Domain Based Verification for UML Models

Pascal André, Gilles Ardourel

LINA - FRE CNRS 2729
2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
gilles.ardourel@univ-nantes.fr

Abstract. The software development processes with UML generate numerous documents, including UML models. In order to improve the specification quality and the process quality, it is necessary first to define specification properties and then to verify them. In this paper we propose a framework for practical verification of UML models that is consistent with an iterative and incremental process. We illustrate it on several examples.

Keywords: UML, Model, Consistency, Verification, Tools

1 Introduction

Modelling object systems with UML [5,7] is a widespread technique in the development community, especially in Model Driven Engineering (MDE). However, the quality of the produced models is usually low: models are hardly communicable, consistent, correct, complete, precise, homogeneous... Many reasons explain this situation:

1. There are no standard definitions for models. Every one defines which combination of notation elements is a model for a given development process, which are not standardised. In other words, models are process dependent and not notation dependent.
2. The UML notation is not fitted to the current practise of modelling. On one hand, it is too large and complex (*e.g.* UML2 defines 13 diagrams) and modellers borrow only a small part of the notation. On the other hand, there are missing concepts for specific application areas or implementation techniques (*e.g.* Real-Time, Web) for which stereotypes are not sufficient.
3. There is no formal agreement on the UML semantics, despite its metamodel. Every UML user has an underlying semantics, depending on his experience in the object oriented development field and the computation domain (information systems, real time...).

In summary, we feel that UML is rather a Babel tower than an Esperanto language. So how can we control the quality of UML products ?

The common answer is to define quality factors and criteria and to measure them on the models. In the following, the criteria are the properties to be checked on models and the measurement is the verification of the properties. During

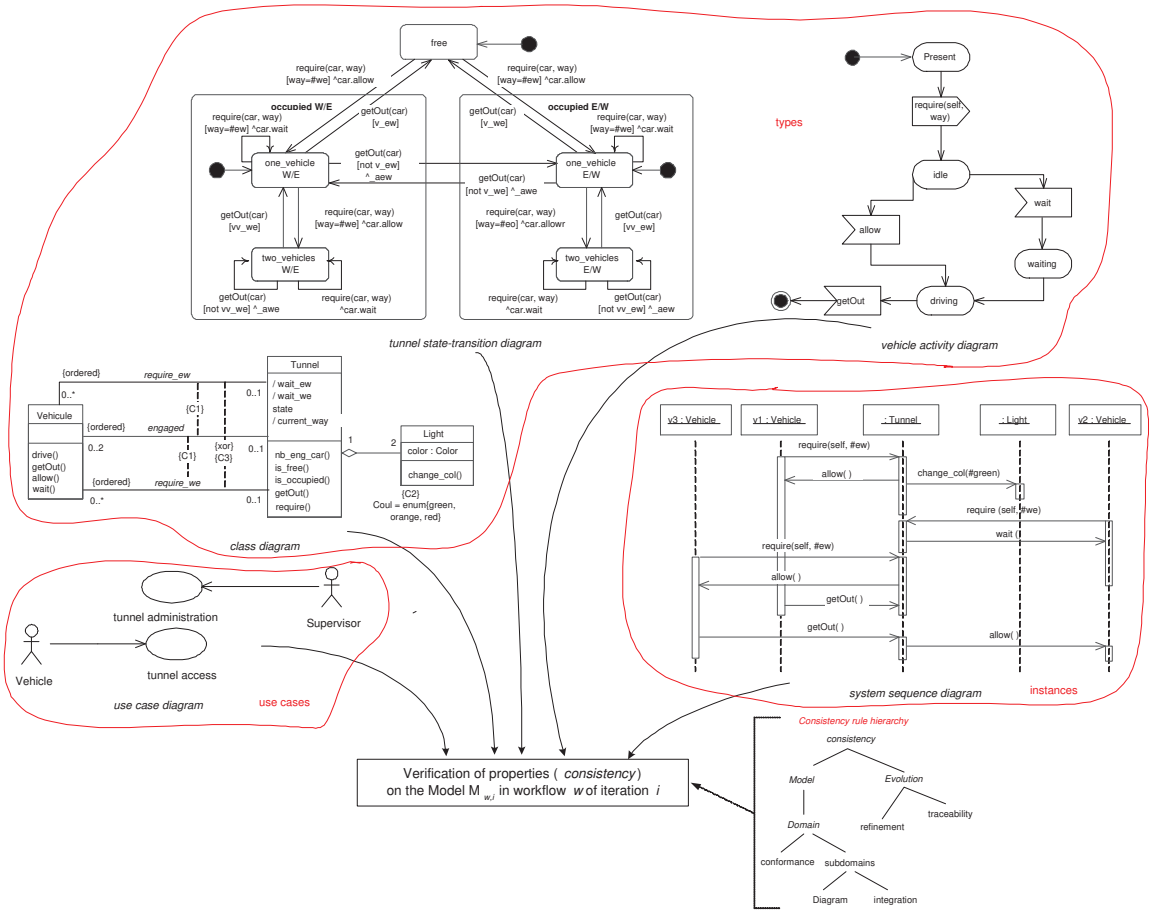

Figure 1 : *Verification of properties on models*

a development process, several models are produced. The models must have properties which are checked by some verifications. The context of this work is thus a quadruple **process-model-property-verification**. In the example of Figure 1 the goal is to check the consistency of some model $M_{w,i}$ in some workflow w of an iteration i in the process. The case study is the management of vehicle access in a tunnel.

The properties can be classified into three distinct categories: **system properties** (dependent on the target application, these include validity, safety, liveness, reliability, etc. They are often associated to a specific point of view -static, dynamic, functional). In the Tunnel case study, a safety (system) property is that the vehicle can always run. **model properties** (mainly consistency and the completeness of the models) **process properties** (focus on the evolution of models during the development process, mainly the traceability, consistency

and the completeness of the model evolution). Some properties traverse the categories. This is the case for the **consistency** property. A consistency (model) property is *"the message send conforms to the operations definition"*. A refinement (process) property is *"the class definitions conforms to the Use Case"*. An evolution (process) property is *"the classes of the second iteration are consistent with the ones of the first iteration"*.

The contribution of this paper is a general framework and a tool support for handling the verification of properties on UML models. In this paper, the target property is consistency, but the discourse applies to other properties. Section 2 states the four foundation principles of the framework. These principles are further exploited in the remainder of the paper. In section 3, we propose a structuring of the models that support a structuring of the consistency. Section 4 overviews the main issues of the techniques for verify the consistency of models. In section5 and 6 we focus on the rule based part of the verification. The tool support is overview in section 7. Last we conclude and discuss several perspectives of this work.

2 A framework for verification

Our motivation is to take the problem of *verifying the properties of UML models* in the current practice of development, *i.e.* in a *large* and *practical* view. By *large*, we mean that we try to cover the main development workflow and notations. Contrarily to the executable UML languages [12,13] which handle a reduced (even formal) notation subset, we intend to cover the concepts of the 9 or 13 UML diagrams. By *practical*, we mean that verification must also include some aspects of the development process: in MDE approaches, the model cover orthogonal aspects or point of view (heterogeneity) ; they are transformed along the workflows, (abstraction levels), they evolve along the increments (iteration). Such a verification an overwhelming task because the concepts involved at different levels are numerous, the semantics of UML model elements cover a wide spectrum and the specifications are incomplete by nature. Moreover, the properties are mutually dependent: what means completeness without consistency ? what means consistency without correctness ?

We propose a framework based on four principles: divide and conquer, reuse and adapt, support customised verification, be independent and evolve.

Principle 1 (divide and conquer) *The verification should apply to the plain UML notation.*

To be applicable in practice, the verification should work on any UML model. In order to organise both the models and the verification and manage their complexity, the notation is partitioned into domains and models. Each domain is related to some workflow of the development process. This point is further studied in section 3.

Principle 2 (reuse and adapt) *The verification problem does not have to be started from scratch, thus it must reuse and adapt existing techniques and tools.*

This point is further studied in section 4. The idea is to smoothly combine the formal techniques and the rule based systems, which are the two main and complementary approaches for verification.

Principle 3 (support customised verification) *The verification must be customisable depending on the wishes of the designer and the current stage in the development (iteration, workflow).*

The property tester can define a verification process to assert a property (predefined combinations are stored in libraries). The verification process should manage incomplete models (there are many 'incomplete' PIM before a PSM) produced along the development. Besides completeness-based rules filtering, it should also be possible to set priorities and weights on the rules according to different policies. This point is further studied in section 5.

Principle 4 (be independent and evolve) *The verification tools should be independent from a specific case tool and should be able to adapt easily to changes in versions of UML.*

This principle conforms to the MDE philosophy: managing repositories to accept various metamodels (described in various MOF), defining a verification service as an additional service of CASE Tools. This point is further studied in section 7.

3 Manage Models and Consistency

Modelling practice shows that the notation panel is too large when considering each phase or each workflow. In this section we study how to combine the model elements during the development process according to principle 1.

The UML notation is structured over consistent element arrangements: the nine or thirteen diagrams [7,5]. The diagram concept does not exist in the notation metamodel (the only structuring mechanism is the *package*) nor in development processes *e.g.* Unified Process (the structuring unit is the model which is a also a package in the metamodel). The model's content is not formally defined. "A model is an abstraction of a system" [15]. Each process workflow accepts one or several models as input and produces one or several model (e.g. UC model, analysis model, deployment model...). Note that domain models, business models, test models and non-functional requirements are omitted here.

Thus, in order to avoid confusions in a seamless use of UML, we propose a three level structure that facilitates the verification process: element, diagram, model. First, we think that diagrams are more than visual notations (graphs): they define (quite) consistent combinations of model elements and should be used accordingly. This idea is not new and is present in most CASE tools but it is not formalised. Second, we advise to restrict the number of diagrams available for each activity. We promote three abstraction levels: external, logical, physical. Last, we separate the instance-related diagrams from the type-related diagrams. The notation structure is summarised in figure 2.

– The external (or user) level corresponds to the requirements workflow. It focuses on a high level notation (use cases, scenarios and flow-activity diagrams). While flow-activity diagrams can be used for modelling the main functions of the system (with swimlanes), we think their semantics is quite ambiguous, inspired from various formal models (Petri Nets, Data Flow Diagrams, Control Flow Diagrams).
– The logical level corresponds to the analysis and design workflows. It focuses on object modelling (classes, sequences, collaborations/communications, state machines, do-activities).
– The physical level deals with the low level abstractions (components, deployment, code) of implementation and tests.

Composite structure diagrams and package diagrams are implicitly parts of many of the above diagrams. The interaction overview diagram is a mix between sequences and activities, which has a rather fuzzy semantics. Timing diagrams are quite disjoint from the above diagrams, especially for consistency. The crossing of the diagrams in the levels results in a partition of the diagrams[1] into domains.

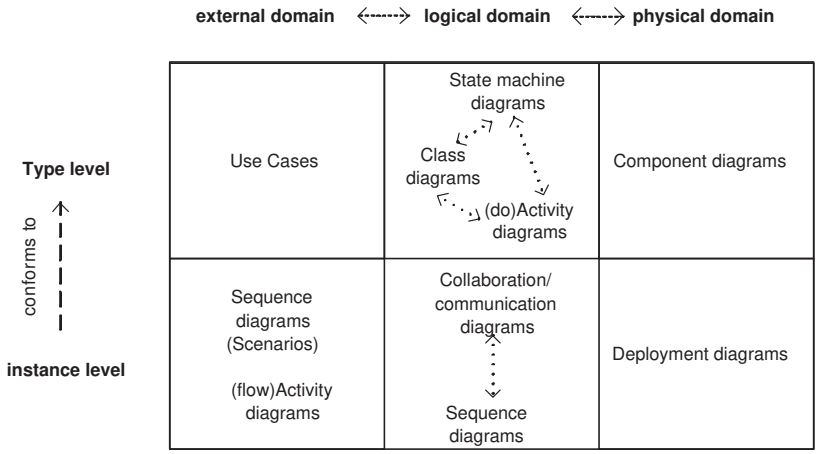

Figure 2 : *Domain structuring for an iteration i*

Each diagram (if we consider scenarios as simplified sequence diagrams) appears in one cell only: this is a partition. This improves the readability of the UML models : it is easy to find the abstraction level it refers, and its relevant place in the development workflow. But it reduces the expressive power because the context of use is restricted. For example, activity diagrams are only related to operations or state activities (inside a state of a state transition diagram).

Each cell is a sub-domain. A model is a view on a domain. Based on this structuration, the **consistency** property can be specialised into smaller properties (e.g. the property hierarchy of figure 1).

[1] But not a partition of the notation since the same model element may appear on several diagrams.

1. Diagram consistency
2. Sub-domain consistency = inter-diagram consistency
3. Domain consistency = Sub-domain consistency + conformance (instance/types)
4. Workflow consistency = Domain consistency + refinement (or traceability)
5. Process consistency = Workflow consistency + evolution consistency

Our structuration is a bit finer than the ones of [16,10], but domain (resp. workflow, process) consistency is close to horizontal consistency (resp. vertical, evolution). Note that refinement is quite hard to define when there is no seamless models, and therefore traceability is a looser constraint.

4 Integrate Verification Techniques

We consider two main approaches for automated property verification of UML models: formal methods and rules based systems. There are also specific techniques for managing consistency among models. Details on these approaches can be found in [10,9,8].

The Formal Methods Approach (FM) is a MDE approach since the model is transformed into a formal model written in a formal language that is supported by automated tools for model checking or theorem proving. These can be plain languages (*e.g.* Z, B, CASL, LTSA) or OO extensions of them (*e.g.* Object-Z, OOZE, Troll, Maude) which have less supporting tools [6,11]. Formal methods are also an answer to the UML Semantics weakness [14,17] in providing a formal semantics for (a part of) the UML notation (or metamodel), *e.g.* in the pUML group [3,4]. The numerous proposals dealt mainly with the class diagram, the state machine diagram and the sequence diagram. The (formal) model of evolution is refinement, thus the proof of the evolution consistency depends on the proof of refinement. Verifying the consistency of multiple viewpoints is still an active research area in the formal methods community. Other operational semantics, say executable UML [12,13], can be classified as formal. They are very useful for validating the UML products but not for verifying them.

The Rules-Based Systems Approach (RBS) is a test approach, where verification rules are applied to models to check target properties. A rule is a statement describing a property (or a part of it). The rule language can be formal (Description Logics, OCL, algorithm, Java code...) or informal. This is also the CASE Tool approach, since CASE Tools implement rules and their checking directly on the graphical modelling editors. Nevertheless, rules can be defined at the model level or the metamodel level. Examples are given in [18,10,9]. The main issue of rule oriented verification is property management, because there is a great number of rules depending on the element combination, the diagrams, the models, the syntax, the typing system and the semantics.

Discussion Table 1 compares both approaches including sections on the Model-Property-Verification triplet. The FM approach is not a direct answer in our context. It assumes complete models, a premise which is not true in the earlier steps of the development. The current proposals cover only a part of the notation with a particular semantics which is not fitted to a large panel of applications. Nevertheless, the formal approaches are very useful to prove the system and model properties within these parts (modelling niches). The advantage of the RBS Approach is that the choice can be customised to the development process and the verification goal. As a counterpart, the main problem is building specific rules and ensuring a complete and consistent set of rules.

	Formal Approaches	**Rule based Approaches**
	Model	
description	a formal model	a source model
UML notation	partial	total
	Properties	
expression	formal statements (e.g predicate)	semi-formal statement
user-defined	formal statements	OCL assertions ?
system prop.	according to the target semantics	none
model prop.	yes	yes
process prop.	refinement	traceability, evolution
	Verification	
process	automatic for model properties individual for system properties	select and test -
checking	formal proof	algorithm (or review)
User knowledge, role experience	UML, formal model property definitions and proofs proof theory	UML rule selection rule database knowledge
Advantages	formal semantics (non ambiguous, consistent...) (refinement to code)	customizable according to - the development step vthe application nature
	automation	no more than UML
	soundness from the formal model	programming and tests
Drawbacks	heavy, user proof experience	implementation dependent
	partial UML notation	consistent, complete ?
	two models (source + formal)	selection require user skill

TABLE I– Synthesis comparison

Table 1 shows that the two approaches are complementary. A verification framework that smoothly combine both approaches would really be the best solution. We propose the following coordination for consistency checking:

1. Diagram consistency is achieved by **transformations** into formal notations, because diagrams are, by nature, based on consistent theories.
2. Sub-domain consistency corresponds to multi-view integrations, both formal techniques and rule-based techniques apply here.
3. Domain consistency can be solved using conformance dependencies (see [8]) in a rule-base approach or a formal typing system.

4. Workflow consistency is merely obtained by a rule-based approach on refinement and traceability. A true formal refinement relations is hard to define here because the domains have quite different notations.
5. Process consistency depends on the four lower level consistency checking technique. The diagram evolution consistency can be seen as diagram refinement in the same formal framework but it is merely obtained using evolution maintenance techniques such as description logics in [16].

In summary, the low levels of consistency are achieved with transformations into formal methods because there are many, often syntactic, rules. The higher levels of consistency are verified using rule-based approaches. In a previous work, we experienced the former approach using algebraic specifications, where consistency has a special meaning [2]. In other words, when the number of rules really grows up (especially when you consider other properties than consistency) we found it beneficial to use a formal model. Formal models are also more appropriate for type subdomains than instance subdomains whereas the opposite is true of the rule-based techniques.

5 Verification

In this section, we describe verifications as structuring units for a verification process whom goal is to check properties.

5.1 Defining a Verification

Several verifications can be linked to a property. For instance, the *consistency* of a model from the logical domain needs the following verifications:

- conformity of the messages sent to an object in a sequence diagram with the messages supported by the type of the object and with the messages that the emitting object is allowed to send.
- conformity of the sequence of messages with the state diagram of the class of the receiving object
- consistency of the actions from a state diagram with its class.

We define a verification by a name, a goal (informal comment) and a workspace. It can be realised with **rules** (algorithms, predicates, code) or **proofs** in a formal model. It is good practice to define rules as OCL (*Object Constraint Language*) expressions whenever possible.

The *workspace* is related with the part of the metamodel which concerns the verification. In UML, this metamodel, which describes UML, is the level M2 defined by the OMG. The workspace is composed of:

- U, the set of elements from the metamodel which are needed for the verification. It is partitionned into U_v, set of the elements to be verified, and U_n the set of elements that need not to be verified.

– C, the set of the metamodel elements concerned but not used by the verification. The instances of the elements of C in the metamodel will be marked as invalid if one of their subelements is marked as invalid.

U depends on the verification, while C and the partition of U into U_v and U_n depends on the verification process. For instance, the verification of conformity between sequence diagrams and a class diagram has class diagrams and sequence diagrams as U (that is classes, interfaces, methods, objects and messages), messages as U_v and sequence diagram as C.

A verification of a model M1 is done in four steps:

– selection of the elements of M1 that are instances of U and conversion if necessary,
– application of the rule or the proof associated with the verification,
– interpretation of the results and marking of the elements of M1,
– propagation of the result on the elements of M1 that are instances of C.

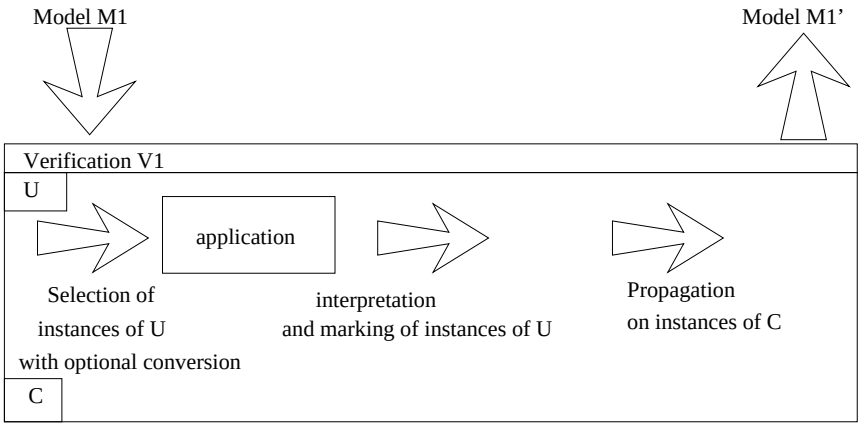

Figure 3 : *Applying a verification on a model*

5.2 Process and Composition of Verifications

In a rule based verifications, the user choose the rules and launch the verification [18]. Each rule is then applied *independently* on the model. This requires a deep knowledge of the rules and of the reasonable sequences of their applications.

We wish to help both the designers of verification and the users, using a verification process. A verification process is the cumulative application of functions working on models. These functions can be **verifications** or **filters** A very simple filter can be the removal of every element previously marked as invalid. The application order of the functions is important when filtering is used. Indeed, a verification $v1$ can invalidate an element which is an input of a verification $v2$ if $U(v2) \cap (U_v(v1) \cup C(v1)) \neq \emptyset$. In cases of cross dependencies, choosing the order is a crucial step of the processus design. The following figure shows a simple verification process.

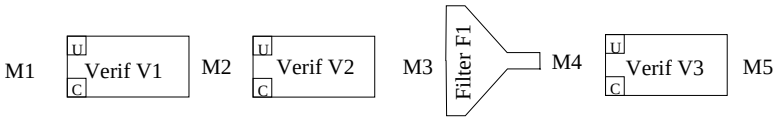

Figure 4 : *Applying a verification process on a model*

In order to simplify the management of verifications and to automatise process creation, we introduce the following composition operator:

comp(name, goal, functions list)→ verification(U, C).

The four steps associated with a composed verification are

- selection: no filtering is necessary
- application: contained functions are applied in sequence
- interpretation of result: final result is a predicate on the value of the annotations given by the contained verification
- propagation of result is done according to the optional set C.

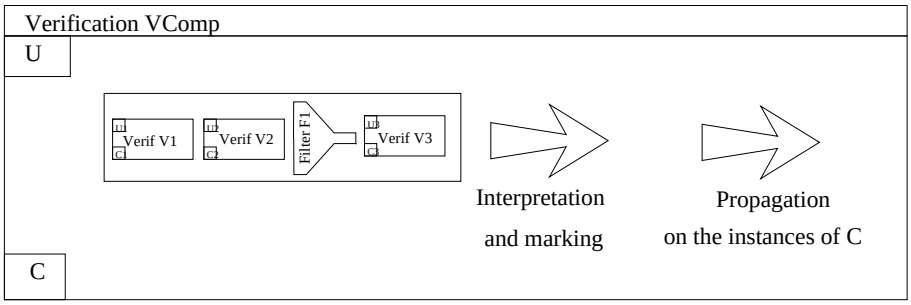

Figure 5 : *Composing verifications*

A precise definition of the sets U_v and C helps to transform the binary result of a formal verification into a result acceptable for the rule-based system.

5.3 Manage Incompleteness in Verifications

The treatment of incompleteness is achieved by two means:

- a parameter in the workspace definition that indicates the level of details that is required. For example, iin a first iteration, the class name or the operation names can be sufficient. We currently defined 5 levels: diagram syntax, nouns resolution (class, package, object, types), nouns resolution (attributes, relations, operations, state), type consistency, action consistency (pre/post conditions, guards).
- an adequate selection of the rules in the definition of the process. Since the verification is customizable, the rule selection requires some skill from the tester.

6 The Verification Process

We first introduce the basic rule management and then we introduce the verification process, since it has an influence on the rule expressing.

6.1 Basic Rule Management

The implementation of the verification process is based on the following triplet: property, rule, control.

The first step is to find the properties of models or groups of models. This is a huge work and unfortunately, there is no library of rules available. We did not find detailed examples of rules to be checked in the plethoric litterature on UML. Usually, the authors give some tracks like some questions to be asked. A more informative source is the OMG products, especially the UML Semantics and UML Notation guide [7]: the allowed element combination, the OCL expressions of the metamodel. Last, one can find ad-hoc rules in the visual modeling tools and try to formalize them.

The second step is to define precisely the way the property can be checked, say a rule. We can simply write the rules as sentences in a structured checking manual, the verification process is to take each rule manually and to check it. To be reliable, efficient and cheap, the rules are to be written in a formal language, merely executable. This can be a Java code, an OCL expression (*Object Constraint Language* [19]), algebraic axioms, predicates...). As far as it is possible, OCL is adequate for UML models, since it belongs to the same standards. The MOF description of UML is a powerful means for structuring the rules since they benefit from the object flavor: the rules can be attached to modeling elements, specialized, delegated through associations... For example, the simple expression `not self.allParents->includes(self)` in the context of a *GeneralizableElement* means that inheritance is not circular. Note that, many classification problems occur: a rule defined for an element applies to all it specialized elements until exceptions have been quoted. For example, each classifier may have attributes and operations, so a use case (which is a classifier) can have attributes and operations. Structuring the rules, like building the UML metamodel, is a classification task.

The third step is to control the rules on the model. This is not simply a boolean function which delivers correct or error. As done for compilers, the error can be classified from ignored warning to non-recoverable error. The erroneous model elements can be removed, corrected. The control algorithm is related to the model management and the verification process. An example is given in section 6.3.

6.2 Hierarchical Rule Management

Controlling the syntactic and semantics aspects of UML models lead to hundreds of rules. A flat list of these is thus unmanageable, we need to organize the rules. We already introduced several classification criteria for the properties:

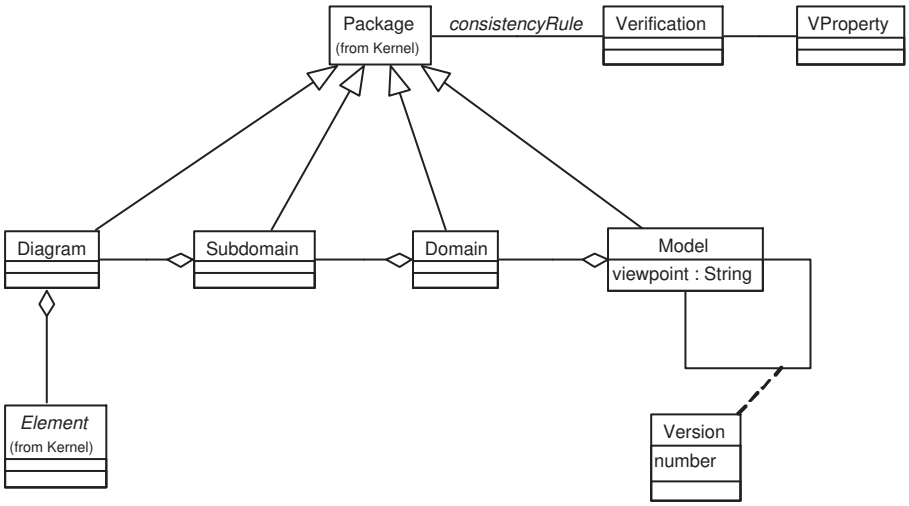

Figure 6 : *Simplified Meta-model for Verifications*

- system / model / process
- external / logical / physical domains
- instance / types
- multi-view consistency
- syntax, typing, semantics

Moreover, the same rule may be defined differently according to the development step (principle 3), because informations are missing. Last, as in every a rule based system, we need a rule selection process that helps the user to check the properties, according to some criteria. In [18], the user selects the rules to apply. It requires some skill and knowledge. It can be combined with other approaches, like precision level (according to the development iteration and the workflow), coarse/fine grain element description. For example, an operation can be described by its name in a scenario, by a profile (name and parameters), by a signature (a name, result and parameter types), by pre/post conditions in OCL, by a target code.

Our hierarchical verification of consistency is based on the domain structuring of figure 2. The checking holds on each domain and each domain relation. The simplest way to structure the rules is to associate the verification processes (section 5.2) with the notation structuring nodes of section 3. This is a quite common meta-representation of rules. The verification hierarchy is not further developped here.

6.3 Some Logical Domain Verifications

Here, we just overview the core of verification and show examples. The static rules are organized in the following way:

1. Combination rules: for example, an interface describes only operations, the specialization relation is not reflexive.
2. Constraints: typing, multiplicities, constraints on relations (or, xor), OCL constraints.
3. Exceptions related to some semantics: for example, no multiple inheritance for the subclasses of a discriminator, a passive object cannot send simultaneously two messages. These include the constraints associated to stereotypes.

The logical model is a multi-aspect model. Each aspect is a diagram. The verification is naturally based on this decomposition: diagram verification and model verification. While we read many documents, we only refer to the UML Semantics (metamodel and well-formed rules) and Notation Guide of [7], because they are the only reliable sources, even if many information and examples are just informal.

After the individual verification of the diagrams, the correctness, consistency and completion of the models are examined. The consistency link between the diagrams is the object paradigm. The domain level is mainly based on the Behavioral and Action parts of the UML Semantics, which were not achieved in the former release of UML.

The verification process handles two abstraction levels: **instances** and **types**. The diagrams related to instances are interaction diagrams (objects, sequences, collaborations). Instance diagrams are occurences that conforms to their type definitions. The diagrams related to types are class, statecharts and activity diagrams. Type diagrams define all what is possible in an object system. The type level is organized as follow: the class diagrams defines the structuring entities (structural view) and the operations (functional view), each class defines its behavior by a state machine (dynamic view), the activities of some states are defined by an activity diagram. We clearly restrict the usage of state machines and activity graphs in UML, in order to manage the type level complexity.

Including this reality, the checking process is the following:

1. Type level
   (a) Classes/State machines: we ensure the well-foundedness of guards, transitions and actions.
   (b) Classes/Activities: we ensure the well-foundedness of guards, transitions and actions.
   (c) State machines/Activities: we ensure the consistency of activities against their composite states.
2. Instance level
   (a) We ensure the consistency of collaborations and sequences representing the same reality (space and temporal viewpoints).
   (b) We ensure the compatibility of messages and objects...
3. Instance/Type level. This part is mainly based on the semantics of the triplet: *message*, *action*, *event*. For a long time, the three concepts have been studied in separate contexts: instance diagrams for the messages and state machines for action and events. The former issues from OO programming and the latter from real-time applications. The semantics of the triplet is still an open

problem. Consider a simple consistency rule: a message induces an event by the receiver, which can receive other events, an action sends messages.

7 Tool Support

Depending on a particular case tool or a particular version of UML, MOF or OCL is a serious obstacle for a tool development. According to the principle 4, we want the support tool to be

- evolving with the standards. Indeed, the UML and MOF models are ongoing standards, thus a verification tool should evolve
- independent from a CASE Tool. Since UML is a standard, the verification should not depend on a specific implementation.

Thus verification can be seen as a component or a service in a MDA approach. The model are transformed (or immersed) in a verification context which is quite different to a transformation into OO code.

The verification framework developped in the Bosco tool [1]. The Bosco tool[2] is a generic open source project that focus on the internal representation of sofware development models. Bosco parameters are XMI files describing meta-models and meta-metamodels. It currently works for MOF release 1.4 and UML metamodels releases. Its entries are software development models (e.g. UML models, ER models, ...) generated from editing tools. Bosco add-ins are model driven functions: the user implements modeling functions such that model checking, code generation, documentation generation, model transformation (e.g. from UML 1.5 to UML 2.0) and so on.

The current release includes transformations to algebraic specifications for the logical type level subdomain. The other domains have been studied, but only a catalog of informal rules issued. The rule based approach handle the full logical domain. The rules are currently implemented by algorithm and OCL constraints.

8 Conclusion and future work

In this paper we proposed a framework for the consistency problem in sofware development with UML and MDA. We assume a large notation, covering the UML notation (but not the profiles), and a practical use of it for modeling the artifacts of the development (iterative and incremental process). This implies to manage the language complexity (numerous concepts, diagrams and models) and the model incompleteness (due to the development habits). Our answer is a general verification framework based on four principles: (1) manage the language complexity by a notation structuration and apply consistency on that structure, (2) try to reuse various techniques in both formal methods approaches and rule-based approach, (3) manage incomplete specification by defining verification levels and customizable verification processes, (4) promote an extended

[2] `http://bosco.tigris.org`

verification tool support, as a service, which is standard evolving and CASE tool independent.

We currently implemented a FM approach for diagrammatic consistency verification and a RBS approach where rules are checked by algorithms and the verification process are customizable.

Since the project is ambitious, there remain many work to do. On one hand, the transformation approach has to be extended to include other (existing) formal techniques especially for the Message Sequence Charts (MSC) and a complete notation for statemachines. On the other hand, the RBS approach is to further develop. The rule catalog has to be completed, especially for OCL descriptions. The structuration of a properties and rule is to be improved and applied to real case studies, generated by current UML CASE Tools. A (theoretical) work on the consistency of the rule set is necessary, based on some logics. Last, the rule definitions should be as independent of the UML standard as possible.

References

1. Pascal André, Gilles Ardourel, and Gerson Sunye. The Bosco Project, A JMI-Compliant Template-based Code Generator. In W. Dosch and N. Debnath, editors, *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering*, pages 157–162, July 2004. ISBN 1-880843-52-X.
2. Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. Checking the Consistency of UML Class Diagrams Using Larch Prover. In T. Clark, editor, *Proceedings of the third Rigorous Object-Oriented Methods Workshop*, BCS eWics, January 2000.
3. Andy S. Evans, Jean-Michel Bruel, Robert B. France, Kevin C. Lano, and Bernhard Rumpe. Making uml precise. In *OOPSLA'98 Workshop on "Formalizing UML. Why and How?"*, October 1998. Vancouver, Canada.
4. Andy S. Evans, Robert B. France, Kevin C. Lano, and Bernhard Rumpe. The UML as a formal modelling notation. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.
5. Martin Fowler and Kendall Scott. *UML Distilled*. Object-Oriented Series. Addison-Wesley, 3rd edition, 2003. ISBN 0-321-19368-7.
6. S.J. Goldsack and S.J.H. Kent, editors. *Formal Methods and Object Technology*. Springer-Verlag, London, 1996.
7. Object Management Group. The OMG Unified Modeling Language Specification, version 1.5. Technical report, Object Management Group, available at http://www.omg.org/cgi-bin/doc?formal/03-03-01, June 2003.
8. Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, and Jean-Louis Sourrouille, editors. *Proceedings of «UML» 2004 Workshop on Consistency Problems in UML-based Software Development III*, Lisbon, Portugal, October 11 2004. http://uml04.ci.pwr.wroc.pl/Workshop-materials.pdf.
9. Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean-Louis Sourrouille, and Miroslaw Staron, editors. *Proceedings of «UML» 2003 Workshop on Consistency Problems in UML-based Software Development II*, San Francisco, California, USA, October 20 - 24 2003. IEEE and Blekinge Institute Of Technology. ISSN 1103-1581 also Research Report 2003:06, http://www.ipd.bth.se/consistencyUML/Consistency_Problems_in_UML_II.pdf.

10. Ludwik Kuzniarz, Gianna Reggio, Jean-Louis Sourrouille, and Zbigniew Huzar, editors. *Proceedings of «UML» 2002 Workshop on Consistency Problems in UML-based Software Development*, Dresden, Germany, October 1 2002. Blekinge Institute Of Technology. ISSN 1103-1581also Research Report 2002:06, http://www.ipd.bth.se/uml2002/RR-2002-06.pdf.
11. Kevin Lano and Howard Haughton, editors. *Object-Oriented Specification Case Studies*. Object Oriented Series. Prentice Hall, 1993.
12. Stephen J. Mellor and Marc J. Balcer. *Executable UML - A Foundation For Model-Driven Architecture*. Addison Wesley, 2002. ISBN 0-201-74804-5.
13. Chris Raistrick, Paul Francis, Ian Wilkie, John Wright, and Colin B. Carter. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004. ISBN 0-521-53771-1.
14. Gianna Reggio and Roel Wieringa. Thirty one problems in the semantics of uml 1.3 dynamics. In *Proceedings of the OOPSLA'99 Workshop on "Rigorous Modelling and Analysis of the UML: Challenges and Limitations*, 1999.
15. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Software Development Process*. Object-Oriented Series. Addison-Wesley, 1999. ISBN 0-201-57169-2.
16. Jocelyn Simmonds, Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Maintaining consistency between UML models using description logic. *L'Objet, Actes de LMO'04*, 10(2-3):231–244, 2004.
17. Anthony Simons. 37 things that don't work in object-oriented modelling with UML. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 209–232. Technische Universität München, TUM-I9813, 1998.
18. Ambrosio Toval, Jose Sàez, and Francisco Maestre. Automated Property Verification in UML Models. In Stefan Gruner Michael Leuschel and Stéphane Lo Presti, editors, *Proceedings of the 3rd Automated Verification of Critical Systems (AVoCS'03)*. DSSE Technical Report DSSE-TR-2003-2, April 2003.
19. Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Object-Oriented Series. Addison-Wesley, 1998. ISBN 0-201-37940-6.