A Methodology for UML Models V&V

A. Baruzzo and M. Comini
Dipartimento di Matematica e Informatica (DIMI),
University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.

***Abstract*—The introduction of UML models in the software life cycle poses new issues and challenges that are not adequately supported by current state-of-the-art development tools, especially concerning V&V activities. Indeed, every tool usually focuses on a small set of specialized activities (such as design, coding or testing), failing to provide a satisfactory (general purpose) V&V framework.**

In this paper we propose a methodology which allows a seamless integration of V&V into a UML-based development environment. The methodology exploits a set of supporting tools designed to be integrated in a unified framework. We believe that such proactive collaboration between tools can reduce significantly both the effort and time required to tackle consistency, correctness, quality and long-term maintainability of UML models, increasing the development productivity and the overall quality of the delivered software system.

***Index Terms*—Software engineering, UML models, Validation and Verification, CASE Tools**

With his book "*Writing Solid Code*"[9], Steve Maguire popularized the concept of assertions in the vast community of C/C++ programmers, and explicitly damped the technique of defensive programming. The ideas expressed in the book where not new but forced the attention on some aspects that are very actual even nowadays. Indeed he encourages the systematic use of assertion to check not only the outcome computed by a program, but also the implicit and often latent *assumptions* that constrain the implementation. This is an important issue because software continues to grow in complexity, both in terms of the number of lines of code and in terms of the number of interacting technologies required for implementation. One way of managing this complexity is to raise the abstraction level of software development. This is the approach followed in model-driven software development (MDD), wherein models of software are the primary artifacts of development, not just mere documentation. However, even rising the level of abstraction from code to models, we are faced to the same Maguire's original questions:

- Are our models correct, consistent, and suited to their purpose?
- Are they an expression of good design?
- Are they maintainable in the long term?

Answering to the first question means both *validate* (we have built the right model) and *verify* (we have built the model right) the specification. Answering to the last two questions implies addressing issues pertaining to design quality. V&V activities for UML models involve several tasks, which range from simple syntactic and static semantic checking to dynamic property verification and to design heuristic fulfilling. In order to automatize the V&V process, these activities need to be supported by different tools. In the following sections, we propose a methodology divided in phases and workflows. For each phase then we discuss the required tool support that should be provided by the development environment.

I. Our methodology for UML model V&V

The methodology we propose combines two existing process models: the *Unified Process* (UP) [7] which is an incarnation of the iterative and incremental life cycle that characterizes almost all object-oriented methods, and the *V-Model* [10], which is a variation of the waterfall model that demonstrates how V&V activities are related to analysis and design.

Similarly to the UP process, the methodology is organized in the following four phases (see Figure 1):

1) *Inception* - which includes feasibility analysis, requirements elicitation, and the necessary development environment setup;
2) *Elaboration* - which performs the traditional object-oriented analysis and design activities;
3) *Construction* - which is heavily focused on coding, unit, integration and system testing;
4) *Transition* - which includes acceptance testing, deployment and maintenance of the delivered system.

The workflows (the set of activities performed in each phase) are grouped into two broad categories:

- *Primary workflows* - which includes business modeling, requirement analysis, system analysis and design, coding, testing, and deployment activities;
- *Supporting workflows* - which includes project management, configuration and change management, development environment setup.

For lack of space we do not describe in detail phases and workflows as they are simply the standard activities of the UP process. Instead, we discuss some aspects that characterize the integration of the V-Model in our approach, differentiating it from other process models.

As the left part of the "V" depicted in Figure 1 shows, our methodology tries to work at the model level as much as possible. Software models, especially those elaborated during phase 2, are very far from being complete. This fact emphasizes a peculiar feature of our approach: in order to embrace the V&V activities at the model level right from the start, the verification methods have to work with *incomplete*

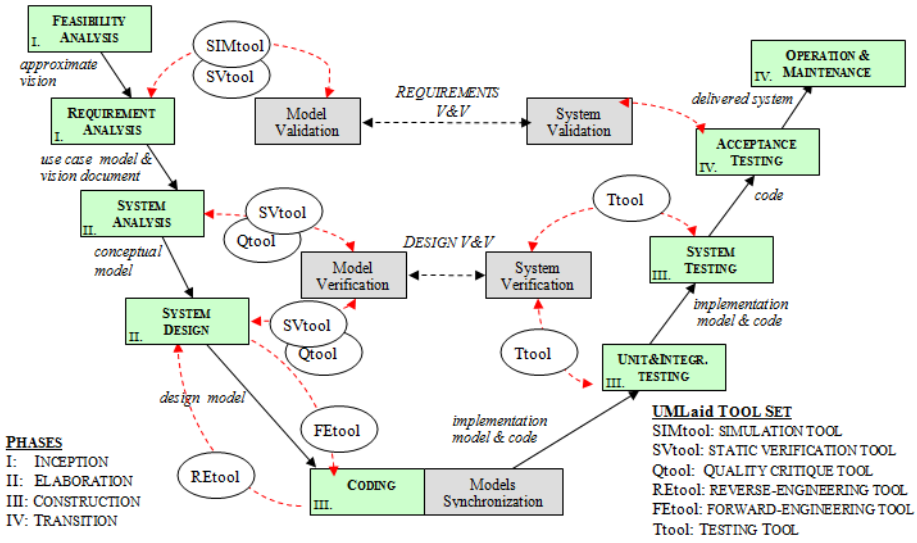

Figure 1. Phases, workflows and supporting tool set for automatize UML Model V&V

specifications and *without the code*. This allows the designer to fix possible design flaws when it is much more economic to do. Obviously, this does not mean that, if code is available, all relevant information cannot be extracted from it and exploited properly. The key feature required by tools to have such capabilities is a compositional definition of the task to be performed.

Since a complete static V&V is not possible (it is undecidable), it is necessary to complement the static methods with dynamic ones, such as testing, as shown in the right part of the V-Model of Figure 1. However, the models previously developed can also be exploited here (using suitable model-based testing tools), in order to partially automatize the generation of test cases, and thus supporting a full-fledge model-driven approach. Moreover, the information produced by the static activities is used to limit the tests just to the parts which have not been proved right. Hence, the integration of the two approaches can alleviate the respective weaknesses.

Similarly to the original V-Model, the proposed methodology makes more explicit some of the iterations and reworks that are hidden in the development process. For example, during the construction phase it is very likely that missing requirements are discovered, and the analysis and design models need to be modified accordingly. At the same time, depending on the results from the testing activities, it is possible that some redesign and programming tasks would also be necessary. Hence, blending the V-Model in a "UP-like" process has the effect that the focus is more on activities and correctness, rather then in documents and artifacts. Moreover, it allows a *seamless* integration of V&V into a model-based development environment.

II. The Supporting Tool Set: Toward a Unified Development Environment

One of the major difficulties in building a set of supporting tools for a methodology is that tool builders rarely address the entire development life cycle. "Instead, they focus on a small set of activities, such as design or testing, and it is up to the user to integrate the selected tools into a complete development environment" [11]. Obviously, it would be desirable to have a unified development environment in which those tools collaborate to achieve the better result, without requiring designers to learn new notations every time. But it is not sufficient to just glue together existing modules. These modules have to be *designed* since the beginning to collaborate. The foundation of our tool set is based upon a *standard notation* and a well-suited *common representation* of all the relevant aspects of a software system that are amenable to automated analysis. As we briefly discuss in Section II-B, we exploit UML as such notation (even if we extend it a little), and we identify in the concept of *model abstraction* the suitable common representation for UML models shared by all tools.

A side-effect of this methodology is to propose UMLaid, a tool set that pervasively supports the following activities:

- Validation of a UML model and its textual specification during phase I through static checkers and model animators/simulators (*SVtool* and *SIMtool*);
- Verification for consistency and completeness of the UML model with respect to the associated OCL specification during phases I-II (*SVtool*);
- Forward and reverse engineering capability in order to keep models synchronized with code in phase III (*FEtool* and *REtool*);
- Model-based testing in phase III (*Ttool*);
- Evaluation of the overall design quality (aimed to identify potential design flaws), especially with respect to good design principles such as design patterns, quality metrics and design heuristics (*Qtool*). This activity is spread over phases I-II-III.

All these aspects have been conceived to be integrated in the overall life-cycle, as illustrated in Figure 1. We can

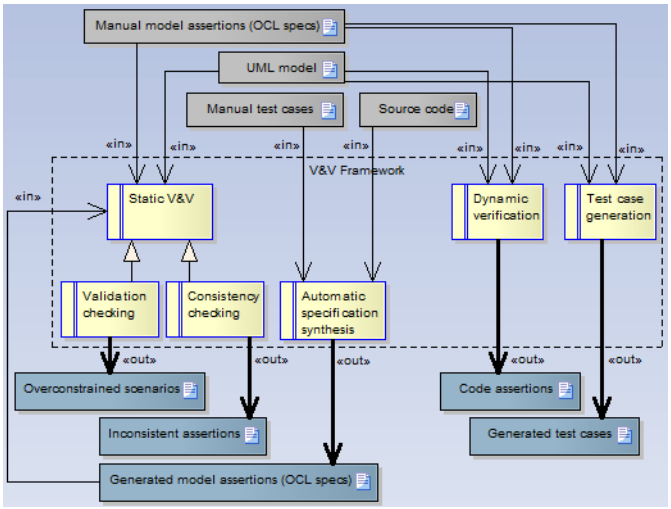

Figure 2. Our approach for V&V of UML models

use every single module independently, but better results are achieved when they communicate together. For example, we can imagine that while the designer is making some model refactoring (perhaps suggested by a quality critique) all tools silently check the changed structure against all the stated software constraints. Similarly, we can figure out the situation in which the verification tool highlights a bug in the model and this event triggers at first the intervention of the designer in charge to fix the bug, and subsequently the quality critique tool in order to find possible design flaws introduced by these changes. Here again, the *compositionality* is important as we can recompute *just* the things relevant to the changed parts.

In the next sections we provide an overview of the tools concerning the V&V and the quality critiquing activities.

A. Verification for correctness

In order to support the presented methodology for what concerns correctness we need a *scalable*, *model-based* tool set which integrates both static and dynamic verification techniques and which can work even in absence of the source code (See Figure 2). This tool set can also help further debugging activities, becoming an important tool for model validation.

Concerning dynamic verification approaches, in literature there are many tool proposals and prototypes that can fit in the needed scenario. For example, we can support the *traditional testing approach* providing both the UML model and the OCL specification as input and relying to Octopus[1] or similar tools in order to generate the assertion-testing code corresponding to OCL invariants, pre- and post-conditions. In this way, we realize an automated instrumentation of the source code supporting the test phase. Similarly, we can exploit the same input to support the *model-based testing approach*, where a set of test case is automatically generated by a tool such as LEIRIOS LTG/UML[2] in order to augment the test case base available to the tester. However, the testing tools eventually need the code in order to execute test cases, thus they cannot be exploited in the early stages of a project.

On the contrary, suitable *static* (semantics-based) tools would not need a complete system to work, but the static approach is in general undecidable. Hence, it cannot be fully automatized, making dynamic verification unavoidable if we want to cover the entire system. This is the reason why we are proposing a mixed approach that relies also on (dynamic) testing techniques, but applies them only on those parts which are not (successfully) covered by static methods. In this way we can shrink significantly the search space for the test case generation.

Unfortunately, there are few static tools compatible with a model-based software development environment suitable for UML. Many researchers have proposed static approaches based on model checking techniques, but they suffer from the state explosion problem and thus they cannot scale to real software system sizes. “Indeed, in software design complexity arises even in a single state machine, from the complex structure of its state, and model checkers can’t handle this structure” [6]. For this reason we have proposed in [2], [3] a different approach to achieve the mutual consistency of UML diagrams and OCL specifications, organized as follows.

- *Check of class diagrams and statecharts*: We check static semantic issues such as existence, visibility, cardinality constraints and dynamic semantic issues involving just classes such as the Liskov Substitution Principle [8].
- *Check of sequence diagrams with respect to class and statechart diagrams*: We integrate the information about states (coming from the statecharts) in all suitable points of a sequence diagram. Then we aim to guarantee that, by following the control flow on a sequence diagram, the state is strong enough to satisfy preconditions, postconditions, and class invariants of each method call.

We are well-aware of the manual effort required to provide a complete OCL specification even for a medium-size real system. However, in real-world situations this weakness can be alleviated by using the static engine to automatically synthesize (generate) some missing OCL specifications, starting from sequence diagrams, source code, and test case specifications, when they are available.

B. Verification for quality and CASE tools aiding software design

Software design should be regarded as an *instrument for reasoning* about a system. With the increase of the design complexity, we need tools that enable suitable reasoning and support experimentation with the design decisions. As an example of a typical issue which emerges in building a real-world UML model, consider the class diagrams in Figure 3 which contain a cycle. Cyclic dependencies across multiple diagrams are very difficult to catch *quickly*, but nonetheless they are important because make impossible to test in isolation each element involved in the cycle. The tool set should be able to provide an abstract (simplified) view of the diagrams,

[1]available at http://octopus.sourceforge.net/
[2]available at http://www.leirios.com/

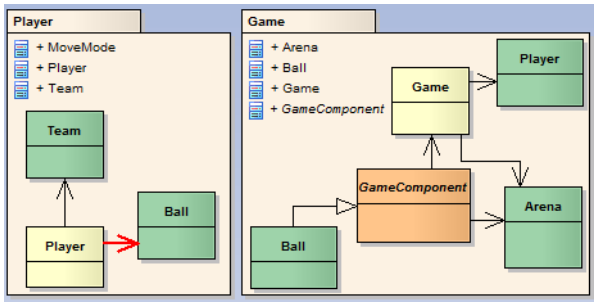

Figure 3. Cyclic dependency across multiple diagrams

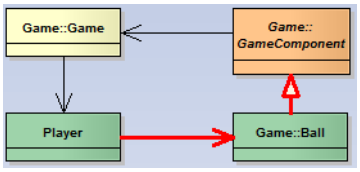

Figure 4. Model abstraction which retains only the cyclical dependencies

in which many classes and relations are hidden, but the components which form any pattern of interest (in this case, circular dependencies) are retained, as illustrated in Figure 4. This "reduced" diagram immediately communicates only the information relevant for a specific user-goal (i.e.,the designer that need to identify the order in which test subsystems). We call the result of this operation of throwing away information not related to a particular purpose a *model abstraction*, which is a very valuable structure because can spot clearly a possibly *critical piece of the design*.

One of the targets for model abstractions in our framework are *design patterns*, because they can be viewed as general laws to build a software quality model [5]. Nevertheless, there is a lack of tools automatizing the use of patterns to achieve well-designed pieces of software, to identify recurrent architectural forms, and to maintain programs. In order to automatically recognize a pattern structure from a UML model, we have to describe such a structure in a *formal* and *precise* way. UML is a semi-formal language so it is not always well suited to express every aspect of a design unambiguously. However, we do not want to invent a new formalism from the scratch. Hence in [1] we have proposed a visual notation obtained by adding to UML just three graphical elements which allow to express patterns with the needed level of formality.

Incidentally, this notation can be used to express with the needed formality also the antipatterns [4] and patterns like circular dependencies [3].

Figure 5 summarizes our approach for design quality critiquing. The tool set builds model abstractions in order to:

- provide the user with *design critiques* (such as instances of antipatterns or violated design heuristics);

[3]Technically, circular dependency is not an antipattern because it can represent essential aspects of an application domain. In this case, following the "code smells" terminology coined by Kent Beck, we will call this kind of pattern *smelly pattern*

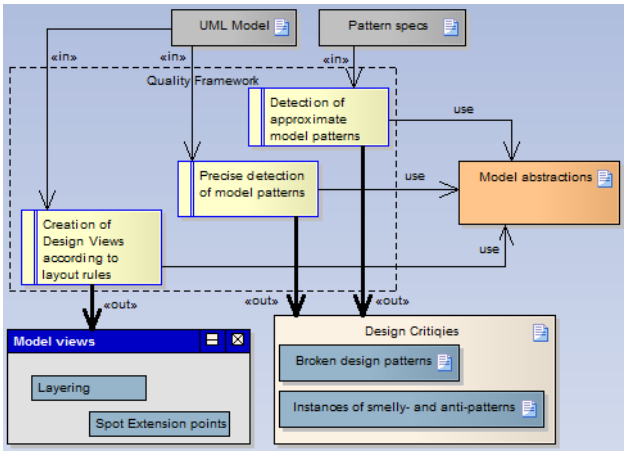

Figure 5. The most important quality-critiquing features of our framework

- automatically build alternative views of model diagrams (*model views*), more suited to specific tasks usually performed when the designer reasons about the software architecture.

III. Current developments and future works

Implementing an integrated set of tools which work together in a complex framework is a huge task. We have almost finished to develop a prototype for both $SVtool$ and $Qtool$ based on these ideas. As a future work, we plan to complete the $Qtool$ with a module that works on the diagram's layout in order to automatically synthesize meaningful model views. On the verification side, instead, we plan to integrate suitable $Ttool$ modules.

References

[1] D. Ballis, A. Baruzzo, and M. Comini. A Minimalist Visual Notation for Design Patterns and Antipatterns. Available on our homepage., 2007.

[2] A. Baruzzo and M. Comini. Static Verification of UML Model Consistency. In D. Hearnden, J. G. Süß, B. Baudry, and N. Rapin, editors, *MoDeV^{2}a: Model Development, Validation and Verification*, pages 111–126. University of Queensland, Le Commissariat à l'Energie Atomique - CEA, October 2006.

[3] A. Baruzzo and M. Comini. A Framework for Computer Aided Consistency Verification of UML Models. Available on our homepage., 2007.

[4] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, 1998.

[5] Y-G. Guhneuc, J.J. Guyomarc'h, K. Khosravi, and H. Sahraoui. Design Patterns as Laws of Quality. Technical Report Technical Report, Department of Informatics and Operations Research, University of Montreal, Quebec, Canada, France, 2006.

[6] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., 2006.

[7] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process, The (2nd Edition)*. Addison-Wesley Professional, 1999.

[8] B. Liskov and J. M. Wing. Family values: A behavioral notion of subtyping. Technical report, Pittsburgh, PA, USA, 1993.

[9] S. Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[10] German Ministry of Defense. V-model: Software lifecycle process model, 1992.

[11] S.L. Pfleeger and J.M. Atlee. *Software Engineering: Theory and Practice 3/E*. Pearson, 2006.