Property Patterns for Runtime Monitoring of Web Service Conversations*

Jocelyn Simmonds[1], Marsha Chechik[1], Shiva Nejati[1],
Elena Litani[2], and Bill O'Farrell[2]

[1]University of Toronto, Toronto ON M5S3G4, Canada
[2]IBM Toronto Lab, Markham ON L6G 1C7, Canada
{jsimmond,chechik,shiva}@cs.toronto.edu
{elitani,billo}@ca.ibm.com

Abstract. For a system of distributed processes, correctness can be ensured by statically checking whether their composition satisfies the properties of interest. However, web services are distributed processes that *dynamically* discover properties of other web services. Since the overall system may not be available statically and since each business process is supposed to be relatively simple, we propose to use runtime monitoring of conversations between partners as a means of checking behavioral correctness of the entire web service system. Specifically, we identify a subset of UML 2.0 Sequence Diagrams (SD) as a property specification language. We show how our language can be used to specify the patterns in the Specification Property System (SPS) [1]. By formalizing this subset using automata, we can check finite execution traces of web services against various complex properties. Finally, we discuss our experience using our language for runtime monitoring of an existing application, and conclude with a description of existing tool support.

1 Introduction

Web services are collections of components which discover and bind to other components using published interfaces, with support of Service-Oriented Architectures (SOA). The goal of SOA is to increase the flexibility of business interactions. Each web service component can be written in a traditional compiled language such as Java®, or in an XML-centric language such as BPEL [2].

Consider, for example, a web-based Loan Application system (LA), distributed as a sample application with the IBM® Websphere® Integration Developer v6.0.2. Users enter loan application information (name, taxpayer id, loan amount) through a web page, and are eventually informed of the status of their applications. The LA workflow first checks the user's credit score and declines a loan if the user has a bad credit score, i.e., less than 750. If the credit score is good, the workflow then checks the loan amount: loans for $50,000 or less are automatically approved; loans for larger amounts are earmarked for manual approval.

The workflow diagram in Fig. 1(a), which is described as a BPEL specification, shows high level steps that are executed in a loan application system, and Fig. 1(b) shows an assembly diagram describing how the main process of the LA system invokes its *partners*, such as `CreditCheck` (implemented in Java), rule groups (`LoanLimit`), or human tasks (`FollowUpDeclinedApp`, `CompleteTheLoan` and `ProcessThe-Application`). Specifically, the `CheckCredit` activity in Fig. 1(a) invokes the

* © Copyright 2008, International Business Machines. All Rights Reserved.

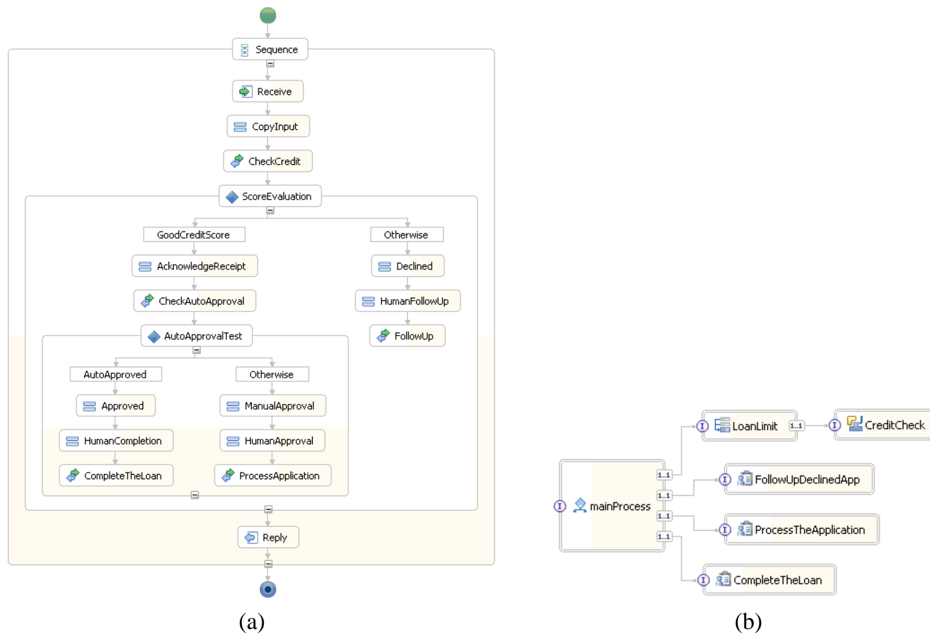


Fig. 1: The LA system: (a) workflow, describing the high level steps of the LA system; (b) assembly diagram, describing how the main process of the LA system interacts with its partners.

`CreditCheck` partner in Fig. 1(b), and the conditional activities `ScoreEvaluation` and `AutoApprovalTest` invoke the `LoanLimit` partner. The partners in Fig. 1(b) implement the following functions: `CreditCheck` uses the taxpayer id to retrieve the corresponding credit score; the `LoanLimit` rule group checks the credit score and the loan amount. The human tasks represent the following application results: declined, approved and manual approval, respectively.

Since the LA system is a composition of several distributed business processes, its correctness depends on the correctness of its partners and their interactions. For example, the system should guarantee that every request is eventually acknowledged and none are lost or blocked indefinitely, or that loans are only given to customers with a good credit score. However, in the provided LA application, the `CreditCheck` module assigns a credit score at random, without using the customer id, thus preventing the overall system from satisfying this property.

Since each web service is a relatively simple process, analysis can concentrate on the message exchange between partners – their *conversations*. While static techniques for checking partner composition against properties of interest, such as [3–7], are appealing, they have a number of limitations: the problem is decidable only under certain conditions [8], since the partners communicate via infinite-sized channels, and existing techniques are unable to deal with complex message interactions and heterogeneity of partners.

Instead, we concentrate on the dynamic analysis via runtime monitoring. Unlike the work of [9–11], our approach is to create an industrial-strength (with partnership with the IBM Toronto Lab) monitoring framework that is non-intrusive, supports the dynamic discovery of web services, deals with synchronous and asynchronous communication and partners implemented in different languages, allows for specifying and efficient monitoring of a variety of temporal behaviour, and is usable by practitioners.

P_1	The loan amount must be always greater than zero.
P_2	The credit score should eventually be checked if the loan amount is greater than zero.
P_3	A loan cannot be granted if the loan amount is less than or equal to zero.
P_4	After checking that the applicant has a good credit score, a loan cannot be granted if the loan amount is less than or equal to zero.
P_5	No one can get a loan without first going through a credit check.

Table 1: Some properties of the LA system.

In [12], we chose a subset of UML 2.0 Sequence Diagrams [13] as our specification language. This language allows specification of events, has an explicit emphasis on components, and is able to deal with positive and negative scenarios of interaction as well as global properties. We have shown that this subset is sufficiently expressive for capturing *safety* (nothing bad can ever happen) and *liveness* (something good will eventually happen) properties. For example, for the LA system described earlier, possible safety and liveness properties are P_1 and P_2, respectively (see Table 1). While liveness properties are not monitorable in general, they can be effectively checked for web services with finitely terminating behaviours. For example, we can check whether the LA process terminates without giving feedback to the customer. Specifically, for finite behaviours, liveness can be seen as the dual of safety: liveness properties are expressed as finite positive traces, and safety properties as finite negative traces.

To enable monitoring, [12] formalized the chosen subset of Sequence Diagrams using finite-state automata. These automata are then used in the implementation of our non-intrusive monitoring framework which runs in parallel with the system being monitored, intercepting events from web service conversations. The resulting system enables conformance checking of finite execution traces against their specifications expressed in our subset of Sequence Diagrams.

In [12], we showed that *assert* and *negate* operators in UML 2.0 [13] can be used to describe *simple* safety and liveness properties, namely, invariants, e.g., P_1 in Table 1, and request-response properties, e.g., P_2 in Table 1. However, in [12], we assumed that only one assert or negate operator can be applied to a sequence diagram, always as the outermost operator. To conveniently specify and verify various system properties that arise in practice, e.g., P_4 and P_5 in Table 1, we need a more expressive language. In this paper, we extend this language by allowing arbitrary nested applications of *assert* and *negate* operators. Furthermore, we enriched the language with several operators, adopted from UML 2.0 [13] and other scenario-based languages [14]. Examples of these operators include *critical*, *ref* (which allows to reuse portions of sequence diagrams in other diagrams) and message complementation. We show that the resulting language can not only be converted into finite-state automata for monitoring, but is also sufficiently expressive to capture a wide variety of frequently used properties, captured and catalogued in the Specification Pattern System (SPS) [1]. This approach also gives basis for tool support to enable usable specification of runtime conversations.

The rest of this paper is organized as follows. We describe syntax of the subset of UML 2.0 sequence diagrams used for expressing properties of webservice conversations in Section 2. Such properties are then converted into monitoring automata using the techniques discussed in Section 3. We then show how our specification language can be used to specify the complete set of temporal logic property patterns in Section 4. We describe the implementation of the runtime monitoring framework and report on the result of applying our framework to the LA system in Section 5. Finally, we conclude

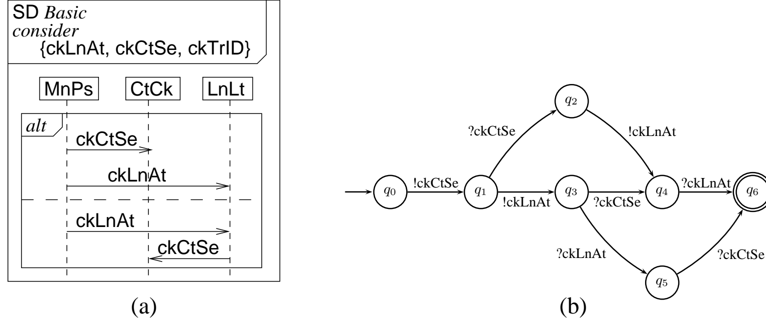


Fig. 2: (a) An SD describing a scenario of the LA example; and (b) the NFA corresponding to the first argument of the *alt* operator in Fig 2(a).

the paper in Section 6 with a summary of the paper, comparison with related work, and an outline of future research directions.

2 A Language for Specifying Conversations

We choose a subset of UML 2.0 Sequence Diagrams as our language for specifying web service conversations. Sequence Diagrams [13] is a popular formalism for modeling behavioural scenarios by describing sequences of messages communicated between different objects over time. Sequence Diagrams have two dimensions: vertical, representing time, and horizontal, representing objects. Each object is illustrated by a rectangle with a vertical dashed line, called a *lifeline*. Lifelines are connected by horizontal arrows denoting messages that are sent from one object to another, synchronously or asynchronously.

An example sequence diagram describing a scenario of the LA system is shown in Fig. 2(a). The diagram contains three objects, `MnPs`, `CtCk`, and `LnLt`. Object `MnPs` corresponds to the main workflow of the LA system, and `CtCk` and `LnLt` correspond to components `CreditCheck` and `LoanLimit`, respectively. The diagram in Fig. 2(a) shows two alternative scenarios: In the first alternative, `MnPs` first sends a check credit score request, i.e., `ckCtSe`, to `CtCk` and then a check loan amount request, i.e., `ckLnAt`, to `LnLt`. In the second alternative, `LnLt` receives a check loan amount request from `MnPs`. Since the credit score has not yet been checked, `LnLt` sends a check credit score request to `CtCk`.

In UML 2.0, Sequence Diagrams can be augmented by a large number of operators to capture various complex scenarios. We use the operators described below in our property specification language. We refer to our language as *SD*.

Compositional operators: Operators *parallel (par)*, *alternatives (alt)*, *strict sequencing (strict seq)*, and *weak sequencing (weak seq)* are used to combine two SDs based on standard notions of compositions. The operator *loop* is used for repeating the scenario described by an SD multiple times, *opt* – for denoting an optional scenario, equivalent to *alt* with only one argument. Finally, *critical* is used to ensure atomicity of the enclosed sequence.

Alphabet changing operators: Operators *consider* and *ignore* are used for modifying the communicating alphabet of SDs.

Assertion and negation operators: Operators *assert* and *negate* allow users to express mandatory and forbidden system scenarios, respectively.

Interaction use operator: SDs can be shared by references, using the *ref* operator. This is a shorthand for copying the contents of the referred SD where the *ref* operator occurs, and is a new feature in UML 2.0.

To describe system scenarios, we often need to express complementation of individual or a group of messages. Since arbitrary and nested use of the *negate* operator is inconvenient, we use an operator *message complementation*, originally introduced in the Property Sequence Charts (PSC) language [14], to negate individual or sets of messages. We denote the complement of a message m by $\neg m$ and define it as the set of all messages that are potentially exchanged between objects of the system except for m.

3 From SDs to Automata

We define the formal semantics of SD by translating it into *non-deterministic finite automata (NFAs)*, following the approach of [15]. This translation allows us to not only formalize our language but also to study its expressiveness. Specifically, in [12], we have shown that certain scenarios in SD can be captured by particular forms of NFAs known as *Safe* and *Live* automata [16], indicating that SD is capable of expressing safety and liveness properties. In what follows, we briefly review the translation of basic sequence diagrams and the operators described in Section 2 into NFA. We then discuss that the *negate* and *assert* operators allow us to express safety and liveness properties, respectively.

Basic sequence diagrams, i.e, diagrams describing a sequence of events without any additional operator, can be translated into NFAs using the procedure in [15]. Consider the scenario in the first argument of the *alt* operator in Fig 2(a). This basic sequence diagram shows that `MnPs` first sends event `ckCtSe` to `CtCk` and then event `ckLnAt` to `LnLt`. We denote the sending of a message m by $!m$ and its receiving by $?m$. Thus, the set of events of the sequence diagram in Fig. 2(a) is {`!ckCtSe`, `?ckCtSe`, `!ckLnAt`, `?ckLnAt`}. Intuitively, lifelines and message arrows in a sequence diagram define a partial order on the set of events of that diagram. Given a basic sequence diagram S, an NFA A_S is equivalent to S iff A_S accepts exactly the set of traces that respect the partial order of S. For example, the automaton A_S corresponding to the scenario in the first argument of the *alt* operator in Fig 2(a) is shown in Fig 2(b).

The semantics of the compositional operators can be given in terms of the standard operations defined on NFAs (e.g., see [17]). For example, *alt* corresponds to the union operator; *strict seq* corresponds to the sequential composition operator; and *loop* corresponds to the Kleene star operator.

Operators *consider* and its dual, *ignore*, are used to change the set of communicating alphabets of an SD. Both of them receive an SD S and a set of events E as input, but *consider* adds the elements in E to the set of events of S, whereas *ignore* removes the elements in E from the set of events of S.

We can specify a critical region in a sequence diagram using the *critical* operator. A critical region means that the traces of the region cannot be interleaved by other messages and thus should be treated atomically. We treat this operator to mean that if the first message of the critical region is observed, then the rest of the behavior must be observed as well.

Let S be an SD enclosed within a *critical* operator, and let A_S be the automaton for S. The automaton for *critical* S is obtained by (1) adding a self-loop at the initial

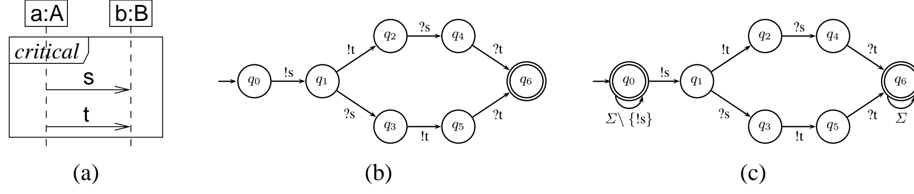


Fig. 3: (a) A basic SD enclosed by a *critical* operator; and its corresponding NFAs: (b) before applying *critical*; (c) after applying *critical*.

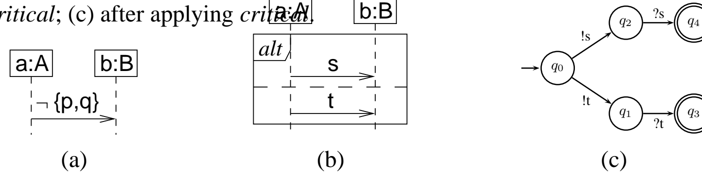


Fig. 4: (a) An SD with message complementation; (b) the same SD after eliminating the *complement* operator, if the underlying alphabet Σ is $\{\texttt{p}, \texttt{q}, \texttt{s}, \texttt{t}\}$; and (c) its corresponding NFA.

state, q_0, of A_S labelled by $\Sigma \setminus \{e \mid q_0$ has an outgoing transition on $e\}$, and (2) making the initial state final. The self-loop at the initial state allows the automaton to wait for a satisfying run to begin. For example, Fig. 3(a) shows a sequence diagram with a critical operator, and Fig. 3(c) – its corresponding automaton.

The operator *ref* is used for sharing portions of SDs between several others. Our treatment of *ref* is to inline the SD being referenced, applying the necessary translation rules to the result in order to obtain the corresponding NFA, as illustrated in Fig. 5.

The message complement operator has been adopted from [14]. If Σ is the set of messages exchanged in an SD, and $m \in \Sigma$, then $\neg m$ is $\Sigma \setminus \{m\}$. For a set $\{m, n\}$ of messages, $\neg\{m, n\} = \Sigma \setminus \{m, n\}$. For example, let $\Sigma = \{\texttt{p}, \texttt{q}, \texttt{s}, \texttt{t}\}$. Then, $\neg\texttt{p} = \{\texttt{q}, \texttt{s}, \texttt{t}\}$ and $\neg\{\texttt{p}, \texttt{q}\} = \{\texttt{s}, \texttt{t}\}$. This operator can be indirectly expressed in UML 2.0 by encapsulating each message in the complement set of a given message into an *alt* operator. Formally, let $S \subseteq \Sigma$ be a set of messages. We replace $\neg S$ by an SD fragment in which operator *alt* is applied to single messages in $\Sigma \setminus S$. For example, consider the SD in Fig. 4(a) with message, $\neg\{\texttt{p}, \texttt{q}\}$, and let $\Sigma = \{\texttt{s}, \texttt{t}, \texttt{p}, \texttt{q}\}$. This SD is equivalent to that in Fig. 4(b) where $\neg\{\texttt{p}, \texttt{q}\}$ is replaced by an *alt* fragment in which `s` and `t` are two alternative messages. The NFA for the sequence diagram without message complement operators can be generated in a straightforward way following the translation for the *alt* operator (see Fig. 4 (c)).

Representing safety properties. To describe a safety property, we enclose an SD S within a *negate* operator to indicate that the scenario represented by S is a forbidden one, and therefore, a safe system should never produce this scenario [16]. For example, the SD in Fig. 6(a) describes the safety property P_1 in Table 1. To obtain the automaton for *negate* S, we first derive an NFA A_S for SD S and add a self-loop labeled Σ at the initial state of A_S (see Figure 6(b)). We then determinize and complement A_S to obtain an automaton for *negate* S. The resulting automaton accepts every trace that does not contain the sequence `!ckLnAt?ckLnAt!lnAtNo?lnAtNo`.

Representing liveness properties. The process for constructing liveness automata is dual of that for constructing safety automata. To describe a liveness property, we enclose an SD S within an *assert* operator to indicate that the scenario represented by S is the

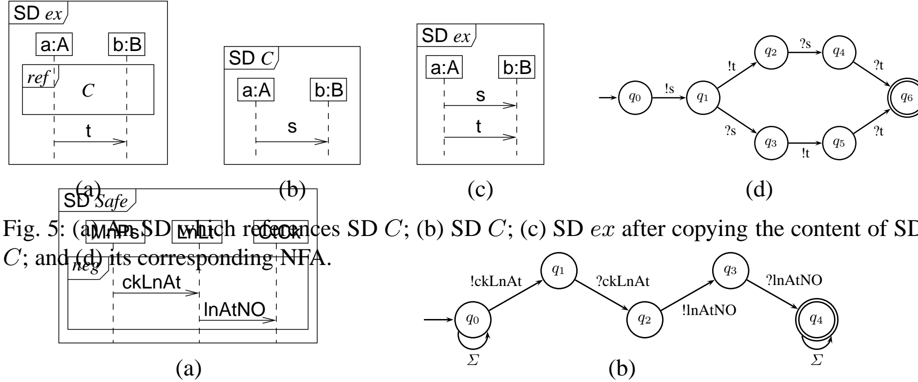


Fig. 5: (a) An SD which references SD C; (b) SD C; (c) SD ex after copying the content of SD C; and (d) its corresponding NFA.

Fig. 6: (a) A Safe SD describing P_1 in Table 1 and (b) its corresponding NFA (before complementation and determinization).

only valid continuation of any system behavior [13]. For example, the SD in Fig. 7(a) describes the liveness property P_2 in Table 1. After deriving the NFA A_S for SD S and adding a self-loop labelled Σ at its initial state, the automaton for *assert* S is obtained by determinizing A_S. Intuitively, A_S is a liveness automaton if every trace recognized by it includes the live part completely. Fig. 7(b) shows the automaton corresponding to the SD in Fig. 7(a) before the determinzation step. This automaton accepts every trace that contains the entire sequence `!ckLnAt?ckLnAt!retLnAtStat?retLnAtStat`.

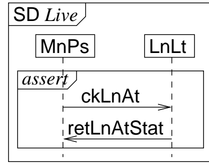
Complexity of the Translation. The size of an automaton A_S corresponding to a basic sequence diagram S is $O(n^k)$ where n is the number of events and k is the number of processes [15]. Applying the sequence diagram operators does not cause a significant increase in the size of the resulting automata except for the *negate* and *assert* operator that involve a determinzation step which can be exponential in the number of states of A_S. However, we note that in practice, the automata we have generated are relatively small, less than 9 states and 30 transitions [12]. Obviously, it remains to be seen whether the approach remains feasible for larger web service systems and more complex properties.

4 SD Templates for Temporal Logic Property Patterns

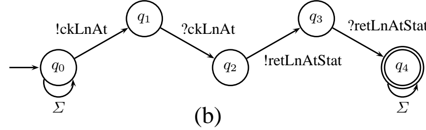
In this section, we introduce several templates expressed in the SD language for describing temporal logic property patterns [1]. We first provide an overview of these patterns in Section 4.1. We then describe our templates in the SD language in Section 4.2 and show how they can encode the property patterns.

4.1 Temporal Logic Property Patterns

The *Specification Pattern System* (SPS), proposed by Dwyer et al. [18], is a pattern-based approach to the presentation, codification, and reuse of property specifications. The system allows patterns like "event P is absent between events Q and S" or "S precedes P between Q and R" to be easily expressed in and translated between linear-time temporal logic (LTL) [19], computational tree logic (CTL) [19] and other state-based and event-based formalisms. SPS has been advocated as a standard tool for measuring



(a)



(b)

Fig. 7: (a) A Live SD describing P_2 in T… (before deter-minization).

Property Patterns

Occurrence
- Absence
- Universality
- Existence
- Bounded Existence

Order
- Precedence
- Response
- Chain Precedence
- Chain Response

Fig. 8: Pattern Hierarchy.

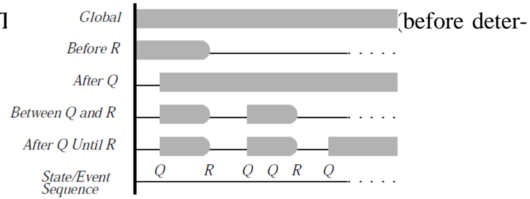


Fig. 9: Pattern Scopes.

the practical usefulness and expressive power of specification languages, e.g., [14] and [20].

The property patterns are organized into a hierarchy based on the kinds of system behaviors they describe (see Fig. 8): *Occurrence* patterns talk about the occurrence of a given event/state during system execution, and *Order* patterns specify relative order in which multiple events/states occur during system execution. The patterns are described below in detail:

Absence	An event does not occur within a given scope;
Existence	An event must occur within a given scope;
Bounded Existence	An event can occur at most a certain number of times within a given scope;
Universality	An event must occur throughout a given scope;
Response	An event must always be followed by another within a scope;
Response Chain	A chain of events must always be followed by another chain of events within a scope;
Precedence	An event must always be preceded by another within a scope;
Precedence Chain	A chain of events must always be preceded by another chain of events within a scope.

Each pattern is associated with *scopes* – the regions of interest over which the pattern must hold. There are five basic kinds of scopes (depicted in Fig. 9):

Global	The entire program execution;
Before R	The execution up to event R;
After Q	The execution after event Q;
Between Q **and** R	All parts of the execution between events Q and R;
After Q **until** R	Similar to *between*, except that the designated part of the execution continues even if the second event does not occur.

For example, consider a property that says between every *enqueue* and *empty* messages, there must be a *dequeue* message. This property falls into the "Existence" pattern group because it indicates the occurrence of an event within a scope. The scope of this property is that of "Between" shown in Fig 9. Using the property pattern catalogue,

the LTL formalization of the above property is as follows: $\Box((enqueue \wedge \neg empty) \Rightarrow (\neg empty W(dequeue \wedge \neg empty)))$.

4.2 Mapping Property Patterns to SDs

In this section, we describe mappings for property patterns in SD. By providing a standardized library of template SDs (see Fig 11), we hope to improve the usability of our specification language. In the mappings, let symbols `p`, `q`, `s`, and `t` be messages sent between partners. Selected mappings are described below; the remainder can be found in Appendix A.

Absence: message `p` cannot occur in a given scope. This can be expressed as a simple safety SD, as shown in Fig. 11(a).

Existence: message `p` must occur in a given scope. This can be expressed as a simple liveness SD, as shown in Fig. 11(b).

Until: a sequence $\mathtt{p}^*$ of messages occurs until the first occurrence of message `q`, in a given scope (see Fig. 11 (h)). This pattern is not part of the SPS; however, it is used to specify the precedence patterns. To express this pattern in the SD language, we note that this pattern, which can be formalized using a single until property [19], can be refuted in two ways only: a) `p` never occurs, or b) after seeing a finite number of `p` messages (expressed using *loop 1, n* in Fig 11(h)), neither a `p` nor a `q` message occurs (expressed as $\neg\{\mathtt{p}, \mathtt{q}\}$ in Fig 11(h)).

Precedence: message `s` (cause) precedes message `p` (effect), as shown in Fig. 11 (i). Note that this pattern allows the cause part to occur without the effect part. We describe this pattern in SD by expressing the two possible cases that this pattern specifies: a) `p` never occurs, or b) `p` never occurs before `s`. The first case corresponds to checking *absence* of `p`; the second – to checking $\neg\mathtt{p}\ U\ \mathtt{s}$, since we want to be sure that *no* `p` messages are sent before the first `s` message.

In the SDs in Fig. 11, symbols `p`, `q`, `s`, and `t` can denote more complex SDs, not just individual messages. In this case, we treat these symbols as place holders and use a *ref* operator to the SDs that should be inserted in place of these symbols. Also, in these cases, message complementation is replaced by negation. In Section 4.4, we provide detailed examples of how these patterns are used to specify properties of the LA system.

4.3 Mapping Property Scopes

In this section, we describe mappings for property scopes in SD. Scopes are used to restrict the traces over which a property will be monitored. Fig. 10 shows how the *ref* operator is used to introduce scope delimiters for the different scopes. Scope delimiters can be simple messages or more complex scenarios. For example, to apply the **Before** R scope to a property, the scope delimiter *R* is inserted after the property we wish to verify (see Fig. 10(a)). In the case of **After** Q scope, the delimiter is inserted before the property (see Fig. 10(b)). Finally, both the **Between** (see Fig. 10(c)) and **After-until** (see Fig. 10(d)) scopes add before/after delimiters. In the **After-until** scope, the property is valid even if the until part does not occur. Therefore, the second delimiter in this scope is optional.

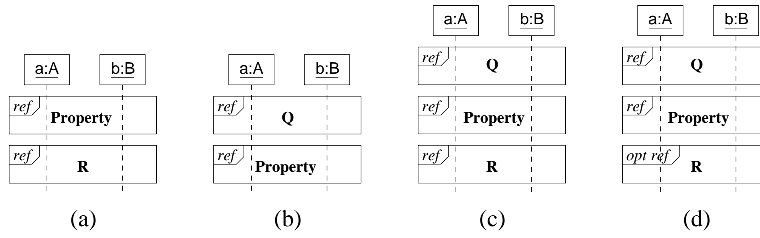


Fig. 10: Scope mapping for sequence diagrams: (a) **Before** R; (b) **After** Q; (c) **Between** Q and R; and (d) **After** Q **until** R.

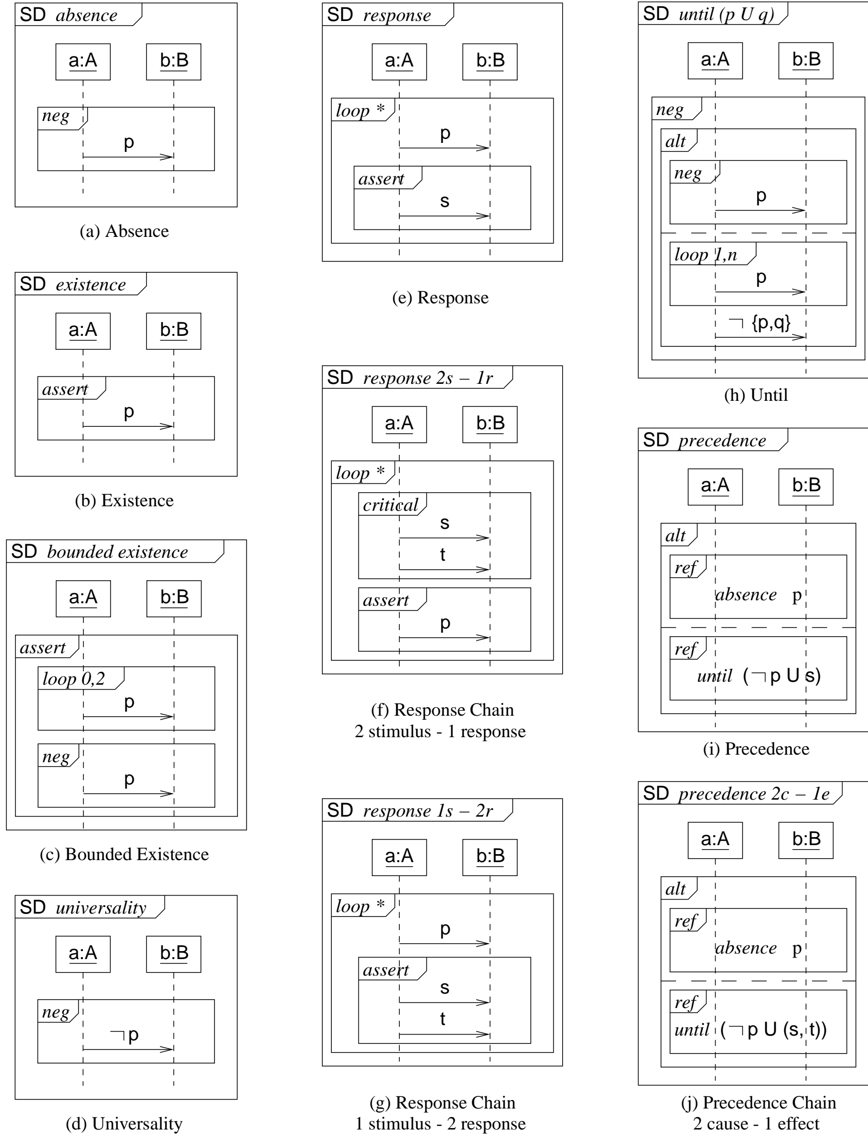


Fig. 11: Property pattern mapping for sequence diagrams

4.4 Specifying Properties of the Loan Application

The following examples show how property patterns can be used to specify example properties of the LA system given in Table 1. Properties P_1 and P_2 in that table correspond to simple safe and live SDs and are described in Figs 6 and 7, respectively. The rest of the properties in that table are discussed below.

Property P_3: We can express this property by using the absence pattern (see Fig 11(a)); P_3 holds if there are no traces where a loan is granted after receiving a `lnAtNO` message. A loan is considered granted if a `ceLn` or `psAn` message is received. See Fig. 12 (a) for the corresponding SD; the monitor is shown in Fig. 12 (e). The remaining automata in Fig. 12 show intermediate steps in the contruction of the monitor.

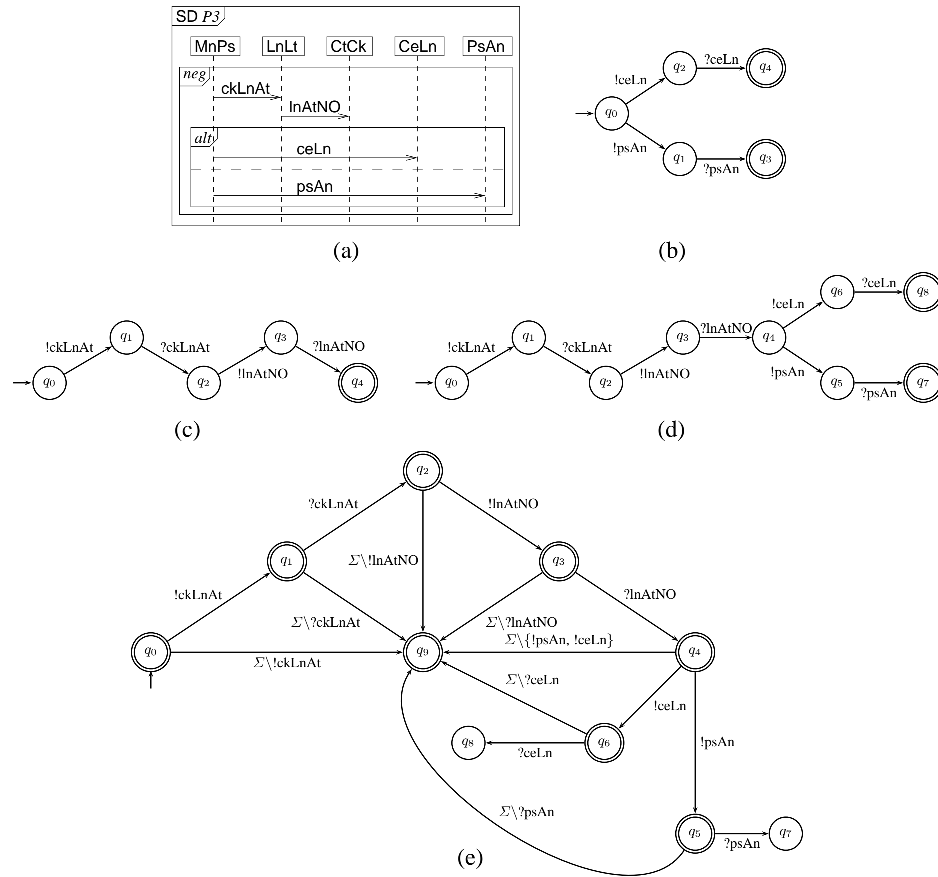


Fig. 12: P_3: Absence pattern. (a) SD describing the LA property P_3 and its corresponding NFAs: (b) NFA for *alt* operator; (c) NFA for the first two messages; (d) NFA for first two messages, followed by the *alt* operator; (e) the resulting monitor (after applying *negate*).

Property P_4: This property is a scoped version of P_3, i.e., P_4 is equivalent to the property P_3 scoped by the expression *After Q* where Q is "checking for a good credit

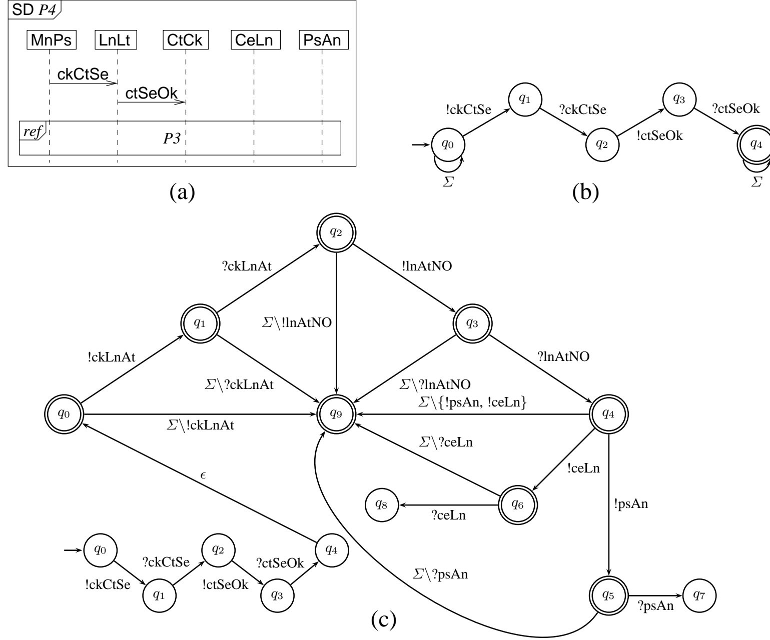


Fig. 13: P_4: Scoped absence pattern. (a) SD describing the LA property P_4 and its corresponding NFAs: (b) NFA for scope Q; (c) the resulting monitor, obtained by concatenating the NFAs for the scope and P_3.

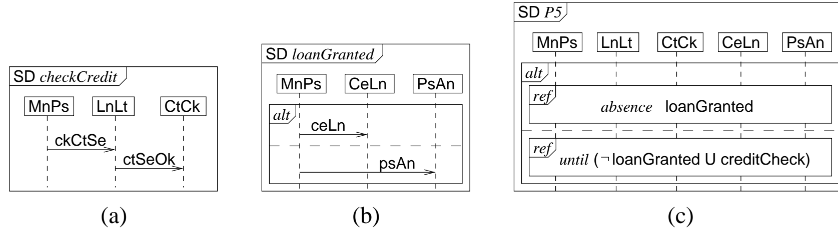


Fig. 14: P_5: Precedence pattern. (a) SD for *checkCredit*; (b) SD for *loanGranted*; (c) SD showing application of precedence pattern.

score". So we only need to monitor the property P_3 on traces where a good credit score is already detected. To do this, we introduce the scope delimiter Q before the property P_3, as seen in Fig 10(b). The SD corresponding to P_4 is shown in Fig 13(a) and consists of two parts: (1) scope Q and (2) property P_3, i.e., the fragment specified by a *ref* operator which should be replaced by the SD for P_3. The monitor for the whole property P_4 is shown in Fig. 13(c).

Property P_5: We can express property P_5 using the precedence pattern by indicating that the scenario *creditCheck* must precede the scenario *loanGranted*. Note that in this pattern, *creditCheck* is not optional and it must occur for the property to hold. Figs. 14(a) and (b) show the SDs for *creditCheck* and *loanGranted*, respectively. The SD for P_5 is shown in Fig. 14(c) which is an instantiation of the Precedence pattern in Fig 11(i) where p is replaced by *loanGranted*, and s by *creditCheck*. In this pattern, message p is replaced by a scenario, and hence, scenario negation is used instead of message complementation.

5 Tool Support and Experience

Tool Support. We have implemented our runtime monitoring framework within the IBM WebSphere® business integration products [21]. In what follows, we describe the architecture of our solution and its intended use. We also report on preliminary experience of using this framework to check correctness of web services. For implementation details, see [12].

Our solution uses the WebSphere Process Server (WPS) [22] and the WebSphere Integration Developer (WID) [23]. The former provides a BPEL-compliant process engine for executing BPEL processes and a built-in Service Component Architecture (SCA), which is a particular instantiation of SOA. The latter provides a development environment for building web service applications and a graphical package for creating UML Sequence Diagrams.

During and after application development, users can create UML SD specifications for their web service applications within the WID environment. If monitoring is enabled, our framework translates these diagrams into monitor automata using the techniques in Section 3. During the execution of the web service, interaction events from the WPS are sent to our framework using sockets. These events are immediately used to update the state of every active monitor automaton, until an error has been found or all partners terminate. This provides an *online* feedback mechanism through the SD editor to report violations.

Our patterns are available as editable UML sequence diagrams (`.dnx` files). Users must first add these files to the WID project of the application they wish to monitor. These patterns can now be modified to create actual system properties, using our Sequence Diagram editor.

Violations in our framework are either due to the occurrence of a negative trace (safety violation), or the absence of a positive trace (liveness violation). To report violations, we display the causes in the Sequence Diagram editor by highlighting the beginning of the negative trace for safety violations (see Fig 15 (a)), and the termination location for liveness violations (see Fig 15 (b)).

Experience. We applied our framework to the Loan Application system, with the goal of specifying and checking the properties mentioned in Table 1. On normal execution traces of this system, these properties should never fail, as this application implements the workflow shown in Fig 1(a). As it is a *sample* application, some details have been simplified. For example, the `CreditCheck` component generates random credit scores.

Testing the LA system, we realized that properties P_1, P_3, and P_4 *always* fail when taxpayer id is 1888 and loan amount is -1000. The faulty execution trace *FT* is as follows:

FT = `(MnPs, ckCtSe, LnLt),(LnLt, ctSeOK, CtCk),(MnPs, ckLnAt, LnLt),`
`(LnLt, lnAtNO, CtCk),(MnPs, ceLn, CeLn)`

where each triple $(Sender, m, Receiver)$ denotes that partner $Sender$ sends message m to partner $Receiver$. It can be seen that: (1) P_1 fails because the loan amount is not greater than zero (indicated by event `(LnLt, lnAtNO, CtCk)`); (2) P_3 is violated because a forbidden behavior – invalid loan is accepted – appears in *FT* (corresponds to the subtrace `(MnPs, ckLnAt, LnLt),(LnLt, lnAtNO, CtCk),(MnPs,`

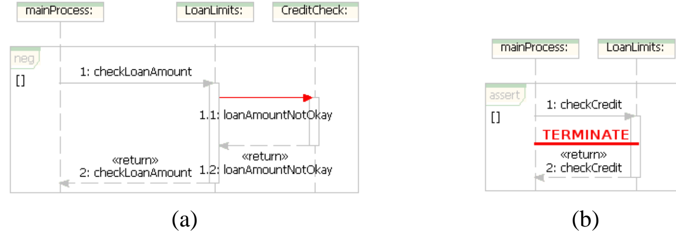


Fig. 15: Reporting errors: (a) event in a negative trace, (b) incomplete positive trace

`ceLn, CeLn)`); (3) P_4 fails for the same reason that P_3 fails because P_4 is a scoped version of P_3. Note that properties P_2 and P_5 in Table 1 are not violated by the above trace.

To get an idea of what caused this behavior, we examine the BPEL diagram in Fig. 1(a). In order to produce the last event in *FT*, the LA system must: obtain the taxpayer's credit score; check if the credit score is greater than 750 (`ScoreEvaluation`); check if the loan amount is greater than zero; and check if the loan amount is greater than $50 001 (`AutoApprovalTest`). The `ScoreEvaluation` should only occasionally be true, as the `CreditCheck` component generates random credit scores. In practice, running the LA system with taxpayer id 1888 as input *always* produces *FT*. The `AutoApprovalTest` will *always* return true, as the loan amount is -1000.

After examining the source code for the `CreditCheck` partner, we realized that the application was distributed with some hard-coded logic, presumably to facilitate testing purposes. An applicant with a taxpayer id that ends with "888" is always trusted and given a good credit score (instead of a random score). This, coupled with the fact that the application does not use the results of the input validation carried out on the loan amount, means that this loan is automatically approved every single time.

6 Conclusion

In this paper, we described our framework for runtime monitoring of web service conversations developed as part of an industrial-strength system. The framework is an aggregation of existing runtime verification techniques and is a continuation of [12]. It is non-intrusive, running in parallel with the monitored system and intercepting interaction events during run time. Thus, it does not require any code instrumentation, does not significantly affect the performance of the monitored system, and enables reasoning about partners expressed in different languages. Furthermore, the use of a subset of UML 2.0 SDs as a specification language ensures that the framework is usable by practitioners to specify a wide range of properties. By formalizing this subset using automata, we can check finite execution traces of web services against these properties. Liveness becomes finitary, where user-specified time limits or the process termination act as the stopping conditions.

We have successfully mapped all the Specification Property System patterns into our SD subset. The availability of customizable patterns should improve the usability of our specification language. More complex conversations can be checked, as it is easy to build properties through SD composition. Using SD references, our properties are also easier to read, since details can be hidden. Finally, we have created a library of such sequence diagram patterns and showed how patterns can be used to specify monitors which lead to discovery of bugs in real webservice applications.

Related Work. Like other partial-order scenario-based formalisms such as MSCs [24] and LSCs [25], UML 2.0 Sequence Diagrams are enjoying an increasing usage as specification languages. [14] proposes a Property Sequence Chart (PSC) language, which is an extended notation of a subset of UML 2.0 SDs. PSC enables expressing safety and liveness properties by assigning attributes *fail* and *required* to messages. This is equivalent to applying operators *negate* and *assert* to individual SD message, respectively. The semantics of PSC is given using Büchi Automata, designed to operate on infinite execution traces. Since we consider only finite executions of web services, automata over finite words are sufficient and significantly easier to implement. In [26], web services are verified using a Petri Net model generated from a DAML-S description of a service.

Future Work. While the initial experience using the framework has been positive, we need to address a number of issues before it becomes fully usable. The first set of issues deals with increasing the range of properties that can be specified and monitored. In the examples presented here, all objects were unique, whereas in practice, users may be interested in verifying interactions between multiple processes of the same type. For example, a user with a good credit score may concurrently apply for two loans, each for less than $50 001, to bypass the manual approval required for a loan for the total amount. In this case, two bank branches may want to communicate to avoid this kind of situation. We feel that the problem can be easily solved by encoding process IDs into the specification, the automata transition relation, and interaction events.

We also plan to begin investigation of techniques to help locate cause of errors from seeing results of successful and unsuccessful runs of the system. For example, given a monitor violation, we would like to produce similar conversations that do not cause a violation, so as to help pinpoint cause of the violation (as the place signaled with the violation is not necessarily the cause). We will experiment with the techniques in [27, 28] for this task.

On a side note, our work so far has been built on a basis that all partners operate within the same process server and thus a centralized monitor is a viable option. In practice, most web services are distributed, requiring a distributed monitoring framework. We plan to investigate techniques used in the DESERT project [29] to turn a centralized monitor into a set of distributed ones, running in different process servers.

Acknowledgements and Trademarks

We thank Yuan Gan and Jonathan Amir for implementing several parts of the monitoring framework, and Simon Moser and Axel Martens for generating many useful discussions. This work is financially supported by the IBM Toronto Centre for Advanced Studies, Ontario Graduate Scholarship and NSERC.

IBM and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

References

1. Dwyer, M., Avrunin, G., Corbett, J.: "Patterns in Property Specifications for Finite-State Verification". In: Proceedings of 21st International Conference on Software Engineering (ICSE'99). (May 1999) 411–420
2. IBM: Business Process Execution Language for Web Services. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
3. Fu, X., Bultan, T., Su, J.: "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services.". In: In Proceedings of the Eighth International Conference on Implementation and Application of Automata (CIAA 2003), Santa Barbara, California (July 2003) 188–200
4. Fu, X., Bultan, T., Su, J.: "Analysis of Interacting BPEL Web Services.". In: Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), New York, NY (May 2004) 621–630
5. Kazhamiakin, R., Pistore, M.: "A Parametric Communication Model for the Verification of BPEL4WS Compositions". In: EPEW/WS-FM. (2005) 318–332
6. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: "Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step". In: EPEW/WS-FM. (2005) 257–271
7. Foster, H., Uchitel, S., Magee, J., Kramer, J.: "Model-based Verification of Web Service Compositions". In: Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003), IEEE Computer Society (2003) 152–163
8. Ghafari, N., Gurfinkel, A., Klarlund, N., Trefler, R.: "Algorithmic Analysis of Piecewise FIFO Systems". In: Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07). LNCS, Austin, Texas (November 2007) 45–52
9. Baresi, L., Ghezzi, C., Guinea, S.: "Smart Monitors for Composed Services". In: ICSOC'04. (2004) 193–202
10. Robinson, W.N.: "Monitoring Web Service Requirements". In: Proceedings of RE'03. (2003) 65–74
11. Mahbub, K., Spanoudakis, G.: "Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience". In: Proceedings of ICWS'05. (2005) 257–265
12. Gan, Y., Chechik, M., Nejati, S., Bennett, J., O'Farrell, B., Waterhouse, J.: "Runtime Monitoring of Web Service Conversations". In: Proceedings of CASCON'07. (November 2007)
13. Object Management Group (OMG): Unified Modeling Language (UML 2.0). http://www.uml.org/
14. Autili, M., Inverardi, P., Pelliccione, P.: "A Scenario Based Notation for Specifying Temporal Properties". In: Proceedings of SCESM, ICSE'06 Workshop. (2006)
15. Alur, R., Yannakakis, M.: "Model Checking of Message Sequence Charts". In: Proceedings of CONCUR'99. (1999) 114–129
16. Grosu, R., Smolka, S.A.: "Safety-Liveness Semantics for UML 2.0 Sequence Diagrams". In: ACSD'05. (2005) 6–14
17. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (1979)
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: "Property Specification Patterns for Finite-state Verification". In: Proceedings of 2nd Workshop on Formal Methods in Software Practice. (March 1998)
19. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
20. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern Based Property Specification and Verification for Service Composition. In: Proceedings of 7th International Conference on Web Information Systems Engineering (WISE 2006). (2006) 156–168

21. IBM: WebSphere Business Integration Software. http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint
22. IBM: WebSphere Process Server. http://www-306.ibm.com/software/integration/wps/
23. IBM: WebSphere Integration Developer. http://www-306.ibm.com/software/integration/wid/
24. ITU-TS: "ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96)". Technical report, ITU-TS, Geneva (1996)
25. Damm, W., Harel, D.: "LSCs: Breathing Life into Message Sequence Charts.". Formal Methods in System Design **19**(1) (2001) 45–80
26. Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: WWW '02: Proceedings of the 11th international conference on World Wide Web, New York, NY, USA, ACM (2002) 77–88
27. Zeller, A.: Isolating cause-effect chains from computer programs. SIGSOFT Softw. Eng. Notes **27**(6) (2002) 1–10
28. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3) (2006) 229–247
29. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: "Synthesis of Correct and Distributed Adaptors for Component-Based Systems: an Automatic Approach". In: Proceedings of ASE'05. (2005) 405–409

A Other Property Patterns

$k-$**Bounded Existence:** message $\mathtt{p}$ can occur at most k times in a given scope. We can check the existence of at most k messages using the *loop* operator. After the loop, we need to check that $\mathtt{p}$ does not occur, which corresponds to the absence pattern (see Fig. 11 (c)).

Universality: only a sequence $\mathtt{p}^*$ of messages can occur in a given scope. This is equivalent to checking for the absence of complement messages (see Fig. 11 (d)).

Response: message $\mathtt{p}$ (stimulus) must be followed by message $\mathtt{s}$ (response), in a given scope. A response can occur without stimuli, so the stimulus is represented using a regular message, whereas the response is mandatory. The existence of stimulus/response pairs are checked in an infinite *loop*, as there can be many stimulus/response pairs in one execution trace (see Fig. 11 (e)).

Response Chain: a sequence $\mathtt{p}_1, \ldots, \mathtt{p}_n$ of messages must be followed by the sequence $\mathtt{q}_1, \ldots, \mathtt{q}_m$ of messages, in a given scope. We show two examples of this pattern: $\mathtt{p}$ responds to $\mathtt{s}, \mathtt{t}$ (see Fig. 11 (f)), and $\mathtt{s}, \mathtt{t}$ responds to $\mathtt{p}$ (see Fig. 11 (g)). Response chain patterns have the same basic form of the response pattern.

- $\mathtt{p}$ responds to $\mathtt{s}, \mathtt{t}$: 2 stimulus – 1 response. The *critical* operator is used to enclose the message sequence $\mathtt{s}, \mathtt{t}$, to ensure atomicity of this sequence. An *assert* cannot be used since the stimulus sequence is optional.
- $\mathtt{s}, \mathtt{t}$ responds to $\mathtt{p}$: 1 stimulus – 2 response. The message sequence now occurs within the *assert* operator, so an additional *critical* operator would be superfluous).

Precedence Chain: a sequence $\mathtt{p}_1, \ldots, \mathtt{p}_n$ of messages must precede the sequence $\mathtt{q}_1, \ldots, \mathtt{q}_m$ of messages, in a given scope. We show an example of this pattern, 2 cause – 1 effect, $\mathtt{p}$ is preceded by $\mathtt{s}, \mathtt{t}$ (see Fig. 11 (j)). This pattern is mapped using the *absence* and *until* patterns, just like in the *precedence* pattern. The implicit *negate* operators in the *absence* and *until* patterns handle the message sequences, so there is no need to add *critical* operators.

B Framework

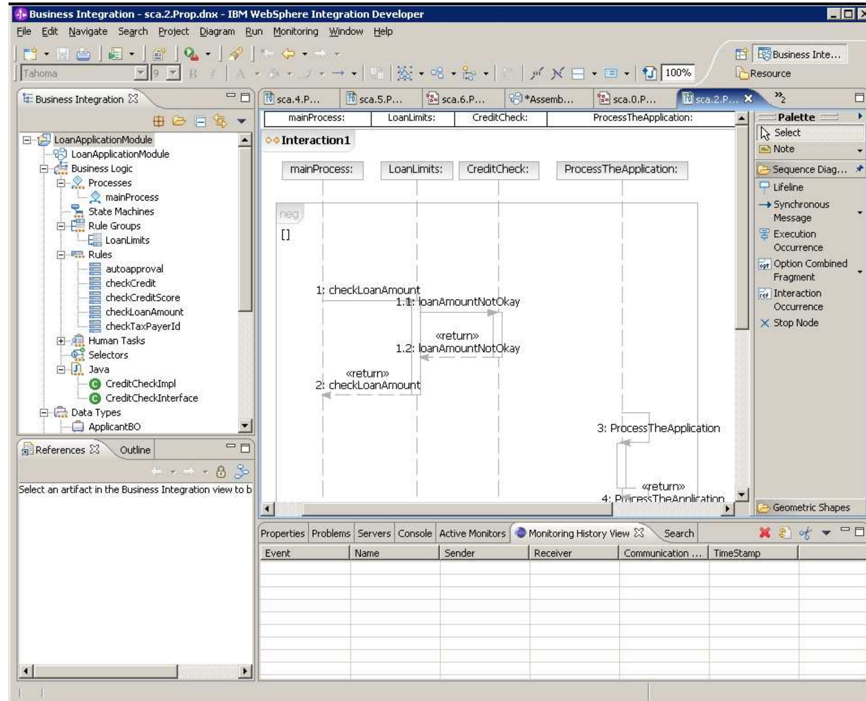


Fig. 16: Screenshot of the Framework's User Interface.