IN3 Working Paper Series
INTERNET INTERDISCIPLINARY INSTITUTE
ISSN 2013-8644

Bounded Verification of Software Models

Challenges and Opportunities

Robert Clarisó Viladrosa (rclariso@uoc.edu)
Universitat Oberta de Catalunya

Working Paper

Working Paper Series WP14-008

Research group: GRES-UOC
Research group coordinator: Robert Clarisó (Universitat Oberta de Catalunya)

Submitted in: December 2014
Accepted in: December 2014
Published in: December 2014

Universitat Oberta de Catalunya
IN3 INTERNET INTERDISCIPLINARY INSTITUTE
http://in3.uoc.edu

Internet Interdisciplinary Institute (IN3)
http://www.in3.uoc.edu
Edifici Barcelona Growth Center
c/ Roc Boronat, 117
08018 Barcelona
Espanya
Tel. 93 4505200

Universitat Oberta de Catalunya (UOC)
http://www.uoc.edu/
Av. Tibidabo, 39-43
08035 Barcelona
Espanya
Tel. 93 253 23 00

Table of contents

Bounded Verification of Software Models
Challenges and Opportunities

Robert Clarisó Viladrosa (rclariso@uoc.edu)
Universitat Oberta de Catalunya

Recommended citation:

CLARISÓ, Robert (2014). "Bounded Verification of Software Models: Challenges and Opportunities" [online working paper]. (Working Paper Series; WP14-008). IN3 Working Paper Series. IN3 (UOC). [Accessed: dd/mm/yy]. <http://journals.uoc.edu/ojs/index.php/in3-working-paper-series/article/view/n14-clariso/n14-clariso>

Abstract

Ensuring the absence of bugs in a software system is an important but very challenging problem. Early error detection within the development process reduces the cost of finding and fixing defects. Thus, the analysis of software models can improve its final quality and reduce its development costs.

A promising research direction in this field is the use of boolean satisfiability (SAT) or constraint programming (CP) solvers to perform bounded verification. Bounded verification consists in formally checking the absence of errors within a finite space defined as a parameter of the analysis. This type of analysis is usually fast in practice

and provides informative feedback. However, it is computationally complex in general and lacks conclusive results outside the verification bounds provided as parameters.

In this paper, we discuss recent trends and results in the application of bounded verification to a specific field within software engineering: the analysis of models of a software system. Furthermore, we will discuss promising contributions that can build upon previous work to improve its applicability in practice within the software industry.

Keywords
Software Engineering, Software Quality, Formal Methods, Formal Verification, Model-Driven Engineering (MDE), Constraint Programming, Boolean Satisfiability (SAT), UML, OCL, Model Transformations.

Introduction

Why does software fail? The root cause for many software failures, defects, glitches and *bugs* lies in the sheer size and complexity of most software systems. Errors may be caused by faulty requirements or they may be introduced later during the implementation. In any case, the goal of Software Engineering is to provide the means to prevent defects, detect them, fix them or reduce their impact throughout the development process.

Given the complexity of software systems, automation is required in order to assist software engineers in their endeavor. Automation is not only a matter of devising a suitable algorithm: there must be an available implementation (a tool) which is capable of analysing the software system system under study. In this context, several problems arise, such as the usability of the tool, its maturity and documentation and its efficiency. The matter of efficiency is particularly critical, as any realistic software application will be very large. Hence, being able to cope with this size is a necessary requirement of any software engineering tool.

In this paper, we consider the efficiency of software engineering tools addressing quality issues. In particular, we consider a specific development methodology (*model-driven development*) and one particular quality assurance technique (*formal verification using bounded methods*). The paper is devoted to studying how to improve the efficiency of bounded verification in model-driven development to ensure its scalability to realistic examples.

The remainder of this paper is structured as follows. Section 1 introduces the quality concerns in software engineering using model-driven development. Section 2 describes the state of the art in formal verification for model-driven development. Section 3 analyzes several techniques that can improve the efficiency of verification techniques. Finally, Section 4 concludes the paper.

1. Problem statement

1.1. Software quality

Quality is one of the main concerns within Software Engineering. In this context, software defects can be studied from two different perspectives: *verification*, determining if a software system is free of errors ("Is the product right?"); and *validation*, studying whether a software system meets the requirements and expectations of all stakeholders ("Is it the right product?").

Focusing on verification, there are several potential approaches to check the absence of errors. For example, testing checks the correct operation of a system in a finite collection of test cases. An alternative is the use of *formal methods* (D'Silva et al, 2008) to prove the correct operation of a system by rigorously analyzing a mathematical representation of that system. Even though more computationally expensive than testing, formal verification can detect errors in complex corner cases which may be difficult to test. Thus, formal verification provides a very high level of quality assurance. The problem of producing correct software through formal verification was considered in 2006 a challenge for the entire international scientific community (Jones et al, 2006).

1.2. Software models

A model is an abstract and simplified representation of reality defined with a specific purpose. In the case of software, several types of models can be defined, where each one describes one facet of the system under development using either a textual or visual syntax. Thus, a model may be a diagram describing the information base of an information system, its graphical interface or its response to events from the user.

In particular, models can be classified into *static* or *structural* (dealing with information or structure, e.g. a class diagram) and *dynamic* or *behavioral* models (dealing with behavior or execution, e.g. a sequence diagram). Many different notations have been proposed to describe both static or dynamic models, e.g. UML (Unified Modeling Language), OCL (Object Constraint Language), SysML (Systems Modeling Language), SBVR (Semantics of Business Vocabulary and Business Rules), IFML (Interaction Flow Modeling Language), Alloy, B, Z, …

The great advantage of models lies in their abstraction power: the ability to provide elegant and concise descriptions of an aspect of the system while hiding irrelevant aspects. Hence, models can be targeted at specific activities, e.g. documentation of design decisions; communication among team members and stakeholders; simulation of the operation of the system; verification, validation and testing of relevant properties about the system; reverse-engineering of legacy systems; or transformation of the model into an executable form (e.g. code generation).

1.3. Quality of software models

Since 1987, a recurring trend in software defects has been observed empirically in many software development projects:

> *"errors are most frequent during the requirements and design activities and are the more expensive the later they are removed"* (Boehm and Basili, 2001 ; Endes and Rombach, 2003).

Thus, providing early error detection has become an important concern in order to improve software quality and reduce development costs. This requires analyzing models of the software system under development before reaching the implementation stage. There is a strong synergy among this trend and an emerging paradigm (Boehm, 2006) to improve the software development process through the use of models: model-driven engineering (MDE) (Boehm and Basili, 2001; France and Rumpe, 2007).

The goal of MDE is "to significantly reduce the time, cost, and effort required to develop complex software intensive systems that meet stringent quality requirements through use of models that are fit-for-purpose" (France and Rumpe, 2013). This paradigm is already being applied in industrial settings with satisfactory results (Hutchinson et al, 2011).

MDE advocates for using models as the central artifact in the development process: instead of developing code, developers create high-level models which are then progressively and automatically translated into the final implementation. This translation process, central to MDE, is called *model transformation*. Automation helps to improve productivity and improve the quality of software by avoiding low-level mechanical and repetitive tasks which may be error-prone. Furthermore, it is possible to define domain-specific languages (DSL) using high-level notations to let domain experts without a software engineering background design their own applications.

The key to the success of MDE is the availability of tools that allow the automation of key aspects of model management: editors, simulators, code generators, … Among them is the quality assurance of the final software system. That is, it should be

possible to detect errors, inconsistencies or incomplete information in the models; help the developer locate and correct those errors; and ensure that model transformations are correct in the sense that they do not introduce errors in the final implementation.

2. State of the art

In this section, we briefly examine the state of the art in formal verification applied to model-driven engineering.

Given that MDE involves producing an implementation through the gradual transformation of models, the correctness of models is a primary concern as model defects directly map into software errors. As models are describing the system at a high-level of abstraction, checking the correctness of models can be simpler than checking the same properties at level of source code. In return, it is necessary to check the correctness of both models and model transformations (Dubios et al, 2013).

2.1. Correctness properties

In order to define a formal verification problem, one of the first notions that needs to be established is: what will be considered an *incorrect* behavior? The type of correctness property being checked has a profound impact on the decidability and complexity of the problem.

First of all, designers can define their own custom correctness properties that need to be checked. In addition to these custom properties, there are some fundamental assumptions about the well-formedness of a model whose validity is taken for granted in almost any system.

For instance, a desirable feature of static models is *consistency*: the ability to simultaneously satisfy all the restrictions described in the model. Consistency can be checked at two levels: among the different elements in a single model (intra-model consistency, e.g. lack of contradictions) or among several models describing different perspectives of the same system (inter-model consistency, e.g. all references to a model element comply with its declaration). Besides detecting contradictions and incorrect references, it is also useful to detect other types of “bad smells” such as redundancies, as they could be a symptom of a more serious defect.

Regarding dynamic models and model transformations, the properties of interest consider the evolution of the system state. Sample properties include checking the *executability* of a fragment of the model (e.g. the ability to satisfy its precondition),

IN3 Working Paper Series is a monograph series promoted by the Internet Interdisciplinary Institute (IN3) of the UOC
IN3 Working Paper Series (2010) | ISSN 2013-8644 | http://in3-working-paper-series.uoc.edu

safety properties such as the preservation of integrity constraints, reachability properties such as the ability to get to a specific system state, liveness properties or custom temporal properties about the execution of the system.

2.2. Formal vs informal modeling notations

Many modeling notations are mainly used to document and explain the architecture/operation of a software system. Hence, a clear and understandable graphical syntax is an asset provided by many modeling languages. However, some modeling notations are not created with the notion of automatically generating the final implementation from the model. Hence, the semantics of each modeling may be defined using natural language (to ease understandability) rather than a more precise formal notation that can eliminate ambiguity.

The first step towards allowing the automatic analysis of a software model is providing a formal semantics of the modeling notation. Some notations like B (Abrial, 1996), Z (Spivey, 1992) or Alloy (Jackson, 2006) are designed specifically for analysis purposes, and hence come with a built-in formal semantics. Meanwhile, other general purpose notations like UML require a previous formalization step before committing to analysis (Broy and Cengarle, 2011).

These two approaches differ in the amount of formal method expertise required by designers who will use verification tools. In formal modeling notations, the designers need to be aware of the formal semantics in order to faithfully model the system under analysis and take advantage of specialized provers (Leuschel and Butler, 2008). Meanwhile, other tools are based on the use of *hidden formal methods* (Hussmann, 1995 ; Berry, 1999), where designers employ a pragmatic modeling notation which has a hidden mathematical foundation that allows a rigorous analysis. Clearly, the latter paradigm is preferred from a point of view of a wide adoption in an industrial context.

2.3. Verification techniques

Many different formalisms have been employed in the formal verification of models (Wille et al, 2013 ; González and Cabot, 2014) and model transformations (Rahim and Whittle, 2013):

- constraint programming (CP) (Cadoli et al, 2004; Malgouyres and Motet, 2006; Horváth and Varró, 2012 ; Cabot et al, 2014)
- description logics (Berardi et al, 2005 ; Balaban and Maraee, 2013)
- term rewriting (Clavel and Egea, 2006 ; Romero et al, 2007)

- relational logic (Baresi and Spoletini, 2006 ; Jackson, 2006 ; Anastasakis et al, 2010 ; Maoz et al ; 2011 ; Kuhlmann and Gogolla, 2012)
- higher-order logic (Brucker and Wolff, 2008)
- SAT Modulo Theories (SMT) (Clavel et al, 2009; Semeráth et al, 2013),
- query containment (Queralt and Teniente, 2012)
- linear programming (Balaban and Maraee, 2013)
- genetic algorithms (Ali et al, 2013)

These approaches can be classified depending on their approach towards decidability:

1. Approaches that restrict the expressiveness of model elements and constraints in order to deal with a decidable verification problem where reasoning is efficient, e.g. (Berardi et al, 2005 ; Balaban and Maraee, 2013).
2. Approaches that allow a high-level of expressiveness in model elements and constraints and deal with undecidable verification problems through interactive provers, e.g. (Brucker and Wolff, 2008 ; Leuschel and Butler, 2008), or incomplete methods, e.g. (Ali et al, 2013).
3. Approaches that allow a high-level of expressiveness in model elements and constraints and deal with undecidable verification problems through bounded verification, e.g. (Jackson, 2006 ; Kuhlmann and Gogolla, 2012 ; Cabot et al, 2014).

Advances in SAT- and CP-solver technology (Bourdeaux et al, 2006) have made techniques in category (3) very promising, as it is possible to achieve automation and efficient reasoning without sacrificing expressiveness. If a solution is found within the verification bounds, these benefits come with no drawback. Furthermore, the solutions computed by these solvers can be used as test input data, i.e. performing model-based testing instead of verification. However, the absence of faults does not guarantee a correct behavior outside the verification bounds.

3. Scalability of Bounded Verification

One of the main challenges of formal verification in industrial contexts is *scalability*: the ability to check properties in real-world models, which may have a size ranging

from hundreds to thousands of elements. A method which does not scale will work well in toy examples but it will not be applicable in practice.

Scalability is an issue because most verification techniques have a high computational complexity and suffer from to so called *state explosion*, an exponential growth rate of the solution space in terms of the size of the input. Part of this complexity is inherent to the problem, so it cannot be overcome in general. Instead, the goal is identifying solutions that allow methods to scale up to practical examples, taking into account that methods cannot scale indefinitely.

In terms of bounded verification, scalability can be studied from two different perspectives. On one hand, we are interested in using techniques that can deal with large methods with many model elements. On the other hand, we are also interested in approaches that can support wide verification bounds to achieve a larger degree of confidence in the outcome of the verification. That is, it may be feasible to verify a large model within a tiny solution space, but this may rule out real faults that lay outside our bounds.

In the following, we discuss four approaches that can improve the efficiency of bounded verification techniques for software models: (1) automatic bound refinement, (2) tool-supported bound selection, (3) smart model slicing and (4) automatic selection of the most promising verification technique.

3.1. Automatic bound refinement

The main shortcoming of bounded verification is the lack of conclusive results outside of the verification bounds. Choosing suitable verification bounds is a non-trivial process as there is a trade-off between the verification time (faster for smaller domains) and the confidence in the result (better for larger domains).

Unfortunately, setting the search space boundaries has proven itself to be a major limiting factor, since existing tools provide little support on this, either by setting inadequate default values; or by forcing users to manually define these boundaries, which is impractical when dealing with large models.

In this sense, bound selection is possibly the last barrier of entry for non-experts. The cause of this lack of tool support is that choosing optimal bounds automatically is as complex as the verification problem itself, so the use of heuristics or approximate methods is required. For example, the *small scope hypothesis* claims that a large amount of faults can be detected by inspecting a small domain. Hence, many tools advocate for an incremental scoping strategy: start with small domains to get feedback quickly and progressively increase the domain size in later executions until a fault is

detected or we achieve a sufficient level of confidence in the result. However, beyond that, designers must select domains on their own.

In tools using low-level formalisms such as SAT, verification bounds affect the transformation from the model to the verification formalism: larger domains require more boolean variables for the encoding and therefore produce larger formulas. Even if part of the domain can be discarded due to the constraints in the model, the analysis will still be slower due to the extra variables and clauses in the formula (Ganai et al, 2002). Furthermore, the fact that part of the domain can be discarded may be obvious at a high level of abstraction (e.g. considering the interaction among different constraints) but it can be time consuming to detect at the level of a boolean formula. For these reasons, analyzing models to tighten verification bounds can improve the efficiency of bounded verification solvers (Rosner et al, 2013).

The purpose of automatic bound refinement is two-fold:

- Automatically infer verification bounds from the model under analysis and the property being checked.
- Improve the bounds provided by the designer, either by removing irrelevant values or by suggesting values of interest.

An automatic procedure for refining bounds operates in the following way:

- First, the model and property under analysis can be abstracted as restrictions on the verification bounds. This abstraction relies on the static analysis framework of abstract interpretation (Cousot and Cousot, 1977) to define safe abstraction procedures. An example of this kind of analysis can be found in (Yu et al, 2007), where OCL integrity constraints are abstracted as properties on the size of collections.
- Second, this system of constraints can be analyzed to (i) efficiently prune unproductive values from verification bounds and (ii) detect potentially promising bounds, e.g. corner or base cases for those constraints that should be considered. A promising strategy to perform this analysis is *constraint and bound propagation*, a family of techniques from the constraint programming field (Apt, 2003; Bourdeaux et al, 2011). Propagation analyzes a constraint satisfaction problem and infers implicit information about its consistency. This inferred information can be used to tighten the problem, either by pruning bounds, strengthening constraints or introducing new ones.

There are several research challenges in the application of this methodology: the ability to abstract expressive models, proving the soundness of the proposed abstraction, dealing with dynamic constructs (e.g. pre- and postconditions or imperative statements) and selecting the most suitable propagation algorithm (exact or heuristic), among others.

3.2. Tool-supported bound selection

The proposed bound refinement approach is based on abstracting of the model to take advantage of implicit constraints that restrict the verification bounds. Nevertheless, the amount of information that can be inferred will depend on the model (models with many constraints will be more amenable to analysis) and the size of the initial bounds (smaller bounds will be more susceptible to propagation). It may be the case that automatic methods fail to provide a sufficient improvement in the verification bounds.

Another take on bound refinement is providing tools that analyze the model and guide users in the choice of verification bounds. That is, providing tools to support *interactive bound selection* on top of fully automated tools.

The rationale is that designers have no clue about which choices of domain bounds are most relevant for the verification problem. It may be the case that one variable is critical to the verification and the size of its domain has a large impact in the size of the verification space, while the domain of other variables (a) does not have a large penalty on the verification time or (b) is fully determined by the domain of other variables, and hence no choice is required at all.

More precisely, the goal of these tools for bound refinement is two-fold:

- Provide interactive procedures to guide users in the choice of bounds while maximizing the amount of information that is inferred automatically throughout the process.
- Help designers estimate whether the verification bounds are sufficiently large to achieve the desired level of confidence in the result.

That is, the goal is minimizing the number of bound choices required from designers and to reduce the degrees of freedom in each choice. Though similar to the choice of *variable ordering* in constraint satisfaction problems (Apt, 2003): deciding which variable should be assigned a value next. This problem deals with variable bounds instead of values, but it can take advantage of heuristics proposed in this context.

3.3. Smart model slicing

A family of techniques that can be used to accelerate formal verification is based on the notion of *slicing*. Slicing was originally proposed in the context of source code (Tip, 1995) and has later been applied to other domains such as hardware description languages (Clarke et al, 2002) or software models (Wu and Yi, 2004 ; Brückner and

Wherheim, 2005 ; Uzuncaova and Khurshid, 2007 ; Lallchandani and Mall, 2010 ; Shaikh et al, 2010 ; Juliand et al, 2013 ; Seiter et al, 2013).

A *program slice* consists of those parts of a program that can potentially affect (or be affected by) a *slicing criterion*. In other words, a slice preserves the behavior of the original program with respect to the slicing criterion while being potentially simpler. Reduced complexity can facilitate program comprehension, software maintenance, testing and formal verification.

Model slicing for formal verification (Wu and Yi, 2004 ; Brückner and Wherheim, 2005 ; Uzuncaova and Khurshid, 2007 ; Shaikh et al, 2010 ; Seiter et al, 2013) is property driven: the slicing criterion is established by the correctness property being checked. The outcome is a set of model slices that (i) abstract features of the model that are irrelevant to this property and (ii) partition the original model into smaller submodels where the property can be checked independently. The specific syntax and semantics of the input modeling notation drives the definition of the slicing procedure, e.g. Alloy (Uzuncaova and Khurshid, 2007), UML/OCL (Shaikh et al, 2010 ; Seiter et al, 2013), Z (Wu and Yi, 2004) or Object-Z (Brückner and Wherheim, 2005).

An important requirement of the slicing procedure is ensuring that verifying model slices is equivalent to verifying the original model. This requirement forces the slicing procedure to be *conservative*, i.e. include any model element that may potentially affect the correctness property directly or indirectly. Being conservative is required to ensure the validity of slicing, but it also reduces the opportunities for simplification.

A promising area of development in this context is the development of *smart slicing criteria* that can overcome the limitations of conservative methods using *graph partitioning* (Bichot and Siarry, 2013) and *graph separators* (Shen and Liang, 1977).

Slicing procedures usually work on a graph abstraction of the original model, where vertices are model elements and edges represent dependencies among them, e.g. integrity constraints. After identifying the relevant model elements (the slicing criterion), the slicing procedure iteratively adds adjacent model elements to the slice until reaching a fixpoint. As a new step, graph partitioning (resp. separators) can be used to identify loosely coupled subgraphs within the final slice, which can be partitioned by cutting edges (removing vertices). However, additional constraints need to be added to each subgraph in order to reconcile the results of their verification.

Similar approaches have been applied to improve the performance of SAT and constraint solvers (Salido and Barber, 2006), e.g. to allow analyzing each subproblem in parallel. These techniques can be more successful at the model level than at the solver level for several reasons:

- Being aware of the semantics of the model allows us to identify more opportunities for partitioning, to recognize spurious dependencies and to properly assign a weight to each dependency according to its restrictiveness.

- Models are much smaller and therefore more precise graph partitioning techniques are available, e.g. (Shen and Liang, 1997). Moreover, the overhead introduced by partitioning is also smaller.

3.4. Automated selection of most suitable verification technique

As discussed in Section 2.3, many different methods have been used in the formal verification of models and model transformations. Each method has different advantages and limitations in terms of termination, completeness, efficiency and expressiveness (the set of supported constructs in the input artifact).

However, all the works presented in the literature have been studied in isolation. For example, given that the characteristics of each method are so different, it does not make sense to compare the efficiency of a SAT-based method with an interactive theorem prover. Thus, little effort has been spent in categorizing input models to find out which is the most suitable verification method for families of input models or, even better, for a specific input model.

In this way, a promising research direction is the definition of:

- An exact *viability test* that can check whether a method is capable of reasoning about the input artifact, i.e. it supports all the constructs used in the input model or transformation. This test should take into account potential refactorings of the input artifact that reduce its expressiveness without affecting its semantics. An example of this type of refactoring could be replacing an *n*-ary association in a class diagram by a class plus *n* binary associations.
- A heuristic *fitness metric* that predicts the performance of a method in the verification of an input artifact. This heuristic should take into account different factors such as the size of the artifact, the type of constraints it contains, the average performance of the method on similar artifacts, ... For instance, CP- or search-based methods may be preferred to SAT-based strategies in models with complex arithmetic constraints (Bourdeaux et al, 2006).

To this end, it is necessary to formalize the features of each verification technique (e.g. in an ontology) in order to be classify each method, analyze each input model and check whether a particular method is able to analyze a specific property of a given model and, if it is, whether the use of that method is suitable for the kind of constraints included in the model.

IN3 Working Paper Series is a monograph series promoted by the Internet Interdisciplinary Institute (IN3) of the UOC
IN3 Working Paper Series (2010) | ISSN 2013-8644 | http://in3-working-paper-series.uoc.edu

4. Conclusions

There are many different approaches aiming to improve the quality of software. One of them, Model-Driven Engineering, attempts to lift the focus of development to a higher level of abstraction, dealing with models rather than source code. Faults in software models can be detected through several techniques such as testing or formal verification.

Many formal verification problems that can be studied on software models are undecidable. Thus, analysis procedures must either rely on user guidance, use approximation or impose restrictions on the software model and the properties under analysis.

In this context, bounded verification methods offer a convenient trade-off: automatic procedures able to verify properties, but whose answer is only valid within a finite universe of discourse. Their output (or its lack thereof) can be used to identify faults or gain sufficient confidence in the correctness of the model.

Bounded verification tools rely on efficient solvers to compute examples and counterexamples of the properties being analyzed. SAT solvers or Constraint Programming can be used to this end. However, even though these approaches are efficient and use many optimizations, their scalability still does not match the needs of real-world models. Hence, additional techniques must be employed to improve their scalability and applicability in the software industry.

This paper proposes several techniques to advance the state-of-the-art in bounded verification of software models: (1) automatic bound refinement, (2) tool-supported bound selection, (3) smart model slicing and (4) automated selection of the best verification technique. The combination of these methods, which can be seen as a pre-processing stage prior to analysis, can improve the efficiency of existing tools. As a whole, they constitute a research agenda towards improving the scalability of bounded verification in MDE.

Bibliographic references

AB RAHIM, L..; WHITTLE, J. (2013). A survey of approaches for verifying model transformations. Software and Systems Modeling, 1–26.

ABRIAL, J.R. (1996). The B-Book: Assigning programs to meanings. Cambridge University Press.

ALI, S.; IQBAL, M. Z.; ARCURI, A.; BRIAND, L. C. (2013). Generating test data from OCL constraints with search techniques. IEEE Transactions on Software Engineering, 39(10):1376–1402.

ANASTASAKIS, K.; BORDBAR, B.; GEORG, G.; RAY, I. (2010). On challenges of model transformation from UML to Alloy. Software and System Modeling, 9(1):69–86.

APT, K. R. (2003). Principles of constraint programming. Cambridge University Press.

BALABAN, M.; MARAEE, A. (2008). A UML-based method for deciding finite satisfiability in Description Logics. In Int. Workshop on Description Logics (DL), (CEUR Workshop Proceedings; 353)

BALABAN, M.; MARAEE, A. (2013). Finite satisfiability of UML class diagrams with constrained class hierarchy. ACM Transaction on Software Engineering Methodology, 22(3):24.

BARESI, L.; SPOLETINI, P. (2006). On the use of Alloy to analyze graph transformation systems. In Int. Conf. on Graph Transformations (ICGT), volume, pages 306–320. Springer. (Lecture Notes in Computer Science; 4178)

BERARDI, D.; CALVANESE, D.; DE GIACOMO, G. (2005). Reasoning on UML class diagrams. Artificial Intelligence, 168:70–118.

BERRY, D. M. (1999). Formal methods: The very idea – Some thoughts about why they work when they work. Electronic Notes on Theoretical Computer Science, 25:10–22.

BICHOT, C.E.; SIARRY, P. (2013). Graph Partitioning. Wiley.

BOEHM, B. W. (2006). Barry Boehm. A view of 20th and 21st century software engineering. In Int. Conf. on Software Engineering (ICSE), pages 12–29, New York, NY, USA: ACM.

BOEHM, B. W.; BASILI, V. R. (2001). Software defect reduction top 10 list. IEEE Computer, 34(1):135–137.

BOURDEAUX, L.; HAMADI, Y.; ZHANG, L. (2006) Propositional Satisfiability and Constraint Programming: A comparative survey. ACM Computing Surveys, 38(4).

BOURDEAUX, L.; KATSIRELOS, G.; NARODYTSKA, N.; VARDI, M. Y. (2011). The complexity of integer bound propagation. Journal of Artificial Intelligence Research, 40:657–676.

BRAMBILLA, M; CABOT, J.; WIMMER, J. (2012). Model-Driven Software Engineering in Practice. Morgan Claypool Publishers.

BROY, M.; CENGARLE, M. V. (2011). UML formal semantics: lessons learned. Software & Systems Modeling, 10(4):441-446.

BRUCKER, A.D.; WOLFF, B. (2008). HOL-OCL: A formal proof environment for UML/OCL. In Int. Conf. on Fundamental Approaches to Software Engineering (FASE), pages 97–100. Springer. (Lecture Notes in Computer Science; 4961).

BRÜCKNER, I.; WEHRHEIM, H (2005). Slicing Object-Z specifications for verification. In Formal Specification and Development in Z and B (ZB), pages 414–433. Springer. (Lecture Notes in Computer Science; 3455).

BUNDALA, D.; OUAKNINE, J.; WORRELL, J. (2012). On the magnitude of completeness thresholds in Bounded Model Checking. In Symposium on Logic in Computer Science (LICS), pages 155–164. IEEE.

CABOT, J.; CLARISÓ, R.; RIERA, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software, 93:1–23.

CADOLI, M.; CALVANESE, D.; DE GIACOMO, G.; MANCINI, T. (2004). Finite satisfiability of UML class diagrams by Constraint Programming. In Int. Workshop on Description Logics (DL). (CEUR Workshop Proceedings; 104).

CLARKE, E. M.; EMERSON, E. A.; SIFAKIS, J. (2009). Model checking: algorithmic verification and debugging. Communications of the ACM, 52(11):74–84.

CLARKE, E. M.; FUJITA, M.; RAJAN, S. P.; REPS, T. W. (2002) Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing for VHDL. STTT, 4(1):125–137.

CLARKE, E. M.; KROENING, D.; OUAKNINE, J.; STRICHMAN, O. (2005). Computational challenges in bounded model checking. STTT, 7(2):174–183.

CLAVEL, M.; EGEA, M. (2006). ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In Int. Conf. on Algebraic Methodology and Software Technology (AMAST), pages 368–373. Springer. (Lecture Notes in Computer Science; 4019)

CLAVEL, M.; EGEA, M.; GARCÍA DE DIOS, M. A. (2009). Checking unsatisfiability for OCL constraints. Electronic Communication of the European Association of Software Science and Technology (ECEASST), 24.

COUSOT, P.; COUSOT, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Symposium on Principles of Programming Languages (POPL), pages 238–252. ACM.

D'SILVA, V; KROENING, D.; WEISSENBACHER, G. (2008). A survey of automated techniques for formal software verification. IEEE Transactions on CAD of Integrated Circuits and Systems, 27(7):1165–1178.

DUBOIS, C.; FAMELIS, M.; GOGOLLA, M.; NÓBREGA, L.; OBER, I.; SEIDL, M.; VÖLTER, M. (2013). Research questions for validation and verification in the context of model-based engineering. In Int. Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa), pages 67–76. (CEUR Workshop Proceedings; 1069)

ENDES, A.; ROMBACH, D. (2003). A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Fraunhofer IESE Series on Software Engineering. Addison-Wesley.

FRANCE, R. B.; RUMPE, B. (2007). Model-driven development of complex software: A research roadmap. In Future of Software Engineering (FOSE), pages 37–54, Washington, DC, USA. IEEE Computer Society.

FRANCE, R. B.; RUMPE, B. (2013). The evolution of modeling research challenges. Software and System Modeling, 12(2):223–225.

GANAI, M. K; ASHAR, P.; GUPTA, A.; ZHANG, L.; MALIK, S. (2002). Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In Design Automation Conference (DAC), pages 747–750. ACM.

GOGOLLA, M.; BÜTTNER, F.; CABOT, J. (2013). Initiating a benchmark for UML and OCL analysis tools. In Int. Conf. on Tests and Proofs (TAP), pages 115–132. Springer. (Lecture Notes in Computer Science; 7942).

GONZÁLEZ, C.A.; CABOT, J. (2014). Formal verification of static software models in MDE: A systematic review. Information and Software Technology, 56(8):821–838, 2014.

GUPTA, G.; PONTELLI, E. (2002). Specification, implementation, and verification of domain specific languages: A logic programming-based approach. In Computational Logic: Logic Programming and Beyond, pages 211–239. Springer. (Lecture Notes in Computer Science; 2407)

HEVNER, A.R.; MARCH, S. T.; PARK, J.; RAM, S. (2004). Design Science in information systems research. MIS Quarterly, 28(1):75–105.

HORVÁTH, A.; VARRÓ, D. (2012). Dynamic constraint satisfaction problems over models. Software and System Modeling, 11(3):385–408.

HUSSMAN, H. (1995). Indirect use of formal methods in software engineering. In Workshop on Formal Methods Application in Software Engineering Practice, pages 126–133.

HUTCHINSON, J.; WHITTLE, J.; ROUNCEFIELD, M.; KRISTOFFERSEN, S. (2011). Empirical assessment of MDE in industry. In Int. Conf. on Software Engineering (ICSE), pages 471–480. ACM.

JACKSON, D. (2006). Software Abstractions: Logic, Language, and Analysis. The MIT Press.

JAYARAMAN, K.; TRIPUNITARA, M. V.; GANESH, V.; RINARD, M. C.; CHAPIN, S. J. (2013). Mohawk: Abstraction-refinement and bound-estimation for verifying access control policies. ACM Transactions on Information Systems Security, 15(4):18.

JONES, C.B.; O'HEARN, P. W.; WOODCOCK, J. (2006). Verified software: A grand challenge. IEEE Computer, 39(4):93–95.

JULLIAND, J.; STOULS, N.; BÚE, P.-C.; MASSON, P.-A.; (2013) Jacques Julliand, Nicolas Stouls, Pierre-Christophe Bu´e, and Pierre-Alain Masson. B model slicing and predicate abstraction to generate tests. Software Quality Journal, 21(1):127–158.

KUHLMANN, M.; GOGOLLA, M. (2012). Mirco Kuhlmann and Martin Gogolla. Strengthening SAT-Based validation of UML/OCL models by representing collections as relations. In European Conf. on Modelling Foundations and Applications (ECMFA), pages 32–48. Springer, 2012. (Lecture Notes in Computer Science; 7349)

LALLCHANDANI, J. T.; MALL, R. (2010). Integrated state-based dynamic slicing technique for UML models. IET Software, 4(1):55–78.

LEUSCHEL, M.; BUTLER, M. J. (2008). ProB: an automated analysis toolset for the B method. STTT, 10(2):185–203.

MALGOUYRES, H.; MOTET, G. (2006). A UML model consistency verification approach based on meta-modeling formalization. In Symp. on Applied Computing (SAC), pages 1804–1809. ACM Press.

MALIK, S.; ZHANG, L. (2009). Boolean satisfiability: from theoretical hardness to practical success. Commun. ACM, 52(8):76–82.

MAOZ, S.; RINGERT, J. O.; RUMPE, B. (2011). CD2Alloy: Class diagrams analysis using Alloy revisited. In Int. Conf. on Model Driven Engineering Languages and Systems (MODELS), volume 6981 of LNCS, pages 592–607. Springer.

PEFFERS, K.; TUUNANEN, T.; ROTHENBERGER, M. A.; CHATTERJEE, S. (2008). A Design Science research methodology for information systems research. J. of Management Information Systems, 24(3):45–77.

QUERALT, A.; TENIENTE, E. (2012). Verification and validation of UML conceptual schemas with OCL constraints. ACM Transactions on Software Engineering Methodology, 21(2):13.

ROMERO, J. R.; RIVERA, J. E.; DURÁN, F.; VALLECILLO, A. (2007). Formal and tool support for Model Driven Engineering with Maude. Journal of Object Technology, 6(9):187–207.

ROSNER, N.; SIDDIQUI, J. H.; AGUIRRE, N.; KHURSHID, S; FRIAS, M. F. (2013). Ranger: Parallel analysis of Alloy models by range partitioning. In Int. Conf. on Automated Software Engineering (ASE), pages 147–157. IEEE.

SALIDO, M. A.; BARBER, F. (2006). Distributed CSPs by graph partitioning. Applied Mathematics and Computation, 183(1):491–498.

SEITER, J.; WILLE, R.; SOEKEN, M.; DRECHSLER, R.(2013). Determining relevant model elements for the verification of UML/OCL specifications. In Conf. on Design, Automation and Test in Europe (DATE), pages 1189–1192, San Jose, CA, USA. EDA Consortium.

SEMERÁTH, O.; HORVÁTH, A.; VARRÓ, D. (2013). Validation of derived features and well-formedness constraints in DSLs - by mapping graph queries to an SMT-solver. In Int. Conf. on Model-Driven Engineering Languages and Systems (MODELS), pages 538–554. Springer. (Lecture Notes in Computer Science; 8107)

SHAIKH, A.; CLARISÓ, R.; KOCK WIIL, U.; MEMON, N. (2010). Verification-driven slicing of UML/OCL models. In Int. Conf. on Automated Software Engineering (ASE), pages 185–194. ACM.

SHEN, H.; LIANG, W. (1997). Efficient enumeration of all minimal separators in a graph. Theoretical Computer Science, 180(1-2):169–180.

SPIVEY, J. M. (1992). Z Notation - a reference manual (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall.

TIP, F. (1995). A survey of program slicing techniques. Journal of Programming Languages, 3(3).

TORLAK, E.; JACKSON, D. (2007). Kodkod: A relational model finder. In Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 4424 of LNCS, pages 632–647. Springer.

UZUNCAOVA, E.; KHURSHID, S. (2007). Kato: A program slicing tool for declarative specifications. In Int. Conf on Software Engineering (ICSE), pages 767–770. IEEE Computer Society.

VARRÓ, G.; SCHÜRR, A.; VARRÓ, D. (2005). Benchmarking for graph transformation. In Symp. on Visual Languages and Human-Centric Computing (VL/HCC), pages 79–88. IEEE Computer Society.

WILLE, R.; GOGOLLA, M.; SOEKEN, M.; KUHLMANN, M.; DRECHSLER, R. (2013). Towards a generic verification methodology for system models. In Conf. on Design, Automation and Test in Europe (DATE), pages 1193–1196. EDA Consortium San Jose, CA, USA / ACM DL.

WU, F.; YI, T. (2004). Slicing Z specifications. SIGPLAN Notices, 39(8):39–48.

YU, F.; BULTAN, T.; PETERSON, E. (2007). Automated size analysis for OCL. In 6th joint meeting ESEC/SIGSOFT FSE, pages 331–340. ACM.

Resumen

Asegurar la ausencia de errores en un sistema software es un problema importante pero también un desafío. La detección temprana de errores en el proceso de desarrollo de software reduce el coste de detectar y corregir los defectos. Así pues, el análisis de modelos puede incrementar la calidad final del software y reducir los costes de desarrollo.

Una línea de investigación prometedora en este campo es el uso de solvers de satisfactibilidad booleana (SAT) o programación con restricciones (CP) para realizar verificación acotada. La verificación acotada consiste en comprovar formalmente la ausencia de errores dentro de un espacio finito definido como parámetro del análisis. Este tipo de análisis es usualmente rápido en la práctica y proporciona un feedback valioso. Sin embargo, su complejidad computacional es elevada en general y no ofrece resultados concluyentes fuera del rango de verificación definido como parámetro.

En este artículo, discutimos tendencias recientes y resultados en la aplicación de verificación acotada a un campo específico dentro de la ingeniería del software : el análisis de modelos de un sistema software. Además, discutimos contribuciones prometedoras que pueden ampliar el trabajo previo para mejorar su aplicabilidad práctica dentro de la industria del software.

Palabras clave

Ingeniería del Software, Calidad del Software, Métodos Formales, Verificación Formal, Desarrollo de Software Dirigido por Modelos (DSDM), Programación con restricciones, Satisfactibilidad booleana (SAT), UML, OCL, Transformaciones de Modelos.

Resum

Assegurar l'absència d'errades en un sistema software és un problema important però també un repte. La detecció ràpida d'errors dins el procés de desenvolupament de software redueix els costos de detectar i corregir els defectes. Així doncs, l'anàlisi de models pot incrementar la qualitat final del software i reduir els costos de desenvolupament.

Una línia de recerca prometedora en aquest camp és l'us de solvers de satisfactibilitat booleana (SAT) o programació amb restriccions (CP) per realitzar verificació afitada. La verificació afitada consisteix en comprovar formalment l'absència d'errades dins d'un espai finit definit com a paràmetre de l'anàlisi. Aquest tipus d'anàlisi és usualment ràpid a la pràctica i proporciona un feedback valuós. En qualsevol cas, la seva complexitat computacional és elevada en general i no ofereix resultats concloents fora del rang de verificació definit com a paràmetre.

En aquest article, discutim tendències recents i resultats en l'aplicació de verificació afitada a un camp específic dins de l'enginyeria del software : l'anàlisi de models d'un sistema software. A més, discutim contribucions prometedores que poden ampliar el treball previ per millorar la seva aplicabilitat pràctica dins la indústria del software.

Paraules clau

Enginyeria del Programari, Qualitat del Software, Mètodes Formals, Verificació Formal, Desenvolupament de Programari Dirigit per Models (DPDM), Programació amb restriccions, Satisfactibilitat booleana (SAT), UML, OCL, Transformacions de Models.

Robert Clarisó
rclariso@uoc.edu
Estudis d'Informàtica, Multimèdia i Telecomunicació
Universitat Oberta de Catalunya

Breu currículum de l'autor
Robert Clarisó és Enginyer Informàtic (2000) i Doctor en Informàtica (2005) per la Universitat Politècnica de Catalunya (UPC-BarcelonaTech). És professor a la Universitat Oberta de Catalunya (UOC), on dirigeix el Grup de Recerca en Enginyeria del Software (GRES-UOC). També ha estat professor associat a temps parcial a la UPC (2006) I a la Universitat Autónoma de Barcelona (UAB, 2006-2011). Els seus temes de recerca inclouen els mètodes formals, l'enginyeria del programari I les eines per e-learning.
http://w.uoc.edu/robert-clariso

Robert Clarisó
rclariso@uoc.edu
Estudios de Informática, Multimedia y Telecomunicación
Universitat Oberta de Catalunya

Breve currículum del autor
Robert Clarisó es Ingeniero Informático (2000) y Doctor en Informática (2000) por la Universitat Politècnica de Catalunya (UPC-BarcelonaTech). Es profesor en la Universitat Oberta de Catalunya (UOC), donde dirige el Grupo de Investigación en Ingeniería del Software (GRES-UOC). También ha sido professor asociado a tiempo parcial en la UPC (2006) y en la Universitat Autónoma de Barcelona (UAB, 2006-2011). Sus temas de investigación incluyen métodos formales, ingeniería del software y las herramientas para e-learning.
http://w.uoc.edu/robert-clariso

IN3 Working Paper Series is a monograph series promoted by the Internet Interdisciplinary Institute (IN3) of the UOC
IN3 Working Paper Series (2010) | ISSN 2013-8644 | http://in3-working-paper-series.uoc.edu

Robert Clarisó
rclariso@uoc.edu
IT, Multimedia and Telecommunications Department
Universitat Oberta de Catalunya

Author's short CV
Robert Clarisó is a BSc (2000) and PhD (2005) in Computer Science by Universitat Politècnica de Catalunya (UPC-BarcelonaTech). He is a lecturer at Universitat Oberta de Catalunya (UOC), where he leads the Research Group in Software Engineering (GRES-UOC). He was a part-time associate professor at UPC (2006) and Universitat Autónoma de Barcelona (UAB, 2006-2011). His research interests include formal methods, software engineering and tools for e-learning.
http://w.uoc.edu/robert-clariso