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. Conversely, 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 developing concurrent software 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 a 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 leading

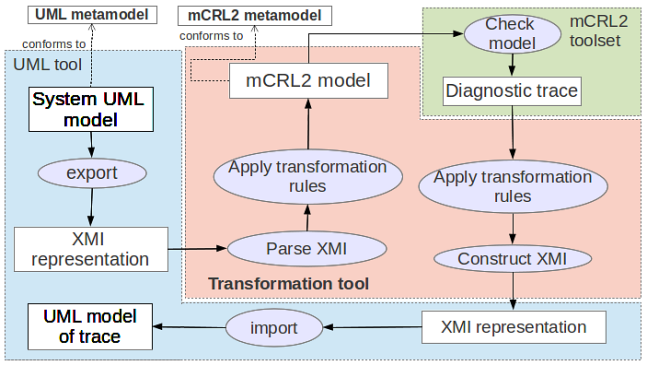


Fig. 1. Automated verification of UML models

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 verifying UML models, based on sequence and activity diagrams. Figure 1 gives an overview of our approach and implemented toolchain. 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.

Although the mCRL2 toolset automatically discovers deadlocks, model checking of application-specific properties relies on using the modal μ-calculus [3] for expressing these properties. 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 distil the right formula.

Based on investigating over 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 hierarchi-

cal 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 above 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 the 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 our 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 [4], which we extended with over 100 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 logic formalism. Conversely, the 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 fulfil a certain property. Unlike LTL or CTL, the 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 to avoid exploring irrelevant parts of the state space. 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 PASS 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 from the patterns classification. 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 latest version of Prospec that we found and tested (Fig. 2 left) 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,

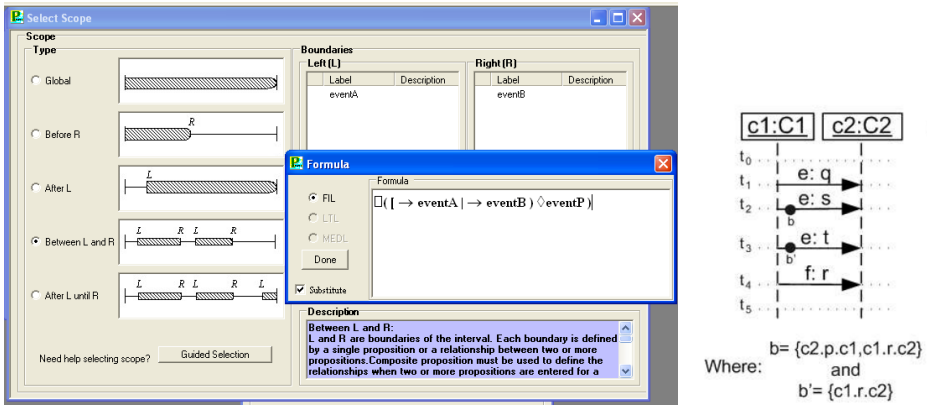


Fig. 2. Left: Prospec tool; right: CHARMY PSC graphical notation

and editing them manually is error-prone. Another problem is the extent to which designers are familiar with this language.

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 Büchi 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. 2 right). 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 standalone framework for architectural descriptions, not inter-operable with UML tools, and as such its use in industrial contexts is limited. 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 Time-Line 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 Büchi 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 Büchi 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 frag-

ments. 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, Büchi automata and 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 lacks many UML features.

3 Preliminaries

3.1 Brief Introduction to mCRL2 and μ-calculus

mCRL2 is a language and accompanying toolset for specifying and analyzing concurrent systems. Our choice for using the mCRL2 language is motivated by its rich set of abstract data types as first-class citizens, as well as its powerful toolset for analyzing, simulating, and visualizing specifications. The fragment of the mCRL2 syntax that is most commonly used is given by the following BNF grammar:

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

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 "$\cdot$" and "||". The sum operator $\sum_{d:D} p$ denotes (possibly infinite) choice among processes parameterized by variable d. The behavior of the conditional process $c \rightarrow p \diamond p$ depends on the value of the boolean expression c: if it evaluates to true, process p is chosen and otherwise process q is chosen. This allows for modeling systems whose behavior is data-dependent.

To enforce synchronization between processes, the allow operator $\nabla_H(p)$ 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. Recursive process equations can be 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 an extension of 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 [22, 23] 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 mCRL2's modal μ-calculus formulas we are concerned with in this paper is defined by the following grammar:

$$\begin{aligned}
&\phi ::= b \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall d{:}D.\ \phi \mid \exists d{:}D.\ \phi \mid [\rho]\phi \mid \langle\rho\rangle\phi \mid \mu Z.\, \phi(Z) \mid \nu Z.\, \phi(Z)\\
&\rho ::= \alpha \mid \mathit{nil} \mid \rho \cdot \rho \mid \rho^* \mid \rho^+\\
&\alpha ::= a(d_1, \ldots, d_n) \mid b \mid \neg\alpha \mid \alpha \cap \alpha \mid \alpha \cup \alpha \mid \bigcap d{:}D.\ \alpha \mid \bigcup d{:}D.\ \alpha
\end{aligned}$$

Properties are expressed by state formulas ϕ, which contain Boolean data terms b that evaluate to true or false and which can contain data variables, and the standard logical connectives *and* ($\wedge$) and *or* ($\vee$), the modal operators *must* ($[\_]\_$) and *may* ($\langle\_\rangle\_$), and the least and greatest fixpoint operators μ and ν. In addition to these, mCRL2's extensions add universal and existential quantifiers $\forall$ and $\exists$.

The modal operators take regular expressions ρ for describing words of actions, built up from the empty word *nil*, individual actions described by an action formula α, word concatenation $\rho \cdot \rho$ and (arbitrary) iteration of words ρ^* and ρ^+. Action formulas describe sets of actions; these sets are built up from the empty set of actions (in case Boolean expression b evaluates to false), the set of all possible actions (in case Boolean expression b evaluates to true); indivual actions $a(d_1, \ldots, d_n)$, action complementation and finite and possibly infinite intersection $\cap$ and union $\cup$. A state of an LTS (described by an mCRL2 process) satisfies $\langle\rho\rangle\phi$ iff from that state, there is at least one transition sequence matching ρ, leading to a state satisfying ϕ; $[\rho]\phi$ is satisfied by a state iff all transition sequences matching ρ starting in that state lead to states satisfying ϕ. For instance, $[\neg(\bigcup n{:}Nat.\ \mathit{read}(n+n))]$false states that a process should not execute any actions other than read actions with even-valued natural numbers. Remember that $[a]\phi$ is trivially satisfied in states with no "a"-transitions.

Combining these modalities, the least ($\mu X.\, \phi(X)$) and greatest ($\nu X.\, \phi(X)$) 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, $\nu X.\, \phi \wedge [\alpha]X$ 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, too. Finally, a strong asset of mCRL2's μ-calculus are the universal $\forall$ and existential $\exists$ quantifiers over potentially infinite data types. For example, $\forall n{:}Nat.\langle\mathit{read}(n)\rangle$true asserts that a process can execute a *read* action, accepting every natural number as a parameter.

In mCRL2, verification of μ-calculus formulas is conducted using tooling that operates on systems of fixpoint equations over first-order logic expressions. This

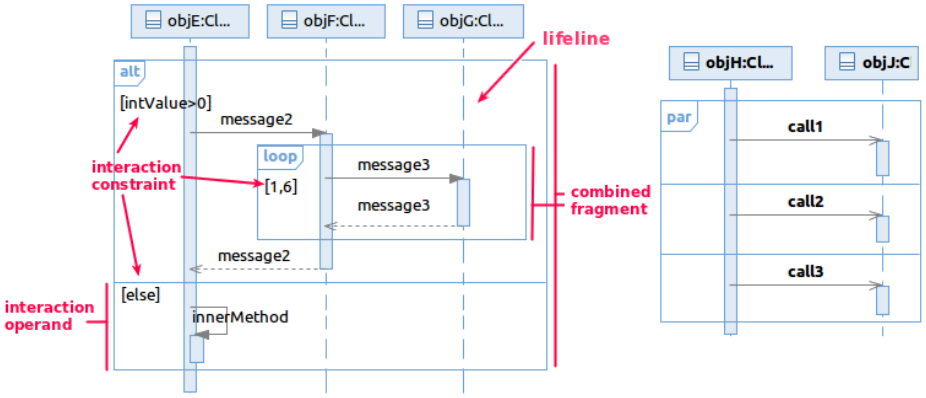


Fig. 3. Sequence diagrams with combined fragments

sometimes requires too much overhead to serve as a basis for lightweight bug-hunting, as it can be difficult to interpret the counterexamples that are obtained from these equation systems in terms of the original mCRL2 process. Observers, or monitors (à la Büchi) defined in the mCRL2 model itself, can sometimes be used to bypass the problem. However, 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 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, with operators 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 practice is limited, because their semantics described in the UML 2.0 superstructure specification [24] is rather vague and confusing. By default, sequence diagrams without the use of these two operators only reflect 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 [25]. In summary, the specification aims to allow depicting required and forbidden behaviors. However, as [25] points 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 single events in 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 must occur. In addition, we disallow nestings between these two fragments. We find that this does not limit the expressiveness of property specifications in practice.

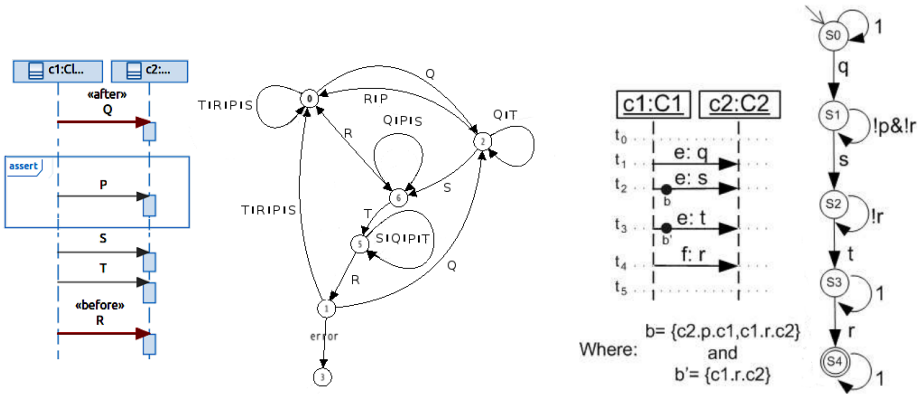
4 The Approach

4.1 The Rationale

To describe our proposal to a correct and straightforward property elucidation, we outline the motivations behind the choices we made, and how they differ from existing related approaches. As already stated, to bridge the gap between everyday practical software requirements specification and the the property patterns classification, several conversational tools have already been proposed.

While we follow on the idea of using a guiding questionnaire to incrementally refine various aspects of a requirement, we find the resulting artifacts (LTL formulas or graphical representations of finite state machines) from using the available ones (discussed in Section 2) not yet suitable for practical application in our context. For one, the practitioner must manually define the events to be associated with the placeholders when instantiating the template. To avoid potential errors, as well as reduce effort in specifications, we want to ideally stay in the same IDE used for modeling the system, and select only existing events that represent valid communication between components. In addition, we can already obtain [1] mCRL2 models from UML designs comprising sequence diagrams. In our experience, visual scenarios are the most suitable and commonly used means to specify the dynamics of a system. We believe that such a visual depiction of a scenario, more than finite state machines, improves the practitioner's understanding of the requirement as well. This is why we chose sequence diagrams as a property specification artifact too.

Most of the invented notations used by existing scenario approaches can fit well in UML 2.0 sequence diagrams. Profiles are a standard way to extend UML for expressing concepts not included in the official metamodel. In short, UML profiles consist of stereotypes that can be applied to any UML model,



(a) Sequence Diagram with monitor

(b) PSC with Büchi automaton [12]

Fig. 4. Scenarios for the precedence chain pattern

like classes, associations, or messages. We used this mechanism to apply the restrictions on the usage of *neg* and *assert*, as well as to distinguish between events presenting interval bounds and regular ones, from the patterns. As an example, Fig. 4a depicts the *precedence chain* pattern (with a *between-Q-and-R* scope), with the stereotypes applied to messages Q and R. The pattern expresses that event P must precede the chain of events S, T, always when the system execution is in the scope between events Q and R. We find this a much more intuitive scenario representation than the CHARMY/PSC one (Fig. 4b), for the same pattern. Notice that we do not have to specify constraints on past unwanted events, as they are automatically reflected in the μ-calculus formula, as long as there is a distinction between interval-marking messages, regular, mandatory, and forbidden ones. Also, the CHARMY/PSC notation presents the scenario in a negative form, using "f:" to explicitly mark an error message. Although any mature visual UML modeling tool can be used, we chose IBM's RSA environment [26]. One of the advantages is that RSA is built on top of Eclipse, making it relatively easy to extend the functionality. To this end, PASS is developed as an Eclipse plugin, using the lightweight UML profile, and as such is available (Fig. 6) to any Eclipse-based UML tool.

Furthermore, most visual scenario approaches cover the (state-based) LTL mappings and extensions of the pattern system. Event-based temporal logics have not received much attention. Even though the original pattern system does not cover μ-calculus, such mappings [27] have been developed by the CADP team. However, there are no pattern extensions available. These are adequate for action- or event-based systems, making them a good match for sequence diagrams, where communication between components is depicted. LTL logic is interpreted over Kripke structures, where the states are labeled with elementary propositions that hold in each state, while μ-calculus is interpreted over LTS, in which the transitions are labeled with actions that represent state changes.

Even though both are complementary representations of the more general finite state automata, conversions between them are not practical, as they usually lead to a significant state space increase. For example, the fact that a lock has been acquired or released can be naturally expressed by actions. Since state-based temporal logics lack this mechanism, an alternative is to introduce a variable to indicate the status of the lock, i.e., expose the state information. With such properties, LTS representations are more intuitive, and easier to query using event-based logics.

Given that communication among components proceeds via actions (or events) which can represent synchronous or asynchronous communication, property specification can be defined over sequences of actions that are data-dependent. Fortunately, μ-calculus is rich enough to express both state and action formulas, and provides means for quantification over data, which other formalisms lack. For example, with our approach, a practitioner can use a wild-card "*" to express that the property should be evaluated for all values that message parameters can carry. This allows us to use patterns which would otherwise make sense only for state-based formalisms. For example, the *universality* pattern is used to describe a portion of the system's execution which contains only states/events that have a desired property. Checking if a certain event is executed in every step of the system execution is not useful, so we adapted it in the context of μ-calculus.

Finally, for the purpose of on-the-fly verification, we provide an automatically generated mCRL2 monitor which corresponds to the property formula. 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. In addition, although mCRL2 offers direct model checking with μ-calculus and can provide feedback when the property fails to hold, this feedback is not at the level of the mCRL2 process specification. Using the monitor, the counter-example will be provided at the UML level.

4.2 Transforming a μ-calculus Formula Into a Monitor Process

A general model checking mechanism used with tools like SPIN is to construct a Büchi automaton for an LTL formula, which accepts exactly those executions that violate the property. A product of the model state space (typically a Kripke structure) and the Büchi automaton is then composed, and checked for emptiness. Although syntactically Büchi is similar to the finite-state monitor for which we aim, the difference lies in the acceptance conditions: a monitor accepts only finite runs of the system, while Büchi can trap infinite executions through detection of cycles, but potentially needs the entire state-space generated in the process. Runtime verification does not store the entire state space of a model, so it cannot detect such cycles. In addition, to expose state information, transitions in Fig. 5 are labeled with elementary propositions rather than actions (notice the $\wedge$ operator). As such, we cannot use existing tools for constructing Büchi automata with our approach.

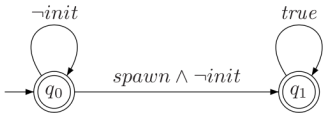


Fig. 5. A Büchi automaton [28]

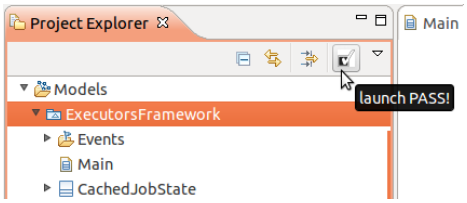


Fig. 6. Launching PASS from the Eclipse Project Explorer

Not every property can be monitored at runtime when only a finite run has been observed so far. Monitorable properties are those for which a violation occurs along a finite execution. This problem has been studied [29], and it is known that the class of monitorable properties is strictly larger than the commonly believed class of *safety* properties. However, an exact categorization of monitorable properties is missing. In particular, the definition of *liveness* requires that any finite system execution must be extendable to an infinite one that satisfies the property. By defining an end-scope of a property, we can also assert violations to *existence* patterns, which are typically in the *liveness* category. Such runtime monitor can also assert *universality* and *absence* patterns with or without scope combinations. We found that we are able to construct a monitor for about 50% of the property patterns.

We translate a fragment of the μ-calculus to mCRL2 processes which can subsequently serve as an observer processes for monitorable properties. We restrict to the following grammar:

$$\begin{aligned}
&\phi ::= b \mid \forall d{:}D.\phi \mid [\rho]\phi \mid \phi \wedge \phi \\
&\rho ::= \alpha \mid nil \mid \rho \cdot \rho \mid \rho + \rho \mid \rho^* \mid \rho^+ \\
&\alpha ::= a(d_1, \ldots, d_n) \mid \neg\alpha \mid b \mid \alpha \cap \alpha \mid \alpha \cup \alpha \mid \bigcap d{:}D.\ \alpha \mid \bigcup d{:}D.\ \alpha
\end{aligned}$$

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

$$\begin{aligned}
[nil]\phi &= \phi \\
[\rho^*]\phi &= [nil]\phi \wedge [\rho^+]\phi
\end{aligned}$$

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

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

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

$$
\begin{array}{ll}
\mathsf{TrR}_l(\alpha) & = \bigoplus_{a \in Act} (\sum d_a{:}D_a.\ \mathsf{Cond}_l(a(d_a), \alpha) \rightarrow a(d_a)) \\
\mathsf{TrR}_l(\rho_1 \cdot \rho_2) & = \mathsf{TrR}_l(\rho_1) \cdot \mathsf{TrR}_l(\rho_2) \\
\mathsf{TrR}_l(\rho_1 + \rho_2) & = \mathsf{TrR}_l(\rho_1) + \mathsf{TrR}_l(\rho_2) \\
\mathsf{TrR}_l(\rho^+) & = X(l) \qquad \textit{where } X(l) = \mathsf{TrR}_l(\rho) \cdot X(l) \textit{ is a recursive process}
\end{array}
$$

where $\bigoplus$ is a finite summation over all action names $a \in Act$ of the mCRL2 process 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:

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

Using the above translation, Fig. 4a shows monitor visualization next to the sequence diagram for the chain response pattern. Such a monitor can be placed in parallel a the system model, to perform runtime verification. Clearly, in the "worst" case, if the model is correct with respect to the property, all relevant model states will be traversed. In practice however, refutation can be found quickly after a limited exploration.

5 Case Study: DIRAC's Executor Framework revisited

DIRAC [30] is the grid framework used to support production activities of the LHCb experiment at CERN. All major LHCb tasks, such as raw data transfer from the experiment's detector to the grid storage, data processing, and user analysis, are covered by DIRAC. Jobs submitted via its interface undergo several processing steps between the moment they are submitted to the grid, to the point when their execution on the grid actualizes.

The crucial Workload Management components responsible for orchestrating this process are the *ExecutorDispatcher* and the *Executors*. Executors process any task sent to them by the ExecutorDispatcher, each one being responsible for a different step in the handling of tasks (such as resolving the input data for a job). The ExecutorDispatcher takes care of persisting the state of the tasks and distributing them amongst all the Executors, based on the requirements of each task. It maintains a queue of tasks waiting to be processed, and other internal data structures to keep track of the distribution of tasks among the Executors.

During testing, certain problems have manifested: occasionally, tasks submitted in the system would not get dispatched, despite the fact that their responsible Executors were idle at the moment. The root cause of this problem could not be identified by testing with different workload scenarios, nor by analysis of the generated logs. In [1] we manually formulated this problem as the following safety property:

```
[true* .
synch_call(1,ExecutorQueues,__queues,pushTask(JobPath,taskId,false)).
true*.
!(synch_call(1,ExecutorQueues,__queues,popTask([JobPath])))*.
synch_reply(1,ExecutorDispatcher,__eDispatch,
__sendTaskToExecutor_return(OK,0))]false
```

, meaning that a task pushed in the queue must be processed, i.e., removed from the queue before the ExecutorDispatcher declares that there are no more tasks for processing. Explicit model checking was not feasible in this case due to the model size (50 concurrent processes), so we resorted to writing a standard monitoring process set to run in parallel with the original model. With a depth-first traversal in mCRL2, we effectively discovered a trace [31] violating the property within minutes, and used our tool to import and automatically visualize the counter-example as a sequence diagram in RSA. Since the bug was reported and fixed, we wanted to check if the problem still persists after the fix, this time using PASS to elicit the property.

5.1 PASS: The Property ASSistant

To cope with the ambiguity of system requirements, PASS guides the practitioner via a series of questions to distinguish the types of scope and behavior as a relation between multiple events. By answering these questions, he is lead to consider some subtle aspects of the property, which are typically overlooked when manually specifying the requirement in temporal logic. The last part of the property (i.e. "before the ExecutorDispatcher declares that there are no more tasks for processing") is easily recognized as a scope restriction, which the user can choose by selecting the appropriate answer from the Scope Question Tree wizard page. This results in a *Before-R* scope restriction, where the actual communication can be selected by double-clicking the end-event placeholder (Fig. 7). This presents the user with a popup window with all the possible message exchanges in the model, so he can choose the actual message, in this case the reply message *__sendTaskToExecutor*. As already pointed out in [6], a closer examination of the patterns classification reveals some aspects which are not considered, and may lead to variants in the original scope and behavior definitions. For example, the definition of the *Before-R* scope requires that the event R necessarily occurs. This means, if R does not occur until the end of the run, the intent or behavior of the property could be violated, yet the property as a whole would not be violated unless P happens. In practice however, it is useful to introduce an *Until-R* variant for cases where the end-delimiter may not occur until the end of the system execution. This is captured by the last question in Fig. 7. Similar considerations have lead to new variants of the *After-Q-Until-R*

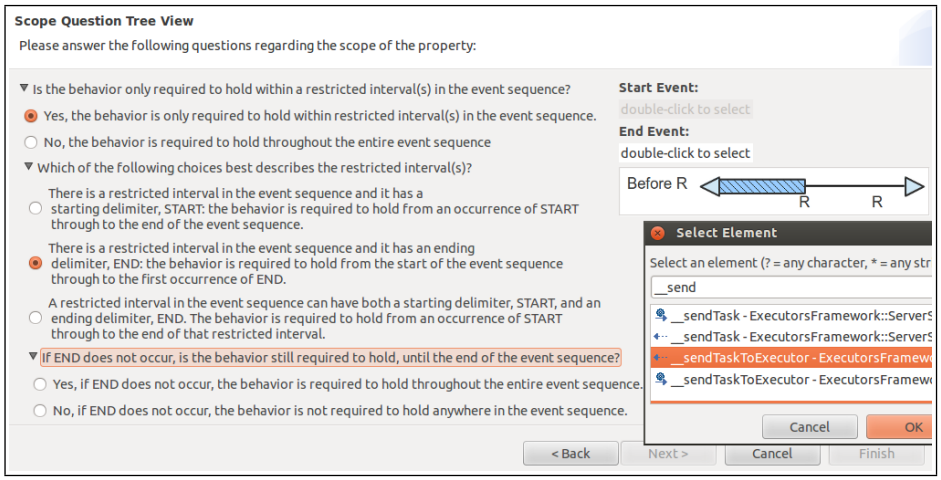


Fig. 7. Eliciting the scope for a property with PASS

and *After-Q-Before-R* patterns. For instance, whether subsequent occurrences of Q should be ignored, or should effectively reset the beginning of the interval in which the behavior is considered, are reflected in the questionnaire.

Due to space restrictions we do not show the Behavior Question Tree part of the wizard, although it is easy to elicit the behavior requirement as a *response* pattern ("a task *pushed* in the queue must be processed, i.e., *removed* from the queue"). The actual events of interest in this case are *pushTask* and *popTask*. Again, an extension of the pattern system allows for the user to decide whether the first event (the cause) must necessarily occur in the first place. Adding 4 scope and 2 behavior variations have lead to more than 100 ((5+**4**)∗(11+**2**)) new unique patterns to be chosen from.

At the end of the questionnaire, the user is presented (Fig. 8) with with a summary of the requested property, which can be reviewed before making the final decision. A μ-calculus formula pertaining the property is presented, along with the possibility to assign concrete parameter values that messages carry. Since the property should be evaluated for all possible values of the taskId's domain, a wildcard "*" can be used (as shown in the *pushTask* message's second parameter). This assignment results in a formula with a *forall* quantifier. In addition, a sequence diagram and a monitor process for the concerned property are generated, to be used in the final model checking phase. One thing worth noticing is the fact that our original manually constructed formula was not entirely correct, and as such could potentially produce spurious counter-examples. The general pattern template obtained with PASS is:

```
[(not R)*. P. (not (S or R))*. R] false
```

, which puts more restrictions on the behavior, while the original one was of the following form:

```
[true*. P. (not S)*. R] false
```

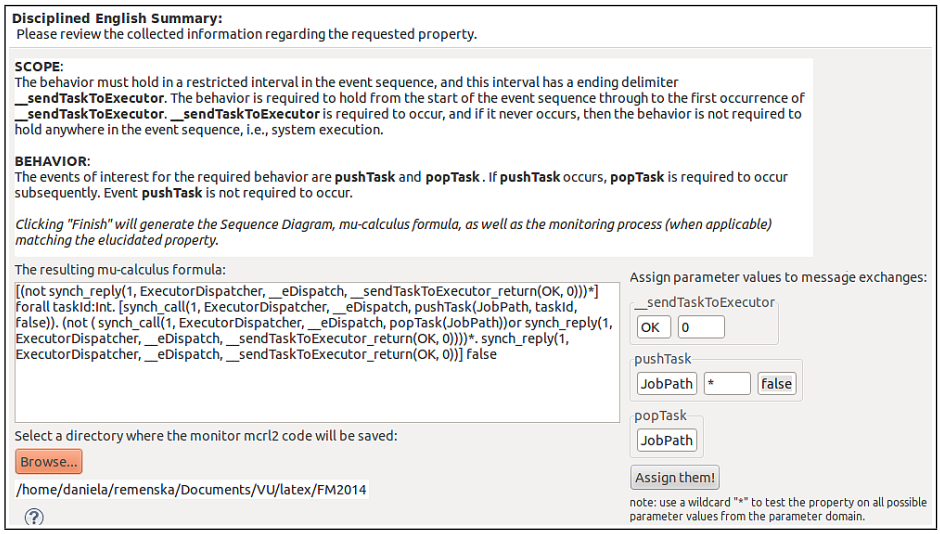


Fig. 8. Summary of the elicited property with PASS

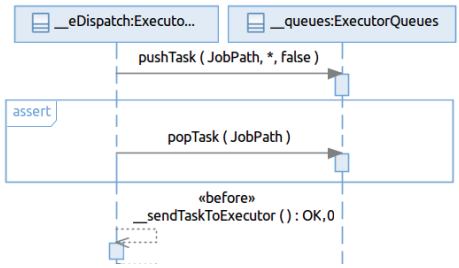


Fig. 9. Damn it

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. Cranen, S., Groote, J.F., Reniers, M.: A linear translation from LTL to the first-order modal μ-calculus. Technical report, Technical Report CSR-10-09, Depart-

ment of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands (2010)
23. Cranen, S., Groote, J.F., Reniers, M.: A linear translation from CTL* to the first-order modal μ-calculus. Theoretical Computer Science **412**(28) (2011) 3129–3139
24. OMG: UML2.4 Superstructure Specification http://www.omg.org/spec/UML/2.4/Superstructure.
25. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams (2007)
26. Terry, Q.: Visual Modeling with IBM Rational Software Architect and UML. Pearson Education India (2006)
27. Mateescu, R.: Property Pattern Mappings for RAFMC http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html.
28. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4) (September 2011) 14:1–14:64
29. Bauer, A.: Monitorability of omega-regular languages. CoRR **abs/1006.3638** (2010)
30. Tsaregorodtsev, A., et al.: DIRAC: A Community Grid Solution. Proc. CHEP'07
31. Remenska, D., Homburg, P.: The mCRL2⇔UML toolset https://github.com/remenska/NFM.