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. Early discovery of design errors which can lead to deadlocks or race conditions is challenging in concurrent software development. In the last decades, more rigorous methods and tools for modeling and formal analysis have been developed. Although approaches for automatically generating formal models from system designs have been proposed, another serious obstacle for adopting model checking tools in industry is the formulation of application-specific properties to be checked. This requires expertise in temporal logic, regardless of the verification tool used. To bring the process of correctly eliciting functional properties closer to software designers, we introduce PASS, a Property ASSistant wizard developed as an Eclipse plugin. Our starting point was the well-established property pattern system, which we extended with new property templates, to capture variations not covered in the original classification. PASS instantiates pattern templates using three notations: a natural language summary, a μ-calculus formula and a UML sequence diagram depicting the desired behavior. Most approaches to date have focused on LTL, which is a state-based formalism. On the other hand, μ-calculus is event-based, making it a good match for sequence diagrams, where communication between components is depicted. Moreover, such communication is data-dependent, so we introduce the possibility to define data quantifiers, to express complex properties in a concise manner. To cope with state-space explosion, we provide one additional notation: a monitor for on-the-fly model checking, or bug hunting. We revisit a case study from the Grid domain, using PASS to obtain the formula and monitor for checking the property with mCRL2.

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, methods and tools for modeling and formal analysis have been developed. Some of the

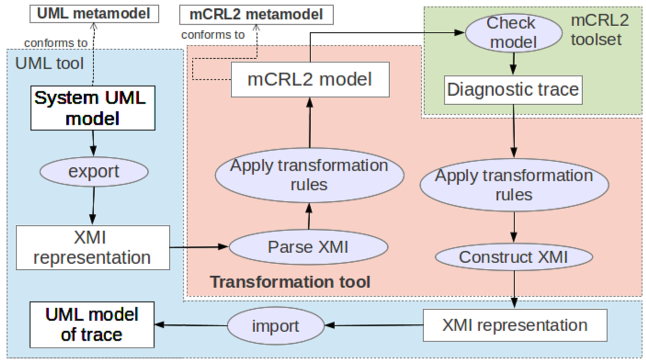

Fig. 1. Automated verification of UML models

leading model checking tools include SPIN, nuSMV, CADP and mCRL2. Despite the research effort, these tools are still not widely accepted in the software industry. One problem is the lack of expertise and the necessary time investment in 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.

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 11 behavior variations that can be combined to create 55 different property templates. 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 (an event or state always occurs before another one), or response (the occurrence of a an event or state must be followed by another event or state), capturing a cause-effect relation. Although the patterns website [5] contains a collection of mappings for different target formalisms, such as LTL, CTL, QRE and Graphical Interval Logic, 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 [6–8] propose conversational tools for elucidating properties, based on the property patterns. These tools guide users in selecting the appropriate pattern for the property in mind, and optionally produce a formula in some target temporal logic. Another category of approaches [9–11] 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. Finally, a third class of approaches [12–18] tackle the property specification problem by proposing graphical notations for specifying properties.

The objective of this work is to simplify the process of correctly eliciting functional requirements, without the need of expertise in temporal logic. First, we introduce PASS, a Property ASSistant which guides and facilitates the process of deriving system properties. Our starting point was the pattern system, which we extended with over 40 new property templates, to capture variations not covered in the original classification. Our strong motivation was to stay in the same UML development environment, rather than use an external helper tool for this. It should increase the tool accessibility by allowing software engineers to remain focused in the realm of UML designs. In addition, a tight bond between elements of the design and instances of the property template is kept, such that, if the design is changed, these changes can be easily propagated in the property template placeholders. To this end, we use the standard MDT-UML2 [19] Eclipse modeling API. Our tool is developed as an Eclipse plugin. Second, the pattern templates instantiated with PASS have three notations: a natural language summary, a μ-calculus formula and a UML sequence diagram depicting the desired behavior. Most approaches to date have focused on LTL, which is a state-based temporal logics formalism. On the other hand, μ-calculus is event-based, and as such is a good match for the sequence diagrams notation. These events can represent methods calls or asynchronous communication between distributed components. Moreover, such communication is data-dependent, which is why we introduce the possibility to define quantifiers, to express complex properties in a concise manner, e.g., every element of a certain type must fulfill a certain property. Compared to LTL or CTL, μ-calculus is powerful enough to achieve this in a natural way. Third, to cope with state-space explosion, we provide one additional automatically-generated notation: a monitor for on-the-fly model checking, or bug hunting. We interpret a sequence diagram as an observer of the message exchanges in the system. This helps in avoiding generation of those parts of the state space for which it is certain that they do not compose with the property monitor. The state space generation is thus property driven, and stops as soon as an error is found. Finally, we revisit a case study we did previously in [1], this time using the PASS tool to automatically obtain the formula and monitor for checking the property in mCRL2.

This paper is structured as follows: in Section 2 we survey the most relevant related approaches, and outline their advantages and shortcomings. Section 3 briefly introduces the syntax and semantics of mCRL2, μ-calculus and UML sequence diagrams. We describe our approach in Section 4. In Section 5 we apply it on a case study from the Grid domain, and we conclude in Section 6.

2 Related Work

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. PROPEL does not support the universality, bounded existence, and the chain patterns. It also does not produce a formula in any of the commonly used temporal logic formalisms. In a similar manner, SPIDER [7] and Prospec [8] offer assistance in the specification process, and extend the original patterns with compositional ones that are built up from combinations of more basic patterns. Unfortunately, we could not find SPIDER online, and the version of Prospec that we tested (Fig. 2a) produces only formulas in Future Interval Logic, not LTL as stated in the work.

Of the approaches that deal with temporal extensions of OCL, [9] introduces the @pre and @next temporal modifiers for specification of past and future state-oriented constraints. 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 dense and cryptic, and editing them manually is error-prone. Another problem is the extent to which designers are familiar with this language.

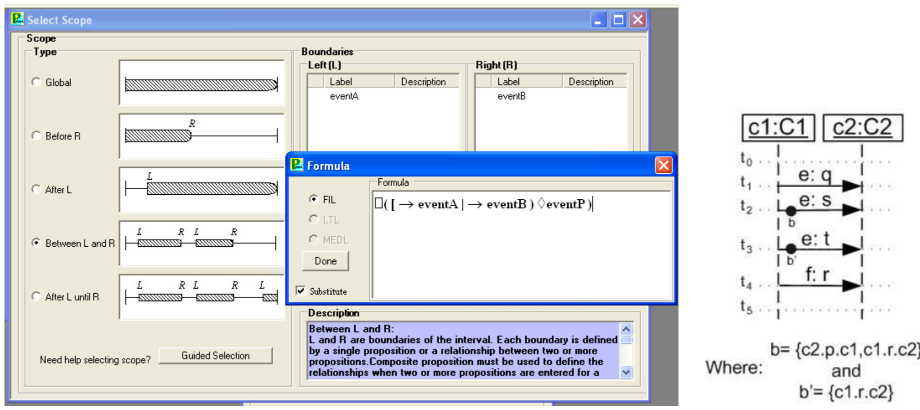

Fig. 2. Left: Prospec tool; Right: CHARMY PSC graphical notation

Graphical notation approaches come closest to the realm of modeling the system behavior. The CHARMY approach [12] presents a scenario-based visual language called Property Sequence Charts (PSC), where a property is seen as a relation on a set of exchanged system messages. The language borrows concepts from UML 2.0 Sequence Diagrams, and its expressiveness is measured with the property patterns. SPIN is used as a backend for model checking of the Buchi automata [20], which are an operational representation for LTL formulas generated automatically with this approach. The PSC notation uses textual restrictions for past and future events, placed as circles directly on the message arrows (Fig. 2b). Such a mix of textual and visual representation of message communication within a diagram can be error-prone. Additionally, asynchronous communication is not supported. Furthermore, even though concepts from UML are borrowed, CHARMY is a stand-alone framework for architectural descriptions, not inter-operable with UML tools, and as such has limited usefulness in industrial context. Another graphical language is proposed in [13], where formulas are represented as acyclic graphs of states and temporal operators as nodes. While they manage to hide the formalism from the user by generating LTL formulas, their notation is still very close to an actual temporal logic formula. The TimeLine Editor [14] 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. Unfortunately the tool is no longer available. HUGO/RT [15] 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 [16] is a tool for automatic verification of UML models comprising state machines. However, properties must be specified in terms of undesired scenarios, which is not always convenient. This is because the verification is based on checking whether it is possible to reach error states, which must be manually specified by the user. Live Sequence Charts (LSC) are also used [17, 18] as a graphical formalism for expressing behavioral properties. Their elements allow to distinguish between possible (cold) and mandatory (hot) behavior. In both approaches, Buchi automata and a LTL formulas are generated automatically from the diagrams. However, UML 2.0 sequence diagrams already 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.

3 Preliminaries

3.1 Brief Introduction to mCRL2 and μ-calculus

mCRL2 is a process algebra language for specification and analysis of concurrent systems. Our choice of mCRL2 as a formal language is motivated by its rich set of abstract data types as first-class citizens, as well as its powerful toolset for analysis, simulation, and visualization of specifications. The syntax of mCRL2 is given by the following BNF grammar:

$$p ::= a(d_1, \ldots, d_n) \mid \tau \mid \delta \mid p + p \mid p.p \mid p||p \mid \sum_{d:D} p \mid c \rightarrow p \diamond p$$

Actions are the basic ingredients for models. They represent some observable atomic event. An action a of a process may have a number of data arguments $d_1, ..., d_n$. The action τ denotes an internal step, which cannot be observed from the external world. Non-deterministic choice between two processes is denoted by the $+$ operator. Processes can be composed sequentially and in parallel by means of "." and "$||$". The sum operator $\sum_{d:D} p$ denotes (possibly infinite) choice among processes parameterized by d. $c \rightarrow p \diamond p$ is a conditional process, and depending on the value of the boolean expression c, the first or second operand is selected. This allows for modeling of systems whose behavior is data-dependent.

To enforce synchronization, the allow operator $\Delta_H(p)$ specifies the set of actions H that are allowed to occur. To show possible communications in a system and the resulting actions, the communication operator $\Gamma_C(p)$ is used. The elements of set C are so-called multi-actions of the form $a_1 \mid a_2 \mid \ldots \mid a_n \rightarrow c$, which intuitively means that action c is the result of the multi-party synchronization of actions $a_1, a_2, \ldots$ and a_n. There are a number of built-in data types in mCRL2, such as (unbounded) integers, (uncountable) reals, booleans, lists, and sets. Furthermore, by a **sort** definition one can define a new data type. A new process is declared by **proc**.

The semantics associated with the mCRL2 syntax is a Labeled Transition System (LTS) system that has multi-action labeled transitions, which can carry data parameters. The language used by the mCRL2 toolset for model-checking specific properties is the modal μ-calculus [21]. This formalism stands out from most modal and temporal logic formalisms with respect to its expressive power.

Temporal logics like LTL, CTL and CTL* all have translations into μ-calculus, witnessing its generality. This expressiveness comes at a cost: very complex formulas with no intuitive and apparent interpretation can be coined. The syntax of a μ-calculus formula is defined by the following grammar:

$$\begin{array}{ll} \phi_1 & ::= true \mid false \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\rho]\phi \mid \langle\rho\rangle\phi \mid \mu Z.\, \phi \mid \nu Z.\, \phi \\ \rho & ::= \alpha \mid \rho \cdot \rho \mid \rho^* \\ \alpha & ::= \alpha \cup \alpha \mid \alpha \cap \alpha \mid \neg\alpha \mid true \end{array}$$

Properties represented by ϕ can be combined using the standard Boolean connectives. The formula $\phi_1 \wedge \phi_2$ is true whenever both ϕ_1 and ϕ_2 are true. ρ represents a set of sequences of actions, while α stands for a set of actions. To describe a sequence of actions ρ, the concatenation $\rho \cdot \rho$ and iteration ρ^* (possibly infinite sequences) operators can be used. Union and intersection of sets of actions is denoted by $\cup$ and $\cap$ respectively. The notation $\neg\alpha$ denotes the complement of the set of actions α with respect to the set of all actions. The set of actions characterized by *true* is the entire set of actions present in the model.

The $\Box$ (*must*) and $\Diamond$ (*may*) modalities have the following semantics: a state of the LTS satisfies $\langle R\rangle\phi$ iff there is at least one transition sequence starting at that state, satisfying R, and leading to a state satisfying ϕ; $[R]\phi$ is satisfied iff all sequences starting at that state and satisfying R, lead to states satisfying ϕ. In other words, $[R]\phi$ describes that ϕ holds in all states that can be reached by a sequence in R, while $\langle R\rangle\phi$ describes that ϕ holds in some state that can be reached by a sequence in R. Remember that $[a]\phi$ is trivially satisfied for states with no a-transitions. Combined with these modalities, the least $(\mu X.\, f(X))$ and greatest $(\nu X.\, f(X))$ fixpoints permit reasoning about finite and infinite runs of a system in a recursion-like manner. For example, we can read $\mu X.\, \phi \vee \langle\alpha\rangle X$ as: X is the smallest set of states such that a state is in X iff ϕ holds in that state or there is an α-successor in X. On the other hand, Conversely, $\nu X.\, \phi \wedge [\alpha]X$ is the largest set of states such that a state is in X iff ϕ holds in that state and all of its α-successors are in X.

Finally, a strong asset of μ-calculus are the universal (*forall*) and existential (*exists*) quantifiers over possibly-infinite data types. For example, *forall n:Nat.⟨read(n)⟩true* asserts that a process can execute a read action with every natural number as a parameter; *[!exists n:Nat.read(n+n)]false* states that a process should only execute read actions with even-valued natural numbers.

Verification using μ-calculus sometimes requires too much overhead to serve as a basis for lightweight bug-hunting, since the entire state-space must be kept in memory in the verification process. Observers, or monitors (à la Buchi) defined in the mCRL2 model itself, are used in some cases to bypass the problem. Not all μ-calculus formulas are amenable to such a conversion.

3.2 UML Sequence Diagrams

Sequence diagrams model the interaction among a set of components, with emphasis on the sequence of *messages* exchanged over time. Graphically, they have

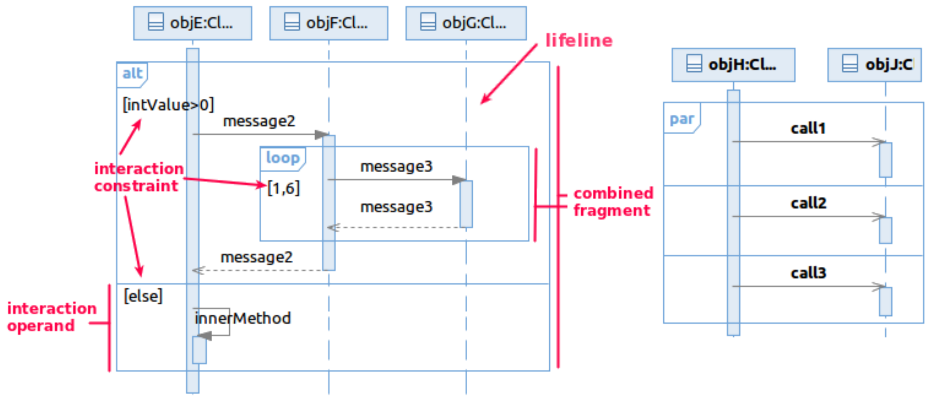

Fig. 3. Sequence diagrams with combined fragments

two dimensions: the objects participating in the scenarios are placed horizontally, while time flows in the vertical dimension. The participants are shown as rectangular boxes, with the vertical lines falling from them known as *lifelines*. Each message sent between the lifelines defines a specific act of communication, synchronous or asynchronous. Messages are shown as horizontal arrows from the lifeline of the sender instance to the lifeline of the receiver.

Sequence diagrams have been considerably extended in UML 2.x to allow expressing of complex control flows such as branching, iterations, and referring to existing interactions. **Combined fragments** are used for this purpose. The specification supports different fragment types, such as *alt, opt, loop, break, par*. They are visualized as rectangles with a keyword indicating the type. Each combined fragment consists of one or more interaction operands. Depending on the type of the fragment, constraints can guard each of the interaction operands. Combined fragments can be nested with an arbitrary nesting depth, to capture complex workflows. Figure 3 shows how some of them can be used. There are also two less-known combined fragments: *assert* and *neg*. Their use in pracice is limited, because their semantics described in the UML 2.0 superstructure specification [22] is rather vague and confusing. By default, sequence diagrams without the use of these two operators only reflects possible behavior, while *assert* and *neg* alter the way a trace can be classified as valid or invalid. The specification characterizes the semantics of a sequence diagram as a pair of valid and invalid traces, where a trace is a sequence of events or messages. The potential problems with the UML 2.0 assertion and negation are explained in [23]. In summary, the specification aims to allow depicting required and forbidden behaviors. However, as they point out, stating that “the sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace” suggests that the invalid set of traces for an *assert* fragment is its complement, i.e., the set of all other possible traces. On the other hand, the standard declares that the invalid set of traces are associated only with the use of a *neg* fragment, which is contradictory. For this reason, we also believe that these two operators should rather be considered as modalities. We restrict their usage to property specifications, and assign the following semantics: *neg* is considered a set-complement operator for the event captured by the fragment, while *assert* specifies that an event sequence must occur for the property to hold. In addition, restrict the usage of *neg* to a single event (or message), and dissalow nesting of *assert* within a *neg* fragment. We find that this does not limit the expressiveness of property specifications in practice.

4 The Approach

4.1 The Rationale

4.2 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:

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

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:

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

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:

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

4.3 The Wizard

5 Case Study: DIRAC's Executor Framework revisited

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.
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. 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
14. 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–
15. 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
16. Lilius, J., Paltor, I.P.: vUML: a Tool for Verifying UML Models. In: . (1999) 255–258
17. 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
18. 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
19. Foundation, T.E.: Eclipse Modeling MDT-UML2 component www.eclipse.org/uml2/.
20. 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–
21. Emerson, E.A.: Model checking and the Mu-calculus. In: DIMACS Series in Discrete Mathematics, American Mathematical Society (1997) 185–214
22. OMG: UML2.4 Superstructure Specification http://www.omg.org/spec/UML/2.4/Superstructure.
23. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams (2007)