, pages 499–510, 1995.
3. K. Barkaoui, J.-M. Couvreur, and C. Dutheillet. On liveness in extended non self-controlling nets. In *16th International Conference on Application and Theory of Petri Nets*, volume 935 of *LNCS*, pages 25–44, London, UK, 1995. Springer-Verlag.
4. M. Broy, M. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. *Models in Software Engineering*, pages 318–323, 2007.
5. J. Burch, E. Clarke, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue from LICS90)*, 98(2):153–181, 1992.
6. J. Campos and J. Merseguer. On the integration of uml and petri nets in software development. In S. Donatelli and P. Thiagarajan, editors, *27th ICATPN - Petri Nets and other models of concurrency*, volume 4024, pages 19–36. Springer-Verlag Berlin Heidelberg, June 2006.
7. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On well-formed coloured nets and their symbolic reachability graph. In K. Jensen and G. Rozenberg, editors, *Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application.* Springer-Verlag, 1991.
8. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
11. E. M. Clarke, E. A. Emerson, and J. Sifakis. Turing award for their original and continuing research on model checking, 2007.
12. W. Damm and H. Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *LNCS*. Springer, 2007.
13. W. Damm and B. Westphal. Live and let die: Lsc based verification of uml models. *Sci. Comput. Program.*, 55(1-3):117–159, 2005.
14. B. Dobing and J. Parsons. How UML is used. In *Communications of the ACM*, volume 49, May 2006.
15. S. Donatelli and G. Franceschinis. The psr methodology: Integrating hardware and software models. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 133–152, London, UK, 1996. Springer-Verlag.
16. G. Engels, R. Heckel, and J. Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogolla and C. Kobryn, editors, *4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts and Tools*, volume 2185, pages 272–286. Springer-Verlag London, October 2001.
17. R. Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

18. A. Gupta, K. McMillan, and Z. Fu. Automated Assumption Generation for Compositional Verification. *Computer Aided Verification*, pages 420–432, 2007.
19. B. Hailpern and P. Tarr. Model-driven development: The good, the bad and the ugly. *IBM Systems Journal*, 45(3):451, 2006.
20. A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. New features in cpn-ami 3: focusing on the analysis of complex distributed systems. In *ACSD*, pages 273–275. IEEE Computer Society, 2006.
21. Z. Hu and S. M. Shatz. Explicit modeling of semantics associated with composite states in UML statecharts. *Automated Software Engg.*, 13(4):423–467, 2006.
22. M. Huth. Some current topics in model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(1):25–36, 2007.
23. ISO/IEC 13568. Z formal specification notation — syntax, type system and semantics, 2002.
24. F. Kordon, J. Hugues, and X. Renault. From Model Driven Engineering to Verification Driven Engineering. In *6th IFIP Workshop on Software Technologies for Future Embedded & Uiquitous Systems (SEUS 2008)*, volume 5287 of *Lecture Notes in Computer Science*, pages 381–393, Capri, Italy, October 2008. Springer.
25. P. Madhusudan, editor. *Proceedings of the 9th International Workshop on Verification of Infinite-State Systems (INFINITY'07)*, Electronic Notes in Theoretical Computer Science, Lisboa, Portugal, September 2007. Elsevier Science Publishers.
26. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.
27. T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, Apr. 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.
28. OMG. *Unified Modeling Language: Superstructure - Version 2.1.2 formal/07-11-02*. OMG, November 2007.
29. SAE. Architecture Analysis & Design Language V2 (AS5506A), jan 2009. available at `http://www.sae.org`.
30. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. In S. Kowalewski and A. Philippou, editors, *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, Lecture Notes in Computer Science, York, United Kingdom, Mar. 2009. Springer. To appear.
31. S. Yao and S. M. Shatz. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *CIC '06: Proceedings of the 15th International Conference on Computing*, pages 289–297, Washington, DC, USA, 2006. IEEE Computer Society.
32. Q. Z. and B. Krogh. Formal verification of statecharts using finite-state model checkers. *IEEE Transactions on Control Systems Technology*, 14(5):943–950, September 2006.