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 the

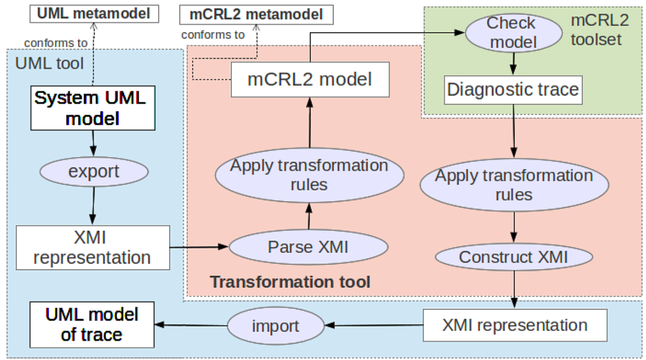

Fig. 1. Automated verification of UML models

development cycle, for becoming proficient in the underlying mathematical formalisms used for describing the models. A more substantial problem is the lack of a systematic connection between actual implementation and the semantics of the existing formal languages. 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 formulae 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, specification 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 variations are: absence (an event or state should never occur during an execution), precedence (which require that a given event or state always occurs before another one), or response properties (which require that the occurrence of a given event or state 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.

We have thoroughly surveyed the advantages and disadvantages of these approaches, 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 formulae 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:

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

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

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

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

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.