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

As modern software systems become more complex and distributed, a major challenge is faced in maintaining their quality and functional correctness. Early discovery of design errors which could lead to deadlocks, race-conditions and other flaws, before they can surface, is of a paramount importance. The Unified Modeling Language (UML) has become the lingua franca of software engineering, in particular for the domain of object-oriented systems. Over time, several mature CASE tools have already adopted UML as the industry-standard visual modeling language for describing software systems. However, use of these tools alone does not assure the correctness of the design, nor does it provide direct

means to test the software under design, 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. In the last decades, more rigorous methods and tools for modeling and analysis have been proposed. 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 OO development cycle, for becoming proficient in them. A more substantial problem is the lack of a systematic connection between actual implementation and the semantics of the existing formal languages.

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:

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

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