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. Developing correct concurrent software is challenging. Design errors can result in deadlocks, race conditions and livelocks, and discovering these is difficult. Approaches for automatically generating formal models from system designs have been proposed and successfully used in industry. A serious obstacle for an industrial uptake of rigorous analysis techniques such as model checking is the steep learning curve associated to the languages—typically temporal logics—used for specifying the application-specific properties to be checked. To bring the process of correctly eliciting functional properties closer to software engineers, we introduce PASS, a Property ASSistant wizard as part of a UML-based front-end to the mCRL2 toolset. 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. We revisit a case study from the Grid domain, using PASS to obtain the formula and monitor for checking the property.

1 Introduction

A challenge during the development of concurrent systems is detecting design errors, as such errors can cause deadlocks, livelocks, race conditions, starvation, etc. The sheer number of different executions and the inherent non-determinism in concurrent systems make complete testing of such software infeasible. Instead, more rigorous formal analysis techniques like model checking are required, which exhaustively analyze the behaviors of (an abstraction of) the system. Several competitive toolsets, such as SPIN, nuSMV, CADP and mCRL2 offer such analysis techniques. Despite the research effort, these tools are still not widely accepted in industry. One problem is the lack of expertise and the steep learning curve associated with becoming proficient in the underlying mathematical formalisms that must be used for describing models in these toolsets.

Bridging the gap between industry-adopted methodologies based on UML software designs and the aforementioned tools and languages, in [1] we devised a methodology for automatically verifying UML sequence and activity diagrams. Our prototype uses the mCRL2 language [2] and toolset as its backend, without users having to leave the UML domain, except when specifying application-specific properties.

While the mCRL2 toolset can automatically discover deadlocks and search for specific events, its model checking facilities require users to specify their application-specific properties in a data-enriched extension of the modal μ-calculus [3]. This is a highly expressive language, subsuming Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). A downside is that it is not very accessible and requires a high level of sophistication and mathematical background. Even so, even LTL and CTL are not widespread in industry, despite being considered simpler, more accessible languages. In fact, most requirements are written in natural language, and often contain ambiguities which make it difficult even for experienced practitioners to capture them accurately in any 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.

In an attempt to ease the use of temporal logic, in [4], a pattern-based classification was developed for capturing requirements and generating input to model checking tools. The authors observed that almost all (> 500) properties they surveyed can be mapped into one of several property patterns. Each pattern is a high-level, formalism-independent abstraction and captures a commonly occurring requirement. 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*, and *after* an event or state occurs; examples of behavior classification are: *absence* (an event or state should never occur during an execution) and *response* (an event or state must be followed by another event or state).

Although the patterns website [5] contains a collection of mappings for different target formalisms such as LTL and CTL, in practice practitioners have to fully understand the solutions before they can select and apply the appropriate ones. To mitigate this problem, several approaches [6–8] propose conversational tools for elucidating properties, based on the patterns. These tools guide users in selecting the appropriate pattern and optionally produce a formula in some target temporal logic. Alternative approaches [9–15] tackle the property specification problem by proposing new graphical notations for specifying properties. A downside of all these approaches is that they still expect users to have some expertise in temporal logic. Moreover, most approaches rely on state-based temporal logics. Such logics conceptually do not match the typical event-based UML sequence diagrams and activity diagrams, in which events represent methods calls or asynchronous communication between distributed components.

The objective of our work is to simplify the process of specifying functional requirements for event-based systems, *without* the need of expertise in temporal logic. We introduce PASS, a Property ASSistant which is a tool that guides and facilitates deriving system properties. Our starting point was the pattern system [4], which we extended with over 100 new property templates. The pattern templates instantiated with PASS have three notations: a natural language summary, a μ-calculus formula and a UML sequence diagram depicting desired behaviors. We utilized mCRL2's rich data extensions of the μ-calculus to express complex data-dependent properties. Lastly, we automatically generate monitors which can be used for property-driven on-the-fly model checking using the standard state-space exploration facilities of mCRL2. Our monitors are essentially sequence diagrams, acting as observers of message exchanges.

We deliberately chose to develop PASS as an Eclipse plug-in, as our strong motivation was to stay within an existing UML development environment, rather than use an external helper tool for this. We are convinced that this increases the tool accessibility by allowing software engineers to remain focused in the realm of UML designs. In addition, a tight connection 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 [16] Eclipse modeling API. We revisit a case study we did previously in [1], this time using PASS to obtain the formula and monitor for checking the property.

Structure. In Section 2 we survey related approaches, and outline their advantages and shortcomings. Section 3 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

PROPEL [6] is a tool that guides users in selecting the appropriate template from the patterns classification. PROPEL adds new patterns covering subtle aspects not addressed by the original patterns (such as considering the effect of multiple occurrences of a cause in a pattern); at the same time it omits patterns such as the universality, bounded existence and chain patterns. The resulting templates are represented using "disciplined natural language" and finite state automata rather than temporal logic expressions. Similar to PROPEL, the tools SPIDER [7] and Prospec [8] extend the original patterns but add compositionality. SPIDER is no longer maintained and available; the latest version of Prospec that we found and tested (Fig. 1 left) produces formulas in Future Interval Logic, not LTL as stated in [8].

Approaches that use a graphical notation for specifying properties come closest to the realm of modeling the system behavior. In [10], formulas are represented as acyclic graphs of states and temporal operators as nodes. Technically, the underlying LTL formalism is hidden from the user but the notation still closely resembles the formalism. As such, it is not very accessible. Another tool, called the TimeLine Editor [11] permits formalizing specific requirements using timeline diagrams. For instance, response formulas are depicted in timeline diagrams by specifying temporal relations among events and constraints. These diagrams are then automatically converted into Büchi automata, amenable to model checking with SPIN. Unfortunately the tool is no longer available. The CHARMY approach [9] presents a scenario-based visual language called Property Sequence Charts (PSC). Properties in this language are relations on a set of exchanged system messages. The language borrows concepts from UML 2.0 Sequence Diagrams and the tool uses the toolset SPIN as a backend for model checking generated Büchi automata [17]. The PSC notation uses textual restrictions for past and future events, placed as circles directly on message arrows (Fig. 1 right). A drawback of PSC is that it does not support asynchronous communication, which is omnipresent in concurrent systems. Furthermore, CHARMY is a standalone framework for architectural descriptions, not inter-operable with UML tools. As such, its use in industrial contexts is limited.

Among the UML-based tools are HUGO/RT [12] and vUML [13]. HUGO/RT 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

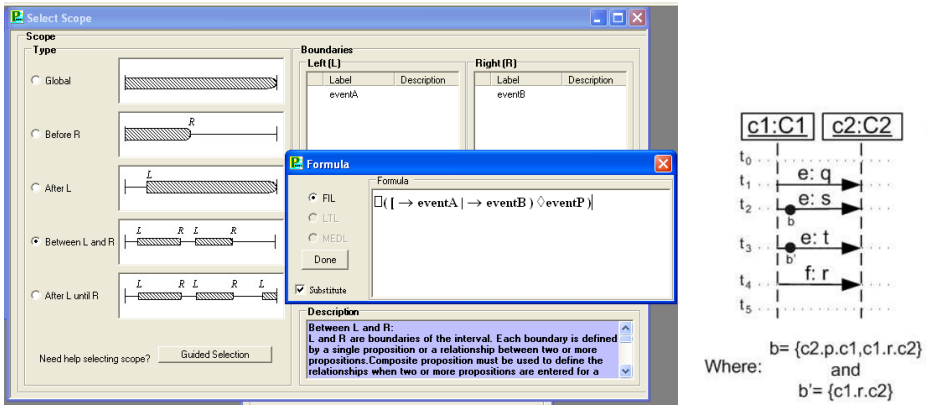


Fig. 1. Left: Prospec tool; right: CHARMY PSC graphical notation

translated together with the system model into Büchi automata for model checking with SPIN. The version we tested supports no asynchronous messages nor combined fragments. vUML [13] is, like HUGO/RT, essentially a tool for automating verifications of UML state machines. Properties must be specified in terms of undesired scenarios. The verification is based on ability to reach error states. This is inconvenient, as users must specify these manually. Live Sequence Charts (LSC) are also used [14, 15] as a graphical formalism for expressing behavioral properties. They can distinguish between possible (cold) and mandatory (hot) behaviors. For both, Büchi automata and LTL formulas are generated automatically from the diagrams. UML 2.0 sequence diagrams borrow many concepts from LSC, by introducing the *assert* and *negate* fragments capturing mandatory and forbidden behavior. However, LSCs lack many UML features.

Finally, there are several temporal extensions of UML's property language OCL. Resembling an OO programming language, OCL constraints quickly 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. Finally, OCL is by itself incapable of reasoning about temporal behavior. In [18] temporal modifiers @pre and @next are introduced for specifying past and future state-oriented constraints. In [19], a real-time constraint extension of OCL is proposed for models described by UML state machines; they claim to be able to describe all the existing patterns in these OCL expressions. To simplify constraint definitions with OCL, [20] proposes 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. Moreover, this work does not introduce means to specify temporal properties.

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:

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

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. 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:

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

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); individual 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 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 emanating from them known as *lifelines*. Each message sent between the lifelines defines a specific communication, synchronous or asynchronous. Messages are shown as horizontal arrows from the lifeline of the sender to the lifeline of the receiver instance.

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 2 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

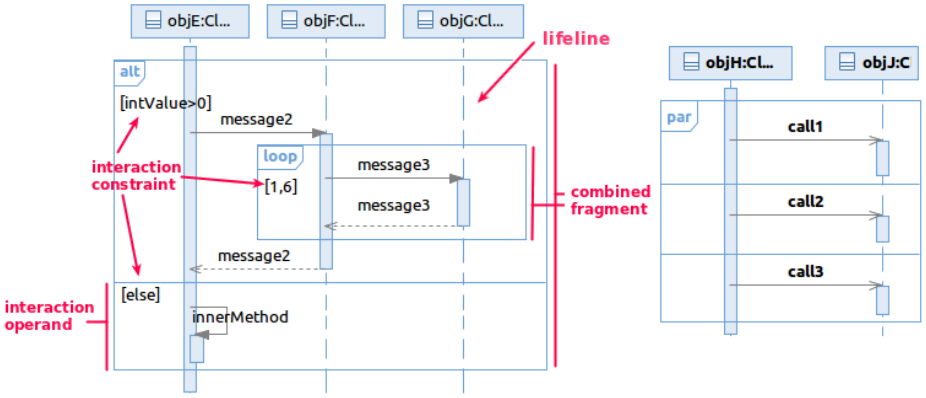


Fig. 2. Sequence diagrams with combined fragments

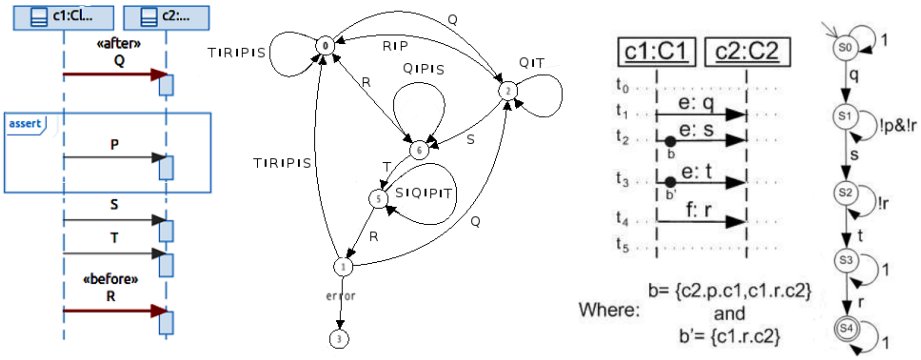
result in an invalid trace" suggests that the invalid set of traces for an *assert* fragment is its complement, i.e., the set of all other possible traces. Conversely, the standard also declares that the invalid set of traces are associated only with the use of a *neg* fragment, which is contradictory. 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 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.



(a) Sequence Diagram with monitor (b) PSC with Büchi automaton [9]

Fig. 3. Scenarios for the precedence chain pattern

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, 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. 3a 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. 3b), 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.

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 [26] 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-es, 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 indi-

cate 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.

Although any mature visual UML modeling tool can be used, we chose IBM's Rational Software Architect (RSA) environment. 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 plug-in, using the lightweight UML profile, and as such is available (Fig. 5) to any Eclipse-based UML tool.

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. 4 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.

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 [27], 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

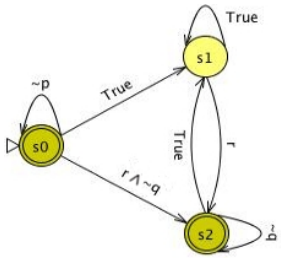


Fig. 4. A Büchi automaton

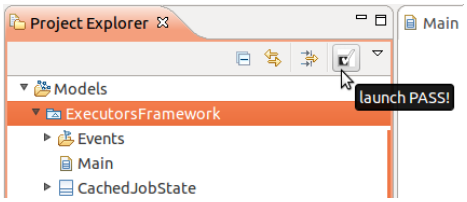


Fig. 5. Launching PASS from the Eclipse Project Explorer

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 core 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:

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

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

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

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. 3a shows monitor visualization next to the sequence diagram for the chain response pattern. Such a monitor can be placed in parallel with 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 [28] 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 point when they execute on the grid.

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 job's input data). The ExecutorDispatcher takes care of persisting the state of the tasks and distributing them amongst the Executors, based on the task requirements. 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, developers experienced certain problems: 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 [1] 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. 6). 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 *R* 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. 6. Similar considerations have lead to new variants of the *After-Q-Until-R* 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

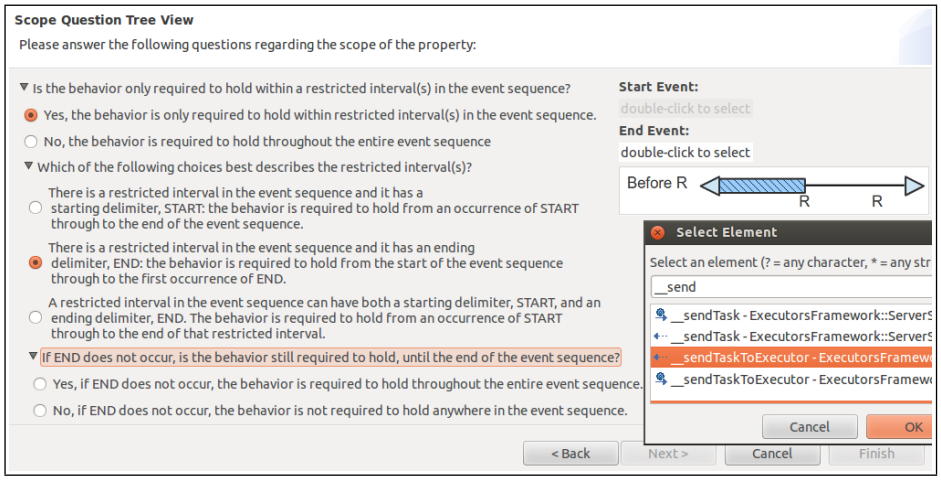


Fig. 6. Eliciting the scope for a property with PASS

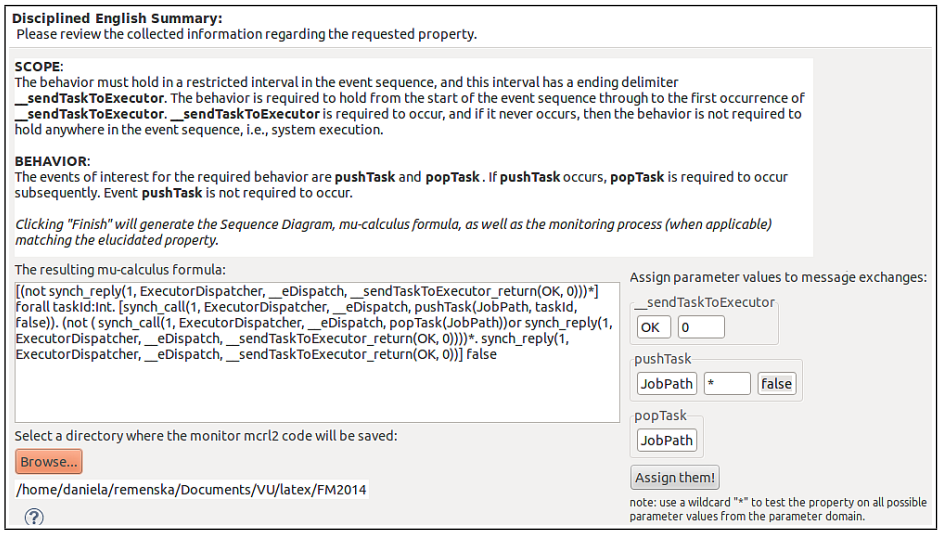


Fig. 7. Summary of the elicited property with PASS

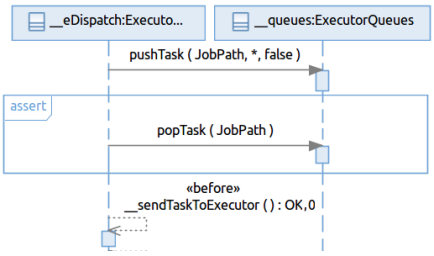


Fig. 8. A sequence diagram for the property

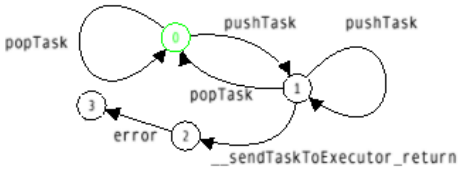


Fig. 9. A monitor for the property

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. 7) with a summary of the requested property, which can be reviewed before making the final decision. A μ-calculus formula pertaining to 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 second parameter of *pushTask*). This assignment results in a formula with a *forall* quantifier. In addition, a sequence diagram (Fig. 8) and a monitor process in mCRL2 (visualized in Fig. 9 without the data, for clarity) are generated, to be used in the final model checking phase. It is worth noticing 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
```

Using the generated monitor, we performed runtime verification on the corrected model. We linearized the model with the mCRL2 toolset, and used LTSmin's symbolic reachability tool [29] for efficient state space exploration. LTSmin is language-independent, and can be used as an mCRL2 model checking back-end. Taking less than 2 hours, the symbolic equation system explorer finished the traversal (2.85 million states) without discovering an *error* step, effectively concluding that the property holds. The PASS tool, along with model and the monitor of this case study is available at [30].

6 Conclusions and future work

In an effort to automate more aspects of formal verification of distributed systems, we introduced PASS, a Property ASSistant that brings the process of correctly specifying functional properties closer to software engineers. Through a series of questions, the practitioner can consider subtle aspects about a property which are often overlooked. Motivated by the wish to stay within an existing UML development environment, rather than use an external helper tool, PASS was developed as an Eclipse plug-in, thus keeping a strong relationship between the model elements and the property template ones. Our approach to specifying properties is based on the pattern system [4], which we extended with over 100 new pattern variations for the event-based μ-calculus formalism. Besides offering a natural language summary of the elicited property, a μ-calculus formula and a UML sequence diagram is provided, depicting the desired behavior. In addition, PASS automatically generates monitors to be used for efficient property-driven runtime verification using the mCRL2 toolset. We believe that automating the property specification process, while keeping practitioners in their familiar environment, should lead to more active adoption of methods for formal analysis of designs. We revisited a case study from the grid domain, and discovered that despite a reasonably good understanding of μ-calculus, our previously manually defined property was in fact not fully correct. Using the monitor, we performed runtime verification, which in the end resorted to full exploration of the state space, and did not disprove the property.

Besides instantiating pattern templates, part of our ongoing work is to define a methodology that would allow the experienced practitioner to directly write sequence diagrams expressing requirements, based on which a μ-calculus formula and a monitor would be provided. Automating performance analysis is on our road-map as well. Profiles are provided as UML extension mechanisms, which allow annotating models with quantitative information (such as expected execution time, resource usage etc.), and this mechanism would permit assessing aspects of the system such as efficiency and reliability. We can use the same target formalism for enhancing the models with such quantitative information. The CADP toolset [31] for analysis of stochastic models is well integrated with mCRL2 for this purpose.

References

1. Remenska, D., et al.: From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems. In: NASA Formal Methods. (2013)
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. (2005)
4. Dwyer, M.B., et al.: Patterns in property specifications for finite-state verification. In: Proc. ICSE'99
5. Dwyer, M.B., et al.: Property Specification Patterns http://patterns.projects.cis.ksu.edu.
6. Smith, R.L., et al.: Propel: an approach supporting property elucidation. In: Proc. ISCE'02
7. Konrad, S., Cheng, B.H.: Facilitating the construction of specification pattern-based properties. In: Proc. RE'05, IEEE
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. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Eng. (2007)
10. Lee, I., Sokolsky, O.: A Graphical Property Specification Language. In: Proc. of 2nd IEEE Workshop on High-Assurance Systems Engineering. (1997)
11. Smith, M.H., et al.: Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs. In: Proc. RE'01
12. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Proc. MoDELS'06
13. Lilius, J., Paltor, I.P.: vUML: a Tool for Verifying UML Models. In: Proc. ASE'99
14. Kugler, H., et al.: Temporal logic for scenario-based specifications. In: Proc. TACAS'05
15. Baresi, L., Ghezzi, C., Zanolin, L.: Modeling and Validation of Publish/Subscribe Architectures. In: Testing Commercial-off-the-Shelf Components and Systems
16. The Eclipse Foundation: Eclipse Modeling MDT-UML2 component www.eclipse.org/uml2/.
17. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proc. ASE'01
18. Ziemann, P., Gogolla, M.: An Extension of OCL with Temporal Logic. In: Critical Systems Development with UML. (2002)
19. Flake, S., Mueller, W.: Formal Semantics of Static and Temporal State-Oriented OCL Constraints. Proc. SoSyM'03
20. Ackermann, J., Turowski, K.: A library of OCL specification patterns for behavioral specification of software components. In: Proc. CAiSE'06
21. Emerson, E.A.: Model checking and the Mu-calculus. In: DIMACS Series in Discrete Mathematics, American Mathematical Society (1997)
22. Cranen, S., Groote, J.F., Reniers, M.: A linear translation from LTL to the first-order modal μ-calculus. Technical report, CSR-10-09, TU/e 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 (28) (2011)
24. OMG: UML2.4 Superstructure Spec. http://www.omg.org/spec/UML/2.4/Superstructure.
25. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software & Systems Modeling **7** (2008)
26. Mateescu, R.: Property Pattern Mappings for RAFMC http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html.
27. Bauer, A.: Monitorability of omega-regular languages. CoRR **abs/1006.3638** (2010)
28. Tsaregorodtsev, A., et al.: DIRAC: A Community Grid Solution. Proc. CHEP'07
29. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: Theoretical Aspects of Computing. (2008)
30. Remenska, D., Willemse, T.A.C.: PASS: Property ASSistant tool for Eclipse https://github.com/remenska/PASS.
31. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Proc. TACAS'11