Property Specification Made Easy: Harnessing the Power of Model Checking in UML designs

Daniela Remenska[1,3], Tim A.C. Willemse[2], Jeff Templon[3], Kees Verstoep[1], and Henri Bal[1]

[1] Dept. of Computer Science, VU University Amsterdam, The Netherlands
[2] Dept. of Computer Science, TU Eindhoven, The Netherlands
[3] NIKHEF, Amsterdam, The Netherlands

Abstract. One of the challenges in concurrent software development is early discovery of design errors which could lead to deadlocks or race-conditions. For safety-critical and complex distributed applications, traditional testing does not always expose such problems. Performing more rigorous formal analysis typically requires a model, which is an abstraction of the system. For object-oriented software, UML is the industry-adopted modeling language. UML offers a number of views to present the system from different perspectives. Behavioral views are necessary for the purpose of model checking, as they capture the dynamics of the system. Among them are sequence diagrams, in which the interaction between components is modeled by means of message exchanges. UML 2.x includes rich features that enable modeling code-like structures, such as loops, conditions and referring to existing interactions. We present an automatic procedure for translating UML into mCRL2 process algebra models. Our prototype is able to produce a formal model, and feed model-checking traces back into any UML modeling tool, without the user having to leave the UML domain. We argue why previous approaches of which we are aware have limitations that we overcome. We further apply our methodology on the Grid framework used to support production activities of one of the LHC experiments at CERN.

Keywords: property specification, model checking, UML, sequence diagrams, modal μ-calculus, property patterns

1 Introduction

One of the challenges in concurrent software development is early discovery of design errors which can lead to deadlocks or race-conditions. Traditional testing does not always expose such problems in complex distributed applications. Performing more rigorous formal analysis, like model-checking, typically requires a model which is an abstraction of the system. In the last decades, more rigorous methods and tools for modeling and formal analysis have been developed. Some of the leading model checking tools include SPIN, nuSMV, CADP and mCRL2. Despite the research effort, these methods are still not widely accepted in industry. One problem is the lack of expertise and the necessary time investment in

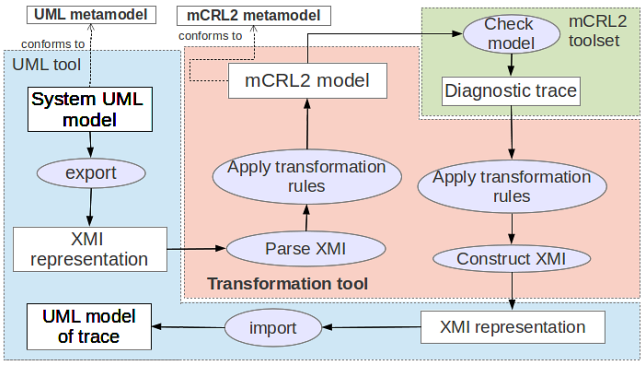

Fig. 1. Automated verification of UML models

the development cycle, for becoming proficient in the underlying mathematical formalisms used for describing the models. To bridge the gap between industry-adopted methodologies based on UML software designs, and model-checking tools and languages, in [1] we devised an automated transformation methodology for verification of UML models, based on sequence and activity diagrams. Our prototype is able to produce a formal model into the mCRL2 process algebra language [2], and feed model-checking traces back into any modeling tool, without the user having to leave the UML domain. We chose mCRL2 because of its strong tool support and rich data types compared to other languages. Figure 1 gives an overview of our approach and implemented toolchain. Although the mCRL2 toolset automatically discovers deadlocks, model checking for application-specific properties requires the use of modal μ-calculus [3]. In principle, regardless of the formal language and tool choice for writing the model, these properties are specified as formulas in some temporal logic formalism, such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Quantified Regular Expressions (QRE) or μ-calculus. The level of sophistication and mathematical background required for using such formalisms is yet another obstacle for adopting formal methods. In practice, software requirements are written in natural language, and often contain ambiguities, making it difficult even for experienced practitioners to capture them accurately with temporal logic. There are subtle, but crucial details which are often overlooked and need to be carefully considered in order to distill the right formula. The objective of this work is to simplify the process of correctly eliciting functional requirements, without the need of expertise in temporal logic.

Based on investigation of more than 500 properties coming from different domains, and specified in several formalisms, a pattern-based classification was developed in [4]. The authors observed that almost all the surveyed properties can be mapped into one of several property patterns. Each pattern is a high-level, formalism-independent abstraction that captures a commonly occurring requirement. These patterns can be instantiated with specific events or states and then mapped to several different formalisms for model checking tools. Their hierarchical taxonomy is based on the idea that each pattern has a *scope*, which defines the extent of program execution over which the pattern must hold, and a

behavior, which describes the intent of the pattern. The pattern system identifies 5 scopes and 8 behavior variations that can be combined to create 40 different properties. Examples of scopes are: globally, before an event or state occurs, after an event or state occurs. Examples of behavior classification are: absence (an event or state should never occur during an execution), precedence (which requires that a given event or state always occurs before another one), or response (the occurrence of a given event or state must be followed by designated event or state), capturing a cause-effect relation. Although the patterns website [5] contains a collection of templates for different target formalisms, such as LTL, CTL, Graphical Interval Logic (GIL), and Quantified Regular Expressions (QRE), which can be considered helpful, practitioners have to fully understand the provided solutions before they can select and apply the appropriate ones in practice.

To mitigate the problem, several approaches propose conversational tools for elucidating properties, based on the property patterns. In [6] the authors developed PROPEL, a tool for guiding users in selecting the appropriate template. Recognizing that there are subtle aspects not covered by the original patterns, such as what happens in a response property if the cause occurs multiple times before the effect takes place, they extended them with variants. The resulting templates are represented using disciplined natural language and finite state automata. In a similar manner, SPIDER [7] and Prospec [8] offer assistance in the specification process, and extend the original patterns with compositional patterns that are built up from combinations of more basic patterns. Another category of approaches deal with temporal extensions of the Object Constraint Language (OCL), as means to specify system properties. OCL is a declarative textual language for describing invariants for classes and pre- and postconditions of operations. Although it forms an integral part of UML, it lacks means to specify constraints over the dynamic behavior of a model. For specification of past and future state-oriented constraints, the @next and @pre temporal modifiers are introduced in [9]. By means of UML Profiles, [10] proposes another OCL extension for real-time constraints. They claim to be able to describe all the existing patterns in these OCL expressions. Their starting point for model descriptions are UML state machines. To simplify constraint definition with OCL, in [11] the authors propose to use specification patterns for which OCL constraints can be generated automatically. The behavioral specification of software components refers to interface specifications, which are not really dynamic views. This work does not yet introduce means to specify temporal properties. Resembling an OO programming language, OCL constraints can become quite complex and cryptic, and editing them manually is error-prone. Another problem is the extent to which designers are familiar with this language. Finally, a third class of approaches tackle the property specification problem by proposing graphical notations, which come closer to the realm of modeling the system behavior. The CHARMY approach [12] presents a scenario-based visual language called Property Sequence Charts, where a property is seen as a relation on a set of exchanged system messages. The language borrows concepts from UML 2.0 Sequence Di-

agrams, and its expressiveness is measured with the property patterns. SPIN is used as a backend for model checking of the Buchi automata [13], which are an operational representation for LTL formulas generated automatically with this approach. Another graphical language is proposed in [14], where formulas are represented as acyclic graphs of states and temporal operators as nodes. While they manage to hide the formal notation from the user by generating LTL formulas, their notation is still very close to an actual temporal logic formula. The TimeLine Editor [15] also attempts to simplify the formalization of certain kinds of requirements. Response formulas are depicted in timeline diagrams by specifying temporal relations among events and constraints. The timeline specification is automatically converted into a Buchi automaton, amenable to model checking with SPIN. HUGO/RT [16] is a tool for model checking UML 2.0 interactions against a model composed of message-exchanging state machines. The interactions represent the desired properties, and are translated together with the system model into Buchi automata for model checking with SPIN. The approach uses some inner format for textual representation of UML interactions (rather than the standard XMI), and the version we tested does not support asynchronous messages, or combined fragments. vUML [17] is a tool for automatic verification of UML models comprising state machines. However, properties must be specified in terms of undesired events, which is not always convenient. This is because the verification is based on checking whether it is possible to reach error states, specified by the user. Live Sequence Charts (LSC) are also used [18, 19] as a graphical formalism for expressing behavioral properties. Every element in an LSC has a temperature which can be either hot or cold. This is used to distinguish between possible (cold) and mandatory (hot) elements and behaviour. In both approaches, Buchi automata and a LTL formulas However, UML 2.0 sequence diagrams borrow many concepts from LSC, by introducing the assert and negate fragments to capture mandatory and forbidden behavior. On the other hand, being an older graphical notation, LSC lack many UML features.

We have thoroughly surveyed the advantages and shortcomings of the most relevant approaches up to date, and a more detailed comparison with our approach is given in Section 5.

2 Preliminaries

2.1 Property Patterns

2.2 Brief Introduction to mCRL2 and μ-calculus

2.3 UML Sequence Diagrams

3 The Approach

3.1 The Rationale

3.2 Transforming a μ-calculus Formula Into a Monitor Process

We translate a fragment of the μ-calculus to mCRL2 processes which can subsequently serve as monitor processes.

We restrict to the following grammar:

$$\begin{array}{ll} \phi_1 & ::= b \mid \forall d : D.\phi_1 \mid [R]\phi_1 \mid \phi_1 \wedge \phi_2 \\ R_1, R_2 & ::= \alpha \mid nil \mid R_1 \cdot R_2 \mid R_1 + R_2 \mid R_1^* \mid R_1^+ \\ \alpha_1, \alpha_2 & ::= b \mid \mathsf{a}(\mathsf{e}) \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \mid \exists d : D.\alpha_1 \end{array}$$

Before we present the translation, we convert the formulas in guarded form. That is, we remove every occurrence of R^* and nil using the following rules:

$$\begin{array}{l} [nil]\phi = \phi \\ [R^*]\phi = [nil]\phi \wedge [R^+]\phi \end{array}$$

The function TrS takes two arguments (a formula and a list of typed variables) and produces a process. It is defined inductively as follows:

$$\begin{array}{ll} \mathsf{TrS}_l(b) & = (\neg b \rightarrow \mathsf{error}) \\ \mathsf{TrS}_l(\forall d : D.\phi_1) & = \sum d : D.\mathsf{TrS}_{l +\!\!+ [d:D]}(\phi_1) \\ \mathsf{TrS}_l(\phi_1 \wedge \phi_2) & = \mathsf{TrS}_l(\phi_1) + \mathsf{TrS}_l(\phi_2) \\ \mathsf{TrS}_l([R]\phi_1) & = \mathsf{TrR}_l(R) \cdot \mathsf{TrS}_l(\phi) \end{array}$$

where TrR takes a regular expression (and a list of typed variables) and produces a process or a condition:

$$\begin{array}{ll} \mathsf{TrR}_l(\alpha) & = \bigoplus_{a \in Act} (\sum d_a : D_a.\ \mathsf{Cond}_l(a(d_a), \alpha) \rightarrow a(d_a)) \\ \mathsf{TrR}_l(R_1 \cdot R_2) & = \mathsf{TrR}_l(R_1) \cdot \mathsf{TrR}_l(R_2) \\ \mathsf{TrR}_l(R_1 + R_2) & = \mathsf{TrR}_l(R_1) + \mathsf{TrR}_l(R_2) \\ \mathsf{TrR}_l(R_1^+) & = X(l) \qquad \textit{where } X(l) = \mathsf{TrR}_l(R_1) \cdot X(l) \textit{ is a recursive process} \end{array}$$

where $\bigoplus$ is a finite summation over all action names $a \in Act$ and where Cond takes an action and an action formula and produces a condition that describes

when the action is among the set of actions described by the action formula:

$$
\begin{array}{ll}
\mathsf{Cond}_l(a(d_a), b) & = b \\
\mathsf{Cond}_l(a(d_a), a'(e)) & = \begin{cases} d_a = e \text{ if a = a'} \\ \text{false} \quad \text{otherwise} \end{cases} \\
\mathsf{Cond}_l(a(d_a), \neg\alpha_1) & = \neg\mathsf{Cond}_l(a(d_a), \alpha_1) \\
\mathsf{Cond}_l(a(d_a), \alpha_1 \wedge \alpha_2) & = \mathsf{Cond}_l(a(d_a), \alpha_1) \wedge \mathsf{Cond}_l(a(d_a), \alpha_2) \\
\mathsf{Cond}_l(a(d_a), \exists d : D.\alpha_1) & = \exists d : D.\mathsf{Cond}_l(a(d_a), \alpha_1)
\end{array}
$$

3.3 The Wizard

mention free drawing and the profile application

4 Case Study: DIRAC's Executor Framework revisited

5 Related Work

6 Conclusions and future work

compositional patterns

References

1. Remenska, D., Templon, J., Willemse, T.A.C., Homburg, P., Verstoep, K., Casajus, A., Bal, H.E.: From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems. In: NASA Formal Methods. (2013) 244–260
2. Groote, J., et al.: The Formal Specification Language mCRL2. In: Proc. MMOSS'06
3. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. In: Science of Computer Programming, Elsevier (2005) 251–273
4. Dwyer, M.B., et al.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering. ICSE '99, New York, NY, USA, ACM (1999) 411–420
5. Dwyer, M.B., et al.: Property Specification Patterns http://patterns.projects.cis.ksu.edu.
6. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: Propel: an approach supporting property elucidation. In: 24th Intl. Conf. on Software Engineering, ACM Press (2002) 11–21
7. Konrad, S., Cheng, B.H.: Facilitating the construction of specification pattern-based properties. In: Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, IEEE (2005) 329–338
8. Mondragon, O., Gates, A.Q., Roach, S.: Prospec: Support for Elicitation and Formal Specification of Software Properties. In: Proc. of Runtime Verification Workshop, ENTCS. 2004
9. Ziemann, P., Gogolla, M.: An Extension of OCL with Temporal Logic. In: Critical Systems Development with UML. (2002) 53–62

10. Flake, S., Mueller, W.: Formal Semantics of Static and Temporal State-Oriented OCL Constraints. Software and Systems Modeling (SoSyM), Springer **2** (2003) 186
11. Ackermann, J., Turowski, K.: A library of OCL specification patterns for behavioral specification of software components. In: Proceedings of the 18th international conference on Advanced Information Systems Engineering. CAiSE'06, Berlin, Heidelberg, Springer-Verlag (2006) 255–269
12. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engg. **14**(3) (September 2007) 293–340
13. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proceedings of the 16th IEEE international conference on Automated software engineering. ASE '01, Washington, DC, USA, IEEE Computer Society (2001) 412–
14. Lee, I., Sokolsky, O.: A Graphical Property Specification Language. In: In Proceedings of 2nd IEEE Workshop on High-Assurance Systems Engineering. IEEE Computer, Society Press (1997) 42–47
15. Smith, M.H., Holzmann, G.J., Etessami, K.: Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs. In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering. RE '01, Washington, DC, USA, IEEE Computer Society (2001) 14–
16. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Proceedings of the 2006 international conference on Models in software engineering. MoDELS'06, Berlin, Heidelberg, Springer-Verlag (2006) 42–51
17. Lilius, J., Paltor, I.P.: vUML: a Tool for Verifying UML Models. In: . (1999) 255–258
18. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'05, Berlin, Heidelberg, Springer-Verlag (2005) 445–460
19. Baresi, L., Ghezzi, C., Zanolin, L.: Modeling and Validation of Publish/Subscribe Architectures. In Beydeda, S., Gruhn, V., eds.: Testing Commercial-off-the-Shelf Components and Systems. Springer Berlin Heidelberg 273–291