Runtime Monitoring of Web Service Conversations

Jocelyn Simmonds, Yuan Gan, Marsha Chechik, *Member*, *IEEE Computer Society*, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse

Abstract—For a system of distributed processes, correctness can be ensured by (statically) checking whether their composition satisfies 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 as a property specification language and show that it is sufficiently expressive for capturing safety and liveness properties. By transforming these diagrams to automata, we enable conformance checking of finite execution traces against the specification. We show how our language can be used to specify the Specification Property System (SPS) [1]. We describe an implementation of our approach as part of an industrial system. Finally, we discuss our experience of specifying and monitoring a number of properties from three existing applications.

Index Terms—Nondeterministic finite automata, runtime monitoring, sequence diagrams, temporal logic patterns, Web service conversations.

1 Introduction

Recent years have seen an emergence of the field of Web services, which use Service-Oriented Architectures (SOAs) to dynamically discover and bind to services in order to increase the flexibility of business interactions. Each service consists of *components* and can discover other components using published interfaces. An SOA component can be written in a traditional compiled language such as Java^{TM}, or in an XML-centric language such as BPEL [2]. An SOA *module* is made up of multiple SOA components, which are commonly referred to as Web services.

Since each Web service is a relatively simple process, analysis can concentrate on the message exchange between partners—their *conversations*. For a classical system of distributed processes, correctness can be ensured by statically checking their composition against properties of interest. The same approach has been taken by several researchers in the context of Web services as well, e.g., [3], [4], [5], [6], [7]. While static analysis is very appealing—errors are discovered ahead of time and without the need to exercise the system, this approach has several major limitations as follows:

- Web services typically communicate via infinite-length channels, so the problem is decidable only under certain conditions [8].
- Web applications usually interact with Web services developed by partners. Partners are only required to make Web service interfaces public, not the code.
- Realistic Web services exchange many types of messages: some synchronous, some asynchronous, and some with acknowledgments and priorities.
- Web services are typically heterogeneous, i.e., each component can be implemented in a different programming language.

Instead, we concentrate on the dynamic analysis via *runtime monitoring*, which tries to ensure the quality of an application through the analysis of runtime events. These events can be analyzed *online*—during the execution of the application, or *offline*—after the execution has been terminated. The latter can be used to express free-form queries over all generated events. However, since these queries are not necessarily known a priori, the runtime data collected might not be sufficient to answer the relevant questions; or, on the other extreme, the amount of data collected may become excessive and hard to manage, leading to intractable analysis. Online techniques, on the other hand, monitor predefined properties, collecting just those events that are related to these properties. While expressing properties beforehand may be nontrivial, the collected data are guaranteed to be both small and sufficient to check these properties; they also serve as an additional, and very valuable, *documentation* of the desired behavior of the system. Monitoring as the system runs also provides a chance of recovery once a problem has been detected, e.g., by terminating execution or trying to return to a stable state. For these reasons, we use online monitoring techniques in this paper.

Our goal is, thus, to create an industrial-strength (in partnership between the University of Toronto and the IBM Toronto Lab) online monitoring framework that is nonintrusive, supports the dynamic discovery of Web services,

- *J. Simmonds and M. Chechik are with the Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, ON, M5S 3G4, Canada. E-mail: {jsimmond, chechik}@cs.toronto.edu.*
- *Y. Gan, B. O'Farrell, E. Litani, and J. Waterhouse are with the IBM Toronto Lab, 8200 Warden Ave, Markham, ON, L6G 1C7, Canada. E-mail: {ygan, billo, elitani, juliew}@ca.ibm.com.*
- *S. Nejati is with the Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway. E-mail: shiva@simula.no.*

Manuscript received 26 Sept. 2008; revised 17 Dec. 2008; accepted 18 June 2009; published online 29 June 2009.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-09-0083.
Digital Object Identifier no. 10.1109/TSC.2009.16.

deals with synchronous and asynchronous communication and partners implemented in different languages, allows for specifying and efficient monitoring of a variety of temporal behavior, permits recovery strategies (this is not part of the current paper), and is usable by practitioners.

We also aim to create an industrial-strength language for specifying temporal behavior that captures the distributed, interactive, and message-driven nature of business processes. Our language should enable specifying a variety of properties and be amenable to efficient runtime monitoring, allowing the analysis of orchestrations involving multiple partners, from the point of view of the orchestrating service. We believe that such a language should have the following characteristics:

1. its notation should be *visual*;
2. it should allow the specification of desired temporal behavior, via sequences of events;
3. it should have an explicit emphasis on components and enable dealing with different types of message exchange; and
4. it should be able to specify positive and negative scenarios of interaction as well as global properties. These characteristics are necessary for the resulting language to be usable by practitioners.

Having considered a few behavioral graphical languages, such as GIL [9], Time Line Editor [10], Message Sequence Charts (MSCs) [11], and Live Sequence Charts (LSCs) [12], we have chosen UML 2.0 Sequence Diagrams (SDs) [13] as the basis for our specification language. SDs, used to capture interactions in the form of message passing between objects, have been widely adopted by industry as a suitable language for describing and documenting scenario-based requirement specifications, with additional constraints expressed using the Object Constraint Language (OCL) [14]. The other UML 2.0 diagrams did not meet our requirements for a specification language: no support for multiple parties (state machines); only allow the specification of simple sequences of events (communication and interaction overview diagrams); too low-level (activity diagrams); cannot be used to specify behavior (class diagrams).

SDs are a feature-rich language without a formal semantics. In this paper, we identify a subset of SDs that is sufficiently expressive for capturing *safety* and *liveness* properties. While liveness properties are not monitorable in general, they can be effectively checked for Web services with finitely terminating behaviors. Specifically, we aim to generate three types of monitors: accepting individual (existential) negative behaviors that correspond to a violation of safety properties; their dual, accepting universal positive behavior that corresponds to finite liveness (once an event occurs, the rest of the events must occur before termination); as well as individual positive behaviors that can easily be specified in SDs. For the latter type, we do not look for violations (if a given trace does not correspond to a desired behavior, perhaps others will), but do report when we were able to observe their satisfaction.

To enable monitoring, we formalize our subset of SDs using finite-state automata. Similar approaches to formalizing sequence diagram variants have been previously proposed by other researchers, e.g., [15], [16], [17]. Since automata and logic are intimately related, an automata-based characterization allows us to investigate connections between SDs and temporal logics, and translate SDs to automata to enable conformance checking of finite execution traces against their specifications expressed in SDs. We then show that this language is 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.

1.1 A Motivating Example

Consider, for example, a Web-based Loan Application (LA) system, distributed as a sample application with the IBM® WebSphere® Integration Developer v6.0.2. Users enter loan application information (name, taxpayer id, and loan amount) through a Web page, and are eventually informed of the status of their applications. The LA workflow first checks if the user's credit score is valid, and will decline their loan request if the user has a bad credit score, i.e., less than 750. A credit score is considered valid if it is between 300 and 850. 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. 1a, which is described as a BPEL specification, shows high-level steps that are executed in a loan application system, and Fig. 1b 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 `ProcessTheApplication`). Specifically, the `CheckCredit` activity in Fig. 1a invokes the `CreditCheck` partner in Fig. 1b, and the conditional activities `ScoreEvaluation` and `AutoApprovalTest` invoke the `LoanLimit` partner. The partners in Fig. 1b 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 `CompleteTheLoan`, `ProcessApplication`, and `FollowUp` follow the application results `Approved`, `ManualApproval`, and `Declined`, 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. Table 1 shows some properties of the LA system that can be expressed using our SD subset. For example, P_1 and P_2 are possible safety and liveness properties, respectively.

1.2 Organization of the Paper

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 Web service conversations in Section 2. We describe the semantics of our chosen subset

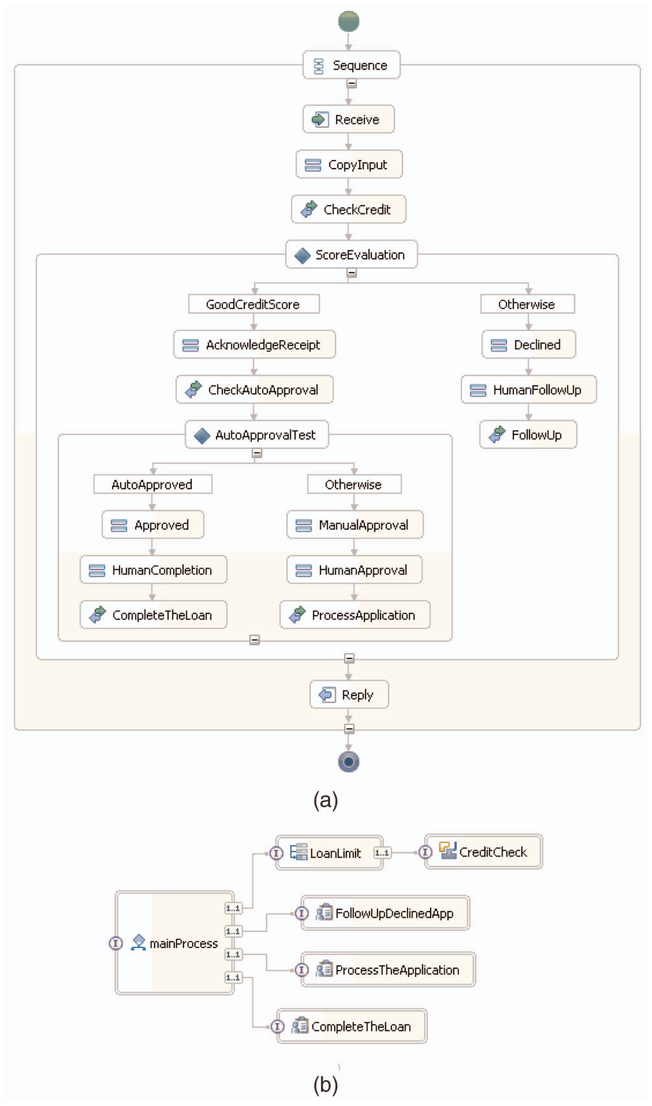

Fig. 1. The LA system: (a) workflow describing the high-level steps of the LA system; (b) an assembly diagram describing how the main process of the LA system interacts with its partners.

of SDs and show how to translate it into automata for runtime monitoring 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 in Section 5 and report on the experience using this framework for three existing Web service systems in Section 6. After comparing our approach with related work in Section 7, we conclude the paper in Section 8 with a summary and an outline of the future research directions.

2 A Language for Specifying Conversations

We choose a subset of UML 2.0 Sequence Diagrams as the language for specifying Web service conversations. This subset satisfies the requirements set forth in the previous section. We formalize this subset in Section 3 and discuss its expressive power in Section 4.

Sequence Diagrams are a popular formalism for modeling behavioral scenarios by describing sequences of messages communicated between different objects over time. An example Sequence Diagram describing a scenario of the LA system is shown in Fig. 2a. Sequence Diagrams have two dimensions: vertical, which represents time, and horizontal, which represents 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 (solid arrowhead) or asynchronously (open arrowhead). We refer to Sequence Diagrams with these features as *basic*. Basic Sequence Diagrams can be augmented by a number of operators to capture more sophisticated scenarios. We use the operators described below in our property specification language and refer to our language as *SD*.

- **Compositional operators:** Operators *parallel (par)* and *alternatives (alt)* are used to compute the intersection and union of two SDs, respectively. The operator *loop* is used for repeating the scenario described by an SD multiple times, and *opt*—for denoting an optional scenario, equivalent to *alt* with only one argument.
- **Alphabet changing operators:** Operators *consider* and *ignore* are used for modifying the communicating alphabet of SDs.
- **Critical operator:** The *critical* operator is used to ensure atomicity of the enclosed sequence.
- **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 reference, 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 an individual message or a set of messages appearing on the same arrow. The *negate* operator is unsuitable for complementing sets because it captures negative sequences of messages rather than set complementation. Instead, we use the *message complementation*

TABLE 1
Several Properties of the LA System

P_1	The credit score should always be valid, i.e., between 300 and 850.
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.

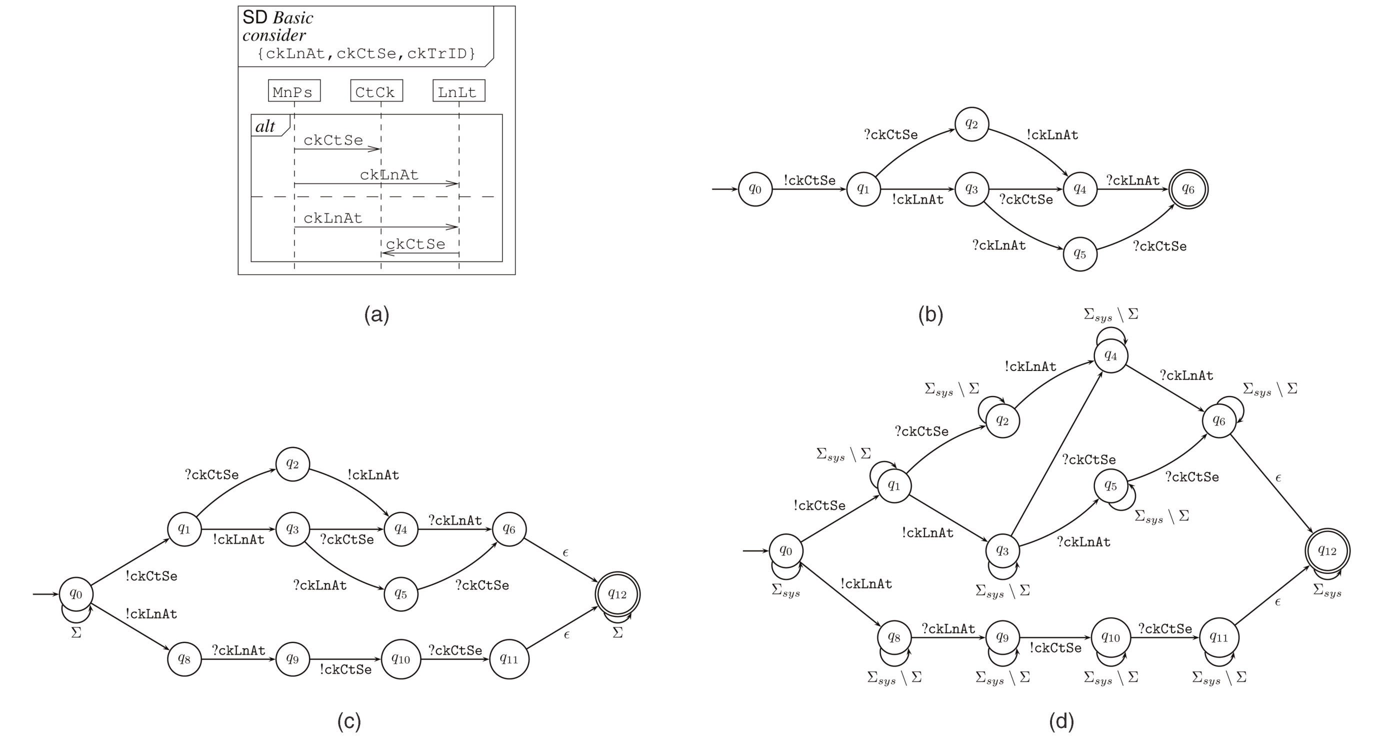

Fig. 2. (a) An SD describing a scenario of the LA example; (b) the NFA corresponding to the first argument of the *alt* operator in Fig. 2a; (c) the NFA corresponding to the complete scenario in Fig. 2a; and (d) the resulting runtime monitor.

operator, originally introduced in the Property Sequence Charts (PSCs) language [18]. We denote the complement of a message m by $\neg m$ and define it as the set of all messages that is potentially exchanged between objects of the system except for m.

An example of sequence diagram describing a scenario of the LA system is shown in Fig. 2a. 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 `CheckCredit` and `LoanLimit`, respectively. The diagram in Fig. 2a shows two alternative scenarios: First, `MnPs` sends a check credit score request, i.e., `ckCtSe`, to `CtCk`, and then a check loan amount request, i.e., `ckLnAt`, to `LnLt`. Second, `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`.

Basic Sequence Diagrams, denoted by *BasicSDs*, are the building blocks of our language. The *critical*, alphabet changing, interaction use, *assert*, and compositional operators, except for *par*, can be intermixed and applied any number of times to BasicSDs. The use of *negate* and *par* operators, however, is restricted to sequence diagrams that do not use an *assert* operator. We discuss this assumption and the rationale behind it in Section 3.6.2 and show in Section 4 that even with this restriction, the resulting language remains very expressive.

The grammar for our language, SD, is given in Fig. 3, where $BasicSD$, *par*, *alt*, *loop*, *critical*, *opt*, *negate*, *assert*, *consider*, *ignore*, and *ref* are terminal symbols, and E is a set of SD messages. Since operators *consider* and *ignore* change the communicating alphabet of SDs, they take a set E of messages as an input argument.

In what follows, we denote by SD the set of Sequence Diagrams generated by the grammar in Fig. 3.

3 Formalizing Sequence Diagrams

In this section, we provide a formal description of semantics of Basic SDs as well as the operators described in Section 2 by adopting the automata-theoretic approach of Alur and Yannakakis [15].

3.1 Nondeterministic Finite Automata

Let Σ be an alphabet. We define a *trace* σ over Σ to be a finite sequence $\sigma_0\sigma_1\ldots\sigma_n$, where $\forall i \cdot 0 \leq i \leq n, \sigma_i \in \Sigma$. We denote by Σ^* the set of all finite traces over Σ.

$$
\begin{array}{rcl}
\texttt{SD} & ::= & \mathit{BasicSD} \mid \texttt{unaryOp SD} \mid \texttt{SD}\ \mathit{alt}\ \texttt{SD} \mid \texttt{NotAssertedSD}\ \mathit{par}\ \texttt{NotAssertedSD} \mid \\
 & & \mathit{assert}\ \texttt{SD} \mid \mathit{negate}\ \texttt{NotAssertedSD} \\
\texttt{NotAssertedSD} & ::= & \mathit{BasicSD} \mid \texttt{unaryOp NotAssertedSD} \mid \texttt{NotAssertedSD}\ \mathit{alt}\ \texttt{NotAssertedSD} \mid \\
 & & \mathit{negate}\ \texttt{NotAssertedSD} \mid \texttt{NotAssertedSD}\ \mathit{par}\ \texttt{NotAssertedSD} \\
\texttt{unaryOp} & ::= & \mathit{consider}_E \mid \mathit{ignore}_E \mid \mathit{loop} \mid \mathit{critical} \mid \mathit{opt} \mid \mathit{ref}
\end{array}
$$

Fig. 3. Grammar of the SD language.

Definition 1 (Projection "$\downarrow$"). *Let $\Sigma' \subseteq \Sigma$ be an alphabet and $\sigma = \sigma_0 \ldots \sigma_n$ be a trace over Σ. The* projection *of σ to Σ', denoted $\sigma \downarrow_{\Sigma'}$, is obtained by replacing every σ_i $(0 \le i \le n)$ by the silent symbol ϵ iff $\sigma_i \notin \Sigma'$.*

Definition 2 (NFA [19]). *A* Nondeterministic Finite Automaton *(NFA) A is a tuple $(\Sigma, Q, \delta, Q_0, F)$, where Σ is a set of input alphabet, Q is a finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states.*

A trace $\sigma = \sigma_0\sigma_1 \ldots \sigma_n$ is accepted by A iff there is a sequence $q_0 q_1 \ldots q_{n+1}$ of states s.t. $q_0 \in Q_0, q_{n+1} \in F$, and for every $0 \le i \le n, (q_i, \sigma_i, q_{i+1}) \in \delta$. The language of $A, L(A)$, is the set of all traces accepted by A.

An example of NFA over the alphabet $\{!\texttt{ckCtSe}, ?\texttt{ckCtSe}, !\texttt{ckLnAt}, ?\texttt{ckLnAt}\}$ is shown in Fig. 2b. In cases where states do not have outgoing transitions for some symbols in Σ, e.g., state q_1 on $?\texttt{ckLnAt}$ in Fig. 2b, it is assumed that this symbol causes a transition to a (non-accepting) dead-end state, which is usually not shown.

Let (q, a, q') be a transition in an NFA A. We often refer to a as the label of the transition from q to q'. For an NFA A with ϵ transitions, let $L(A)$ be the set of traces of A with the occurrences of ϵ removed.

States in NFAs may have several outgoing transitions on the same input symbol, or may have transitions labeled ϵ, indicating a *silent* move. *Deterministic* finite automata (DFAs) are NFAs, where each state has at most one outgoing transition on each nonsilent symbol. Every NFA can be converted into a DFA using the subset construction algorithm [19].

3.2 Basic SDs

We define Basic SDs as follows:

Definition 3 (Basic SDs [15]). *A Basic SD S is a tuple $(\mathcal{I}, E, f, \mathcal{O})$, where*

- *$\mathcal{I}$ is a finite set of objects.*
- *E is a finite set of event occurrences that is partitioned into send events, denoted by $!E$, and receive events, denoted by $?E$. The set of events sent and received by an object $i \in \mathcal{I}$ is denoted by E_i.*
- *f: $!E \rightarrow\ ?E$ is a bijective mapping that associates each send event e with a unique receive event $f(e)$, and each receive event e' with a unique send event $f^{-1}(e')$.*
- *$\mathcal{O}$ is a set of total order relations $<_i$ defined over the events E_i for every object i. It corresponds to the order in which the events are physically displayed along the lifeline of an object i.*

Definition 4 (Partial order [15]). *Let $S = (\mathcal{I}, E, f, \mathcal{O})$ be a Basic SD. We define a partial order relation $<$ over E as follows:*

$$< = \ [(\cup_{i \in \mathcal{I}} <_i) \ \cup \ (\{(s, f(s)) \mid s \in !E\})].^*$$

The scenario in the first argument of the *alt* operator, shown in Fig. 2a, is a Basic SD. Here, the set of objects is

$$\mathcal{I} = \{\texttt{MnPs}, \texttt{CtCk}, \texttt{LnLt}\},$$

the set of events is

$$E = \{!\texttt{ckCtSe}, ?\texttt{ckCtSe}, !\texttt{ckLnAt}, ?\texttt{ckLnAt}\},$$

the total order $<_{\text{MnPs}}$ for the object $\texttt{MnPs}$ is

$$!\texttt{ckCtSe} <_{\text{MnPs}} !\texttt{ckLnAt},$$

and the partial order $<$ associated with the entire diagram is

$$!\texttt{ckCtSe} < !\texttt{ckLnAt},$$
$$!\texttt{ckCtSe} < ?\texttt{ckCtSe},$$
$$!\texttt{ckLnAt} < ?\texttt{ckLnAt}.$$

This partial order assumes that messages are communicated asynchronously. Partial order for synchronous communication is a subset of the above because of synchronization. In the rest of this paper, we assume that messages are passed asynchronously. Also, without loss of generality, we assume that all event labels are unique.

We define the semantics of Basic SDs by translating them into their equivalent NFAs. Intuitively, an NFA A_S is equivalent to a Basic SD S iff A_S accepts exactly the set of traces that can be generated by S, i.e., those traces that respect the partial order of S. Therefore, translation of S to A_S reduces to the translation of the underlying partial order of S to A_S. The algorithm for translating partial orders to NFAs, proposed in [15], is as follows: Given a partial order $<$ over E, let *cut* c be a subset E that is closed with respect to $<$, i.e., if $e \in c$ and $e' < e$, then $e' \in c$. Since all the events of a single process are linearly ordered, a cut can be specified by a tuple that gives the maximal event of each process. The set of all possible cuts associated with the partial order of a Basic SD generates the state space of its corresponding NFA. The empty cut is the initial state and cuts with all the events is the final state. There is a transition labeled e from cut c to cut d, if the cut d equals the cut c plus the single event e.

Theorem 1. *A Basic SD $S = (\mathcal{I}, E, M, \mathcal{O})$ is semantically equivalent to an NFA $A_S = (\Sigma, Q, \delta, Q_0, F)$, where Σ is equal to E, Q is the set of all cuts, Q_0 is the empty cut, F is the maximal cut including all of the events, and δ allows a transition from a cut d to a cut c on an event $e \in E$ iff $d = c \cup \{e\}$.*

The above theorem follows from [15].

Since both the empty and maximal cuts are unique, Q_0 and F consist of only one state each. The set of cuts obtained by unwinding the underlying partial order in the Basic SD in Fig. 2a is

$$\begin{gathered}\{\langle\rangle, \langle !\texttt{ckCtSe}\rangle, \langle !\texttt{ckCtSe}, ?\texttt{ckCtSe}\rangle,\\ \langle !\texttt{ckCtSe}, !\texttt{ckLnAt}\rangle, \langle !\texttt{ckCtSe}, ?\texttt{ckCtSe}, !\texttt{ckLnAt}\rangle,\\ \langle !\texttt{ckCtSe}, !\texttt{ckLnAt}, ?\texttt{ckLnAt}\rangle, \langle !\texttt{ckCtSe}, !\texttt{ckLnAt}, ?\texttt{ckCtSe}\rangle,\\ \langle !\texttt{ckCtSe}, ?\texttt{ckCtSe}, !\texttt{ckLnAt}, ?\texttt{ckLnAt}\rangle\}.\end{gathered}$$

Note that the number of states of the corresponding automaton in Fig. 2b is less than the number of the above cuts because we reduced the states with the identical outgoing transitions to a single state.

3.3 Compositional Operators

The semantics of the compositional operators can be given in terms of the standard operations defined on NFAs (e.g., see [19]). In particular,

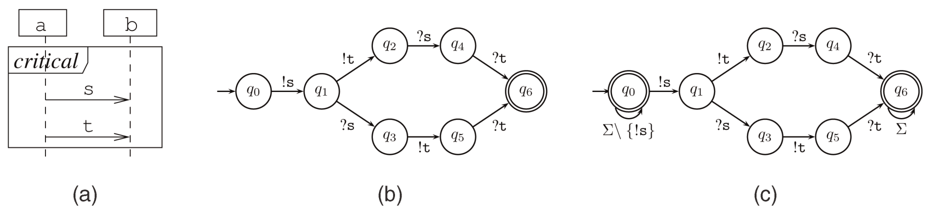

Fig. 4. (a) A basic SD enclosed by a *critical* operator and its corresponding NFAs: (b) before applying *critical*; (c) after applying *critical*.

- *par* corresponds to the parallel composition operator or the intersection operator over NFA;
- *alt* corresponds to the union operator;
- *loop* corresponds to the Kleene star operator.

The theorem below, which follows from Theorem 1 and [19], shows that the set of NFAs associated with SDs is closed under the compositional operators.

Theorem 2. *Let S, S_1, and S_2 be SDs, and let $S = S_1 op S_2$, where op is a compositional operator. Then, $A_S = A_{S_1} op A_{S_2}$.*

For example, the automaton in Fig. 2c corresponds to the sequence diagram in Fig. 2a. As shown in the figure, the automaton is obtained by computing the union of the two Basic SDs corresponding to the two alternative scenarios of the SD in Fig. 2a. We have also added a self-loop to the initial state of the automaton in Fig. 2c labeled with the underlying alphabet of the SD in Fig. 2a. The self-loop allows the automaton to guess when the scenario specified by the SD begins.

3.4 Alphabet Changing Operators

Operators *consider* and *ignore* are used to change the set of communicating alphabet 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. Formally, let S and S' be SDs, E be a set of events, and $A_S = (\Sigma, Q, \delta, \{q_0\}, F)$ be the automaton associated with S. For $S' = consider_E S$, $A_{S'} = (\Sigma \cup E, Q, \delta, \{q_0\}, F)$, and for $S' = ignore_E S$, $A_{S'} = (\Sigma \setminus E, Q, \delta', \{q_0\}, F)$, where

$$\delta' = \big(\delta \cap (Q \times (\Sigma \setminus E) \times Q)\big) \cup \{(q, \epsilon, q') \mid \exists \sigma \in E \cdot (q, \sigma, q') \in \delta\}.$$

It is easy to see that the set of NFAs associated with SDs is closed under the operators *consider* and *ignore* as well.

Recall that any missing transition at a state leads to an error state. Increasing the input alphabet Σ of A_S without changing the transition relation δ means that more execution traces end up in the error state, while shrinking the input alphabet without changing the transition relation means that more execution traces are accepted. For example, the *consider* operator in Fig. 2a extends the underlying alphabet Σ of the automaton in Fig. 2c from $\{\texttt{!ckCtSe}, \texttt{?ckCtSe}, \texttt{!ckLnAt}, \texttt{?ckLnAt}\}$ to $\{\texttt{!ckCtSe}, \texttt{?ckCtSe}, \texttt{!ckLnAt}, \texttt{?ckLnAt}, \texttt{!ckTrID}, \texttt{?ckTrID}\}$.

3.5 Critical Operator

A critical region in a sequence diagram can be specified using the *critical* operator. A critical region means that the scenarios of the region cannot be interleaved by other messages, and thus, should be treated atomically. We formalize the semantics of this operator as follows: If the first message of the critical region is observed, then the rest of the behavior must be observed as well, without seeing any intermediate message.

Let S be an SD enclosed within a *critical* operator and A_S be the automaton for S. The automaton for *critical* S is obtained by adding a self-loop to every initial state of A_S labeled by $\Sigma\backslash\{e \mid \exists q_0 \in Q_0 \cdot q_0 \text{ has an outgoing transition on} e\}$. This self-loop transition at the initial state allows the automaton to wait for a satisfying run to begin. The initial state also becomes final.

Definition 5. *Let $A_S = (\Sigma, Q, \delta, \{q_0\}, F)$ be an NFA associated with an SD S and S^{crit} be an SD obtained by enclosing S with a* critical *operator. The automaton corresponding to S^{crit} is $A_S^{crit} = (\Sigma, Q, \delta', \{q_0\}, F \cup \{q_0\})$, where*

$$\delta' = \delta \cup \{(q_0, e, q_0) \mid e \in \Sigma \wedge \nexists q \in Q \cdot q \neq q_0 \wedge (q_0, e, q) \in \delta\}.$$

For a sequence enclosed by a critical operator, once the first symbol of the sequence has been seen, the entire sequence should be seen as well. For this reason, the self-loop at the initial state of an automaton corresponding to a critical region is labeled by Σ minus the initial symbols of the expected sequences. For example, Fig. 4a shows a sequence diagram with a critical operator, and Fig. 4c—its corresponding automaton. Similar to the automaton in Fig. 2c, we have added a self-loop to the initial state of the automaton in Fig. 4c to allow this automaton to guess when the scenario of interest begins.

3.6 Assertion and Negation Operators

The *negate* operator provides a mechanism for specifying undesirable (negative) scenarios and the *assert* operator allows us to specify desirable (positive) scenarios. The former operator can be used to express safety properties, e.g., P_1 in Table 1, and the latter—finitary liveness properties, e.g., P_2.

Various formal treatments of the semantics of the *assert* and *negate* operators are given in the literature, e.g., [16], [17], [20]. These operators have a rich expressive power, and yet, their arbitrary combinations are not well understood. In particular, it is unclear whether negating an asserted scenario should mean that this scenario is not required to occur or that its negation has to occur. In this section, we define the semantics of *assert* and *negate* operators in terms of NFAs. Our formalization allows us to arbitrarily combine these operators as long as we never attempt to apply a *negate* operator to a sequence diagram containing an *assert*ed fragment.

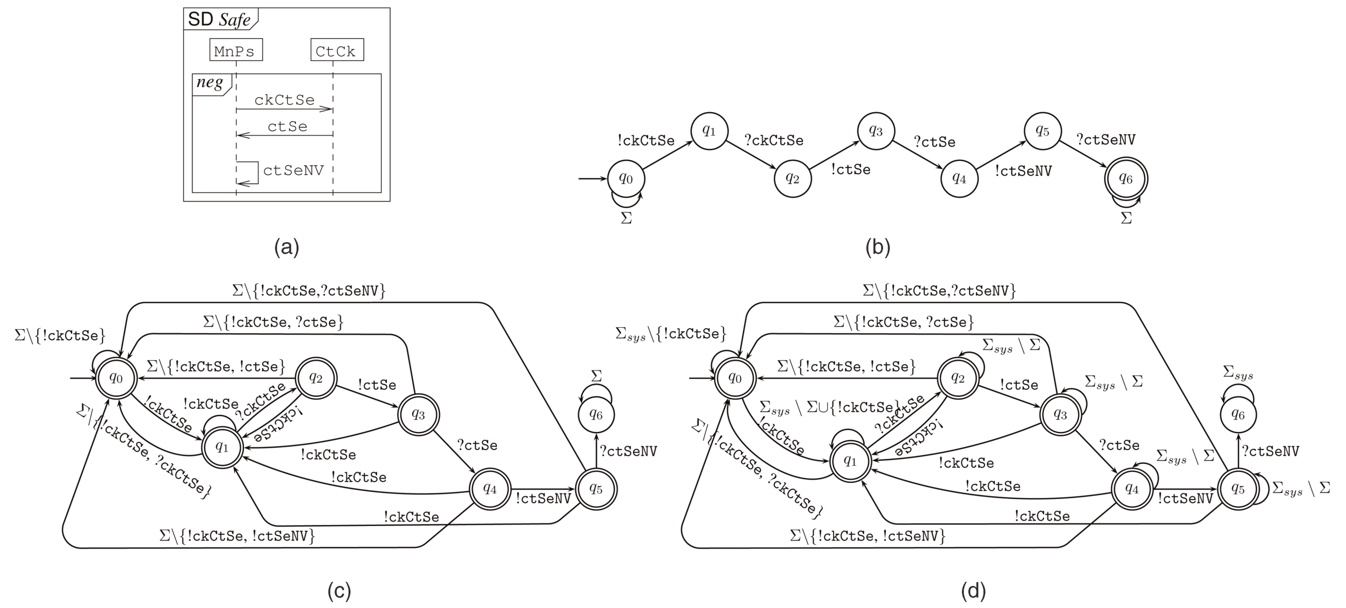

Fig. 5. (a) An SD describing P_1 in Table 1 and its corresponding NFAs: (b) before applying *negate*; (c) automaton after determinization and complementation; (d) the resulting monitor.

3.6.1 The Negate Operator

As mentioned above, *negate* allows us to express safety properties. By applying *negate* to an SD S, we indicate that the scenario represented by S is forbidden, and therefore, a safe system should never produce it [17]. For example, consider Fig. 5a, which shows an SD corresponding to the safety property P_1 in Table 1. `MnPs` sends a check credit score request, i.e., `ckCtSe`, to `CtCk`. In response, `CtCk` sends the actual credit score (`ctSe`) to `MnPs`. A `creditScoreNotValid` (`ctSeNV`) message is sent if the value is not in the correct range. This property is expressed in SD by applying a *negate* operator to the sequence `!ckCtSe.?ckCtSe.!ctSe.?ctSe.!ctSeNV.?ctSeNV`.

The *negate* operator over SDs is equivalent to the complementation operator of NFA. Given an SD S and its corresponding automaton A_S, we first add a self-loop transition labeled Σ, i.e., the underlying alphabet of S, to the initial state of A_S in order to enable A_S to guess when a satisfying run begins. Note that after adding this self-loop, A_S becomes nondeterministic. To obtain the automaton for the negated SD, we need to first determinize A_S and then complement the result.

For example, an automaton corresponding to the SD in Fig. 5a, after adding the self-loop and before complementation, is shown in Fig. 5b. Fig. 5c shows the final, complemented, automaton.

Note that since the sequence S is nonempty, the initial state of the complement of A_S is always accepting, and hence, the empty string is always in the language of the complement of A_S. This is expected because the negate operator holds 1) when the negative scenario S does not completely occur and 2) when no messages at all are exchanged.

3.6.2 The Assert Operator

The meaning of the *assert* operator is given by the UML standard as follows [13]: "*the sequences of the operand are the only valid continuations. All other continuations result in invalid behavior*." This interpretation has been formalized in different ways [16], [17]. The one that we have adopted is that of [16], which is described as follows: Given an asserted behavior $\sigma = \sigma_0 \ldots \sigma_n$ and a system behavior σ', every occurrence of σ_0 in σ' should be followed by the rest of σ. Thus, an SD with an *assert* is interpreted universally: "for every run, once it satisfies the start of the sequence, it must complete the sequence before termination." Note that the difference between *assert* and *critical* is that the former checks all possible suffixes of the input run to probe the sequence, whereas the latter only checks the first occurrence of its sequence.

In [16], *alternating* automata with *universal* initial states are used to capture this meaning of *assert*. Such automata accept a trace if *all* of the runs emanating from their initial states are accepting. NFA, however, accepts a trace when *there exists* an accepting run emanating from the initial state. Rather than moving outside NFA (and thus, complicating the monitoring framework), we chose to reinterpret the acceptance for the *assert* operator instead: An NFA for an asserted trace σ checks all suffixes of the system traces, and if one is not accepted, a failure is reported. This "universal" treatment is given to the entire sequence diagram, not just the part containing *assert*. This works correctly as long as such NFAs are not complemented or composed (in parallel)—the negation and parallel composition operators over automata with universally interpreted acceptance are different from those operators of NFA. While negation and parallel composition operators for NFA are computed via subset construction and cross-product, respectively, these operators for the alternating automata simply convert universal states into existential or add an additional universal state, respectively [21]. Thus, we restrict the application of *negate* and *par* to SDs that do not contain an *assert*, as described in Section 2.

Since alternating automata can be converted into NFA with a possibly exponential blowup in size, we could have

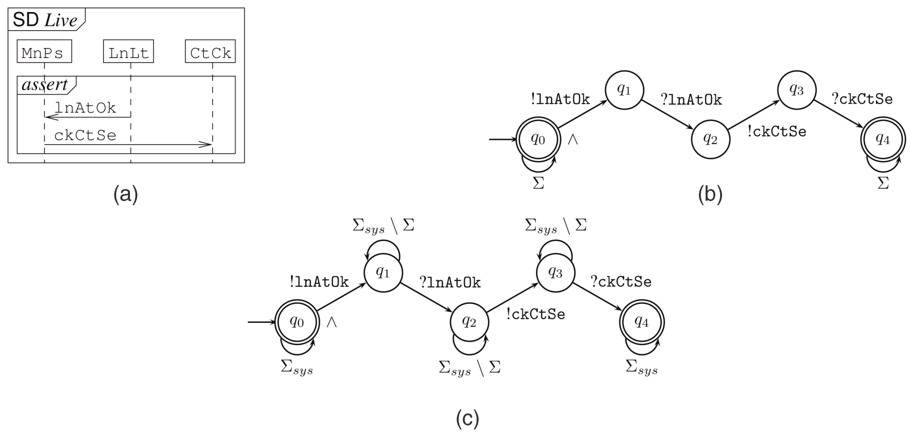

Fig. 6. (a) An SD describing P_2 in Table 1 and its corresponding NFAs: (b) after applying *assert*; (c) the resulting monitor.

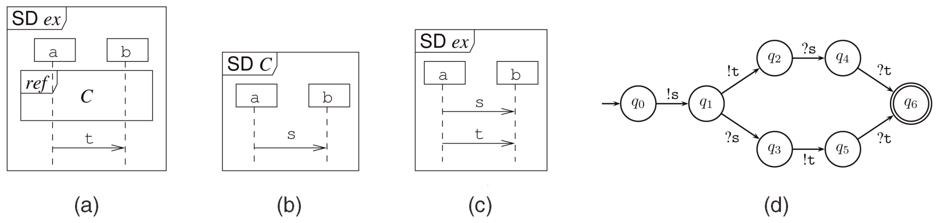

Fig. 7. (a) An SD with references SD C; (b) SD C; (c) SD ex after copying the content of SD C; and (d) its corresponding NFA.

translated the *assert* operator directly into NFA. However, we chose not to do it to preserve the succinctness and relatively small size of our monitoring automata.

Given the above discussion, the translation of *assert* operator is straightforward: After deriving the NFA A_S for SD S and adding a self-loop labeled Σ at its initial state, the automaton for *assert* S is obtained by interpreting the initial state as universal (we follow the notation of [16], denoting this state with a "$\wedge$") and making it accepting. For example, the SD in Fig. 6a describes the liveness property P_2 in Table 1—the desirable scenario is enclosed in the scope of an *assert* operator. Fig. 6b shows the automaton corresponding to this SD.

3.7 Interaction Use Operator

The *ref* operator is used for referring to an SD fragment from within another SD. Our treatment of *ref* is to inline the SD being referenced, as illustrated in Fig. 7.

3.8 Message Complementation

The message complement operator has been adopted from [18]. 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 $\Sigma = \{\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}\}$. Then, $\neg\mathtt{p} = \{\mathtt{q}, \mathtt{s}, \mathtt{t}\}$ and $\neg\{\mathtt{p}, \mathtt{q}\} = \{\mathtt{s}, \mathtt{t}\}$.

This operator, although not being part of UML 2.0, can be expressed in terms of UML operators as follows: Let $S \subseteq \Sigma$ be a set of messages. We replace $\neg S$ by an SD fragment in which the operator *alt* is applied to individual messages in $\Sigma \setminus S$. For example, consider the SD in Fig. 8a with a message $\neg\{\mathtt{p}, \mathtt{q}\}$, and let $\Sigma = \{\mathtt{s}, \mathtt{t}, \mathtt{p}, \mathtt{q}\}$. This SD is equivalent to the one in Fig. 8b, where $\neg\{\mathtt{p}, \mathtt{q}\}$ 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. 8c).

3.9 Generating Monitors from NFA

To be able to use an automaton A_S obtained from an SD S for runtime monitoring, we need to extend the language of A_S to handle system behaviors over alphabets larger than S. We do so by adding stuttering self-loops to the automaton's states. Semantically, this means that A_S does not change its state when the input symbol is outside the alphabet of S.

Definition 6 (Stuttering). *Let Σ_{sys} be the set of system events and $A = (\Sigma, Q, \delta, Q_0, F)$ be an NFA s.t. $\Sigma \subseteq \Sigma_{sys}$. The automaton $A' = (\Sigma_{sys}, Q, \delta', Q_0, F)$ is* the stutter-closed form *of A w.r.t. Σ_{sys} if $\delta' = \delta \cup \{(q, \Sigma_{sys} \setminus \Sigma, q) \mid \forall q \in Q\}$.*

The transformation of Definition 6 is language preserving.

Theorem 3. *Let $A = (\Sigma, Q, \delta, Q_0, F)$ be an NFA and Σ_{sys} s.t. $\Sigma \subseteq \Sigma_{sys}$ be given. Let A' be* the stutter-closed form *of A w.r.t. Σ_{sys} (see Definition 6). Then, for every trace $\sigma \in \Sigma_{sys}$, $\sigma \in L(A')$ iff $\sigma \downarrow_{\Sigma} \in L(A)$ (see Definition 1).*

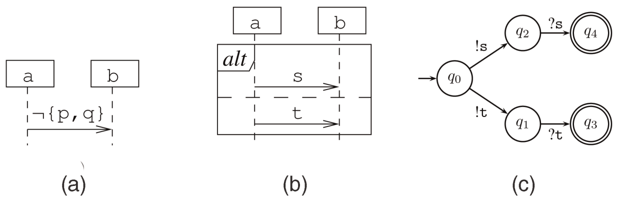

Fig. 8. (a) An SD with message complementation; (b) the same SD after eliminating the *complement* operator if its underlying alphabet Σ is $\{\mathtt{p}, \mathtt{q}, \mathtt{s}, \mathtt{t}\}$; and (c) its corresponding NFA.

Proof. The proof follows from the fact that the construction of Definition 6 does not change the state space of A:

$$\begin{aligned} & \sigma \in L(A') \\ \Leftrightarrow \; & \text{(By definition of language acceptance in } A') \\ & \exists q_0, \ldots, q_{n+1} \in Q \cdot q_0 \in Q_0 \wedge q_{n+1} \in F \wedge \\ & \forall \sigma_i \in \sigma \cdot \delta'(q_i, \sigma_i, q_{i+1}) \\ \Leftrightarrow \; & \text{(By definition of } \delta') \\ & \exists q_0, q_{n+1} \in Q \cdot q_0 \in Q_0 \wedge q_{n+1} \in F \wedge \\ & \forall \sigma_i \in \sigma \downarrow_\Sigma \cdot \delta(q_i, \sigma_i, q_{i+1}) \\ \Leftrightarrow \; & \text{(By definition of language acceptance in } A) \\ & \sigma \downarrow_\Sigma \in L(A). \end{aligned}$$

□

For example, the monitor corresponding to the SD in Fig. 5a is shown in Fig. 5d. The language accepted by this monitor is

$$\begin{aligned} \Sigma^*_{sys} \quad \setminus \quad & \big(\Sigma^*_{sys} \cdot !\texttt{ckCtSe} \cdot (\Sigma^*_{sys} \setminus \Sigma)^* \cdot ?\texttt{ckCtSe} \cdot \\ & (\Sigma^*_{sys} \setminus \Sigma)^* \cdot !\texttt{ctSe} \cdot (\Sigma^*_{sys} \setminus \Sigma)^* \cdot ?\texttt{ctSe} \cdot \\ & (\Sigma^*_{sys} \setminus \Sigma)^* \cdot !\texttt{ctSeNV} \cdot (\Sigma^*_{sys} \setminus \Sigma)^* \cdot ?\texttt{ctSeNV} \cdot \Sigma^*_{sys}\big). \end{aligned}$$

That is, the monitor rejects a trace that begins with a check credit score request (ckCtSe), which gets received (perhaps with some events not in the vocabulary of this SD in the middle), followed by receiving the credit score (ctSe) and sending a message indicating that it is invalid (ctSeNV), followed by arbitrary events in the system. Thus, the behaviors during which the check credit score requests are made and results in an invalid credit score are rejected; these correspond to violations of property P_1.

The monitor for the SD in Fig. 6a is shown in Fig. 6c. Its language is

$$\begin{aligned} \big((\Sigma_{sys} \setminus !\texttt{lnAtOk})^* \cdot (& !\texttt{lnAtOk} \cdot (\Sigma_{sys} \setminus \Sigma)^* \\ & \cdot ?\texttt{lnAtOk} \cdot (\Sigma_{sys} \setminus \Sigma)^* \\ & \cdot !\texttt{ckCtSe} \cdot (\Sigma_{sys} \setminus \Sigma)^* \\ & \cdot ?\texttt{ckCtSe})^*\big)^*. \end{aligned}$$

This monitor accepts traces that either do not exhibit !lnAtOk at all, or, if !lnAtOk has been seen, exhibit the entire sequence $?\texttt{lnAtOk} \cdot !\texttt{ckCtSe} \cdot ?\texttt{ckCtSe}$. Traces not accepted by this monitor violate property P_2 of the LA system.

Note that we do not add stuttering self-loops to the *critical* regions because behavior specified in *critical* regions cannot be interleaved by other messages.

3.10 Complexity of the Translation

The size of an automaton A_S corresponding to a basic SD S, i.e., the number of states in A_S, is $O(n^k)$, where n is the number of events and k is the number of objects [15]. Applying the SD operators does not cause a significant increase in the size of the resulting automata except for the cases where we need to determinize these automata, which can exponentially increase their state spaces. However, in our experience, the generated automata have been very small (see Section 6). Obviously, it remains to see whether the approach scales to larger Web service systems and more complex properties.

3.11 Discussion

3.11.1 On Using Our Language in MDA Tools

In this paper, we formalized the *syntax* of the SD language using a context-free grammar (see Fig. 3). As discussed in Section 5.1, we used the Rational Software Architect (RSA) [22] plug-in for WebSphere to generate an editor for SD diagrams. To do so, we have identified a fragment of the UML metamodel that captures the SD operators described in Fig. 3 and specified logical properties constraining the nesting and ordering of these operators. We have implemented a separate Java module to check these constraints over the generated SD diagrams in our tool. In the future, we plan to encode these constraints as part of the metamodel by expressing them in the OCL [14]. This would make our editor reusable in other UML environments.

3.11.2 On the Expressive Power of Our Language

In this section, we provided a transformation from our language SD to NFA, showing that SD can capture safety and finitary liveness properties. Our transformation further shows that SD is not more expressive than regular expressions, i.e., the language that NFA recognizes.

The main restriction in SD is that we do not allow the nesting of *assert*s within the scope of *negate*s. For example, sequence diagrams such as the one shown in Fig. 20a are not included in SD (we discuss these diagrams in more detail in Section 6.2). However, this restriction is mainly syntactic because we can always push the *negate* operator down to the atomic level and reformulate the sequence diagram into a semantically equivalent one in which *negate* is not applied within the scope of *assert*. For example, Fig. 20b shows a sequence diagram, which is semantically equivalent to the one in Fig. 20a and is within the SD language.

Note that after removing the *negate* operator, the resulting sequence diagram may have brand new scenarios: to do the removal, we need to elicit the set of all possible scenarios complementary to the scenario enclosed by the *negate*. For example, the scenario *negate*(reserveHotel, hotelReserved) in Fig. 20a is replaced by two new scenarios, reserveHotel, timeout and reserveHotel, hotelNotReserved, in Fig. 20b. The process of enumeration and analysis of all possible alternative scenarios obviously require domain knowledge, and thus, cannot be automated in general. However, the online nature of our monitoring framework allows us to register for and collect the alternative scenarios with ease.

4 SD Templates for Temporal Logic Property Patterns

In this section, we study the expressive power of our SD language by using it to express temporal logic property patterns [1]. Property patterns have been shown to capture a wide variety of commonly used properties, and being able to express property patterns is a good indication of an expressive power of a new language.

We first provide an overview of property patterns in the following section and then introduce several SD templates and show how they can encode the property patterns in Section 4.2.

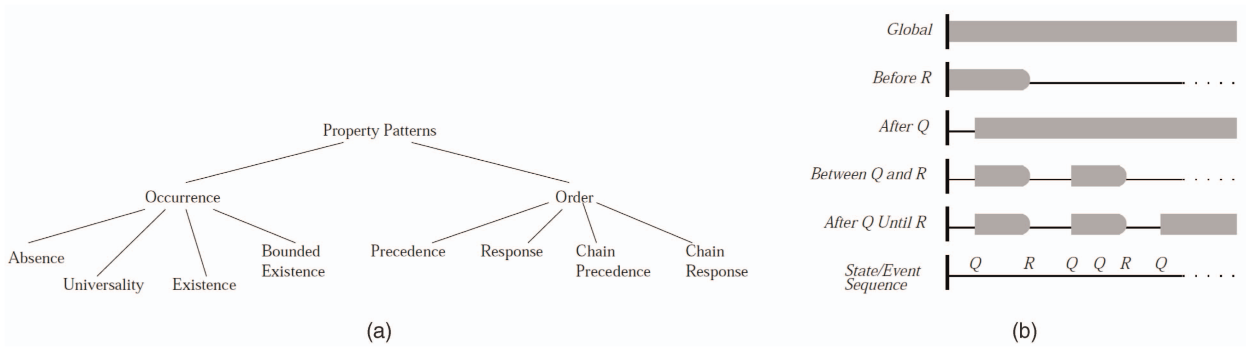

Fig. 9. Specification property system: (a) a pattern hierarchy and (b) pattern scopes.

TABLE 2
SPS Patterns

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.

TABLE 3
SPS Scopes

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.

4.1 Temporal Logic Property Patterns

The Specification Pattern System (SPS), proposed by Dwyer et al. [23], 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), computational tree logic (CTL) [24] and other state-based and event-based formalisms. SPS has been advocated as a standard tool for measuring the practical usefulness and expressive power of specification languages, e.g., [18] and [25].

The property patterns are organized into a hierarchy based on the kinds of system behaviors they describe (see Fig. 9a): **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 in Table 2.

Each pattern is associated with *scopes*—the regions of interest over which the pattern must hold. There are five basic kinds of scopes: **Global**, **Before**, **After**, **Between,** and **After-Until**. Definitions of these scopes are given in Table 3 and pictorially described in Fig. 9b.

For example, consider a property of a queue that says that there should be a `dequeue` event between every `enqueue` and `empty`. This is the **Existence** pattern, with the **Between** scope. Looking up the LTL formalization of this pattern/scope combination from the catalogue and substituting our event names, we obtain the formula

$$\Box((\texttt{enqueue} \wedge \neg\texttt{empty}) \Rightarrow (\neg\texttt{empty}\ W\ (\texttt{dequeue} \wedge \neg\texttt{empty}))).$$

4.2 Mapping Property Patterns to SDs

In this section, we provide several SD templates for the SPS patterns (see Fig. 11) and show how these templates are used to express patterns in the SPS hierarchy. Selected mappings are described below; the remainder can be found in the Appendix. Note that the actual direction of the arrows is determined when a template is instantiated.

Absence: message p cannot occur in a given scope. This can be expressed as shown in Fig. 11a.

Existence: a message p must occur in a given scope. This can be expressed as shown in Fig. 11b.

Until: This pattern is not part of the SPS; however, it is used to specify the **Precedence** patterns. A sequence $\texttt{p}^*$ of messages occurs until the first occurrence of message q, in a given scope (see Fig. 11h). This pattern, formalized using a single "until" temporal operator [24], can be refuted in one of the two ways: either p never occurs, or after seeing a finite number of p messages (expressed using *loop 1, n*), neither a p nor a q message occurs (expressed as $\neg\{\texttt{p},\texttt{q}\}$).

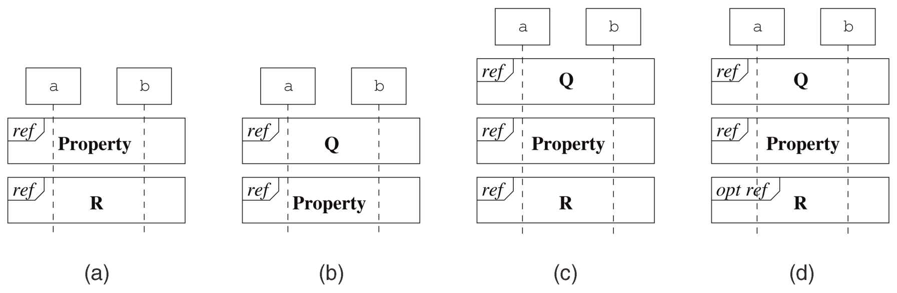

Fig. 10. Scope mapping for sequence diagrams: (a) **Before** R; (b) **After** Q; (c) **Between** Q **and** R; and (d) **After** Q **until** R.

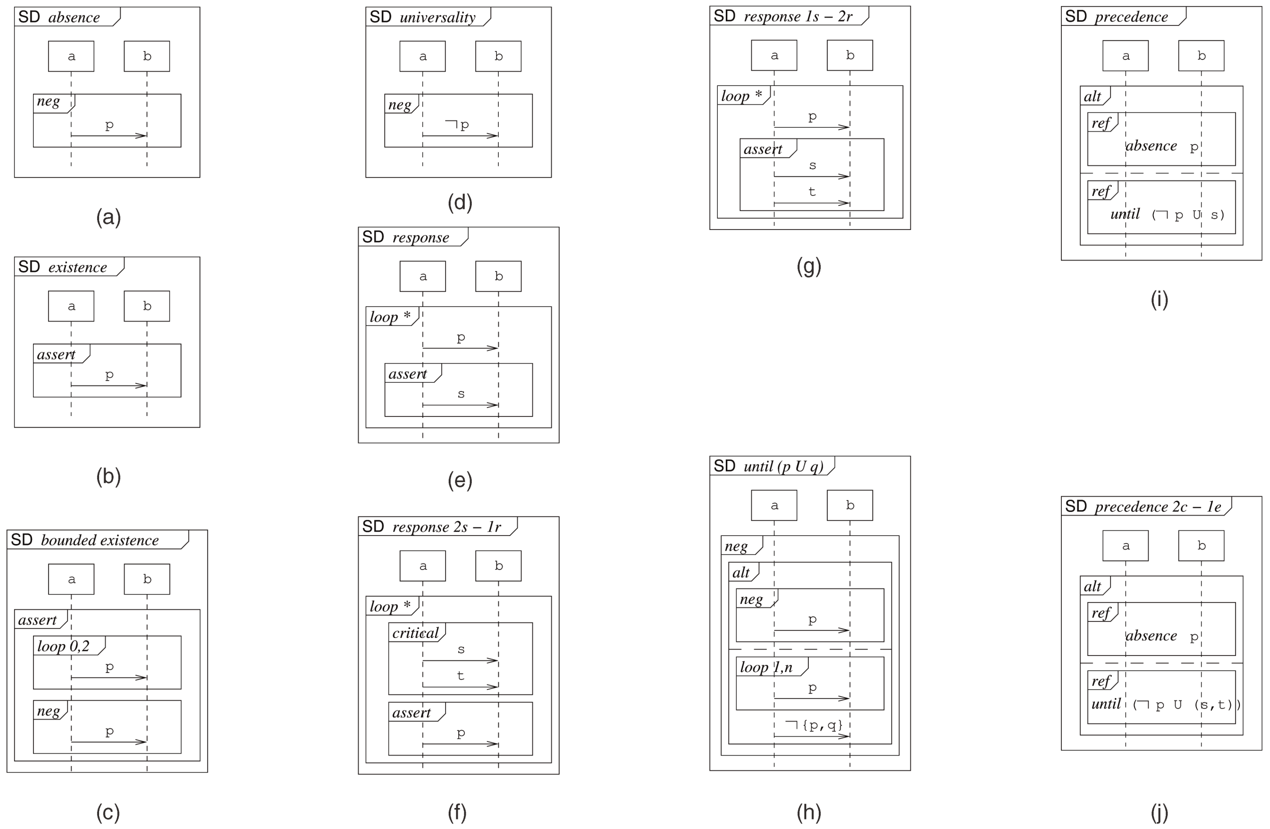

Fig. 11. Property pattern mappings for SDs: (s, t) means message s followed by message t. (a) **Absence**. (b) **Existence**. (c) **Bounded Existence**. (d) **Universality**. (e) **Response**. (f) **Response Chain:** 2 stimulus - 1 response. (g) **Response Chain:** 1 stimulus - 2 response. (h) **Until**. (i) **Precedence.** (j) **Precedence Chain:** 2 cause - 1 effect.

Precedence: a message s (cause) precedes a message p (effect), as shown in Fig. 11i. This pattern allows the cause part to occur without the effect. We describe this pattern in SD by expressing the two possible cases that this pattern specifies: 1) p never occurs or 2) p never occurs before s. The first case corresponds to checking *absence* of p; the second—to checking $\neg$p U s (the "until" template), 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 complex SDs rather than just the individual messages. In this case, we treat these symbols as placeholders, use a *ref* operator for the SDs that should be inserted in their place, and replace message complementation by negation.

4.3 Mapping Property Scopes

We now show how to express property patterns involving scopes that are used to define the traces over which a property will be monitored. Scopes can be simple messages or more complex scenarios in our specification language. The *ref* operator is used to introduce scope delimiters in the corresponding locations. 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. 10a). In the case of the **After** Q scope, the delimiter is inserted before the property (see Fig. 10b). Finally, both the **Between** (see Fig. 10c) and **After-until** (see Fig. 10d) 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. Thus, there is an implicit *opt* operator in each scope delimiter.

4.4 Specifying Properties of the Loan Application

We now show how property patterns can be used to express properties of the LA system given in Table 1. Properties P_1

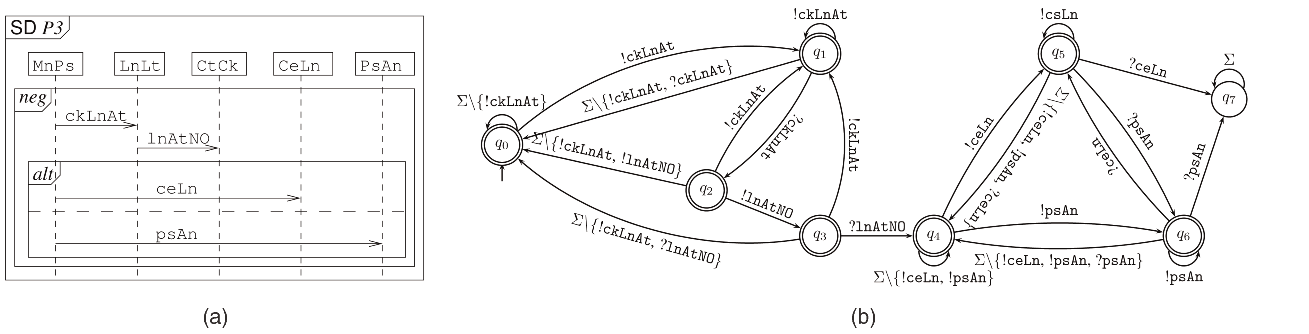

Fig. 12. P_3: **Absence** pattern. (a) SD describing the LA property P_3 and (b) the resulting monitor.

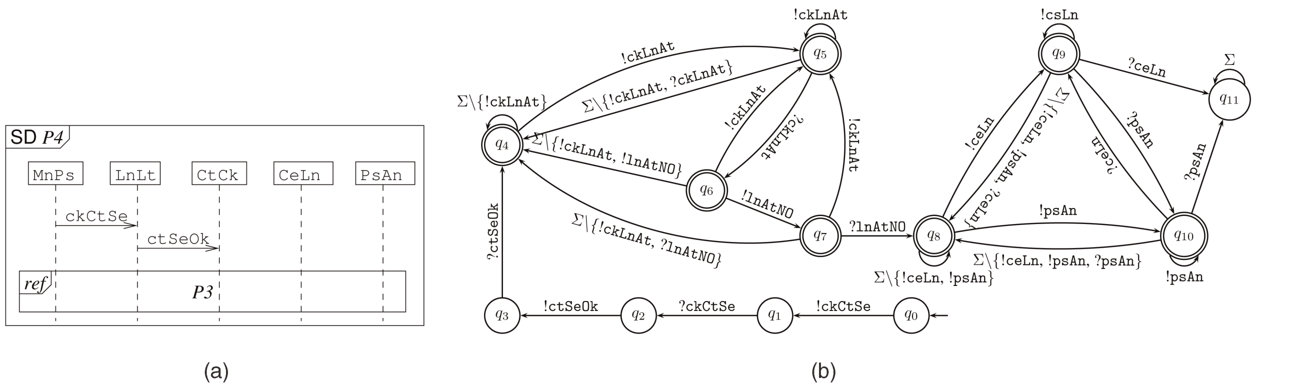

Fig. 13. P_4: **Absence** pattern, Scope **After**. (a) SD describing the LA property P_4 and (b) the resulting monitor, obtained by concatenating the NFAs for the scope and P_3.

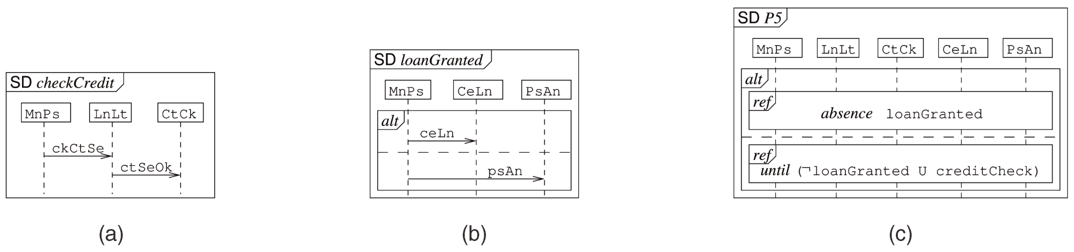

Fig. 14. P_5: The **Precedence** pattern. (a) SD for *checkCredit*; (b) SD for *loanGranted*; and (c) SD showing application of the **Precedence** pattern.

and P_2 are described in Figs. 5 and 6, respectively. The rest are discussed below.

Property P_3**:** "A loan cannot be granted if the loan amount is less than or equal to zero."

We express this property using the **Absence** pattern (see Fig. 11a): our property holds if there are no scenarios, where a loan is granted after the system has been warned that the loan amount is less than or equal to zero. In the LA system, the `LnLt` component checks the predicate "loan amount is >0," sending a `loanAmountOkay` (`lnAtOK`) message if the condition holds, and a `loanAmountNotOkay` (`lnAtNO`) message otherwise. A loan is considered granted if it is manually or automatically approved, which can be monitored by checking if the main workflow `MnPs` sends a `completeTheLoan` (`ceLn`) or `processTheApplication` (`psAn`) message. See Fig. 12a for the corresponding SD; the resulting monitor is shown in Fig. 12b.

Property P_4**:** (an example of a scoped property) "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."

This property is equivalent to the property P_3 with the **After** Q scope, where Q is "checking for a good credit score." To express it, we introduce the scope delimiter Q before the property P_3, as seen in Fig. 10b. The SD corresponding to P_4 is shown in Fig. 13a 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 resulting monitor is shown in Fig. 13b.

Property P_5**:** "No one can get a loan without first going through a credit check."

At this point, we have identified common scenarios that occur in the LA system: SDs *creditCheck* (Fig. 14a) and *loanGranted* (Fig. 14b). We can now express property P_5 using the **Precedence** pattern: SD *creditCheck* must precede SD *loanGranted*. Note that the SD *creditCheck* is not optional and must occur for the property to hold. The SD for P_5 is shown in Fig. 14c.

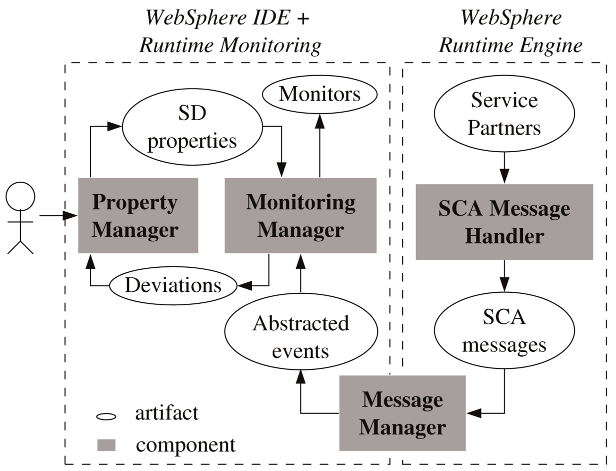

Fig. 15. Architecture of the framework.

5 Architecture and Implementation

We have implemented our runtime framework within the IBM WebSphere business integration products [26]. In what follows, we describe the architecture of our solution and discuss some of the more challenging parts of the implementation.

5.1 Architecture

Our solution uses the WebSphere Process Server [27] and the WebSphere Integration Developer [28]. 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.

The architecture of our framework is shown in Fig. 15. With the help of the *PropertyManager* (PM), users create UML SD specifications for their Web service applications. This component also checks if the user-specified properties belong to our SD subset and generates the corresponding NFA as a by-product of this check. If monitoring is enabled, the *MonitoringManager* (MonM) translates these NFAs into monitor automata using the techniques described in Section 3. During the execution of the Web service, *MessageManager* (MM) obtains interaction events from the *SCAMessageHandler* (MH) and directs the relevant events to MonM, which, in turn, updates the state of every active monitor automaton, until an error has been found or all partners terminate. The intercepted events are *never* stored, neither by the MH nor by the MM. We describe these components below.

PropertyManager consists of a graphical tool for specifying interaction properties as UML SDs. Once users create an SD and enable monitoring, the PM loads the XML model of the SD, checks that it uses the language subset described in Section 3, unwinds the partial order of the diagram into an NFA using the algorithm introduced in Section 3, and passes the NFA to MonM. In the case of a property failure, the PM is also responsible for displaying errors to the user.

SCAMessageHandler is deployed on the process server and establishes a bridge through which our runtime monitoring framework communicates with the server to obtain information about Web service execution. On the process server, the SCA is responsible for the invocation of native SCA service components and for the binding and interaction with external services. *SCAMessageHandler* monitors interactions within the SCA application server runtime environment and is responsible for observing and routing these invocation requests and responses to the correct components.

MessageManager is responsible for obtaining service request/response messages exchanged between business components from the SCA layer. MM, registered as a listener to *SCAMessageHandler*, intercepts events for operation invocation and filters out irrelevant messages such as locating a service. For the "interesting" events, MM extracts key information related to the operation invocation: what are the sender and receiver of the given message, whether the invocation is synchronous or asynchronous, what type of message is being exchanged, whether priorities are being used, etc. MM then packs all these information together with the time stamp of when the events were intercepted and sends them to the message queue associated with MonM via a TCP/IP communication channel.

MonitoringManager is the main component of our framework, as it constructs monitoring automata, processes events, and keeps track of the acceptance status of all monitors. Upon receiving a monitoring request together with the NFA representation of an SD from PM, MonM converts the NFA to a DFA and further to a monitor using the algorithms described in Section 3. To facilitate checking multiple properties for a single Web service system, MonM can manage a number of monitors simultaneously. Upon receiving an event from its message queue, MonM identifies those monitors that include this event as part of their communicating alphabets and changes their states according to their transition functions. All other monitors do not receive this event at all, which means that they stay in the same state. Note that this filtering mechanism used in our implementation differs from the one described in Section 3.9 (the stuttering step). The two are equivalent, and we used the stuttering construction in order to study the expressive power of our language. When updating the state of a monitor, MonM checks whether it is in a valid state; otherwise, it marks the corresponding property as being violated and records the erroneous event so that the PM is able to replay the error to the user.

5.2 Implementation

Since the WebSphere business integration tools are based on Eclipse, the functional components of our framework have been implemented as Eclipse plug-ins as well. Based on the architecture described in the previous section, we implemented four plug-ins. Fig. 16 depicts the interactions and dependencies among these, using double-arrowed lines.

Monitoring.Core Plug-in is the component corresponding to the *MonitoringManager* in the architecture. It consists of four packages: the *MonitorCore* package, which acts as an entry point to *MonitorCore* plug-in; the *Monitor* package, responsible for receiving expanded SDs and translating them into monitor automata; the *EventAnalysis* package, which handles events received from *MessageManager Plug-in* and forwards relevant events to monitors; and the *Utilities* package, which provides automata-related manipulation functions.

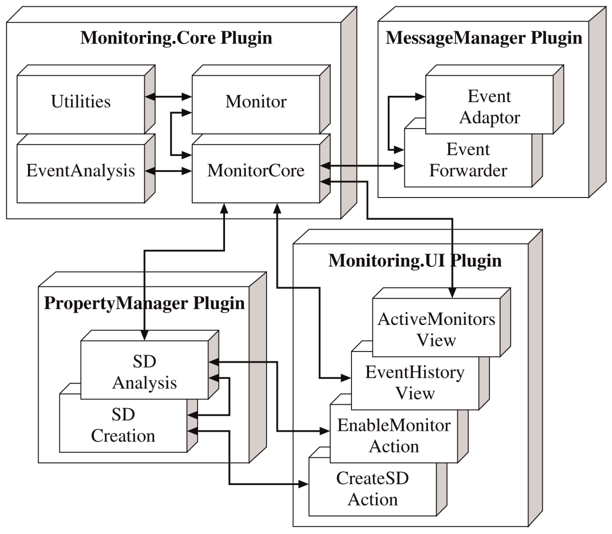

Fig. 16. The overview of the framework plug-ins.

MessageManager Plug-in implements the *MessageManager* functionality in the architecture. It contains two packages: *EventAdaptor* and *EventForwarder*. The *EventAdaptor* package registers itself as a listener to the *SCAMessageHandler* built into the WebSphere Process Server infrastructure, observing all invocation events flowing in the server SCA layer. To be effective, the *EventAdaptor* needs to be deployed into the server. Thus, when the server runs, the change made by the package is picked up by the WebSphere Process Server. The *EventForwarder* package simply acts as a bridge between the *EventAdaptor* package and the *MonitorCore* package to transfer events from the former to the latter. Since the *EventAdaptor* and *EventForwarder* run in the different address spaces, the communication between them is established through a TCP/IP socket. Specifically, the *EventForwarder* acts as the server role in a socket while the *EventAdaptor* takes the client side. Whenever it observes an event in the SCA layer, the *EventAdaptor* sends it to the socket port.

PropertyManager Plug-in corresponds to *PropertyManager* functionality in the architecture. It contains all Sequence Diagram-related functionalities, which are grouped into two packages. The *SDCreation* package adopts an existing graphical UML package provided by WebSphere as the Sequence Diagram editor. This existing graphical UML package, which acts as the front-end of the *SDCreation*, stores SDs in XML format and further provides the data structure along with APIs to manipulate SDs in memory. The back-end of the *SDCreation* is responsible for checking whether specified objects and messages are valid in a Web service composition when users use them to create a property for monitoring. The *SDAnalysis* package is where user-specified SD properties get translated into NFAs. It recursively traverses the data structure passed in from the front end to extract all SD constructs and unfold the partial order. The current implementation supports all operations introduced in Section 2. In our framework, we adopted the implementation of compositional operations over automata from the Charmy project [29].

Monitoring.UI Plug-in serves as an extension point to the framework and provides various graphical interfaces that users need to interact with the runtime monitoring tool. For example, *CreateSDAction* and *EnableMonitorAction* provide action icons in Eclipse for users to create an SD and then enable it for monitoring. The satisfaction of monitored properties and the system execution history can be seen in the *ActiveMonitors* and *EventHistory* windows, respectively.

Fig. 17 shows the screenshot of the user interface of our runtime monitoring framework. The *BusinessIntegration* view (panel in the top-left corner) shows the individual files of the LA system implementation. The panel in the middle of the window is the editor for creating SDs and viewing the monitoring results. The bottom two panels belong to the runtime monitoring framework. The tab on the left is the *ActiveMonitors* view, which lists all monitor-enabled properties. The view also shows the acceptance status of the monitored properties. The tab on the right is the *MonitorHistory* view from which users can trace the execution of Web services.

5.3 Other Implementation Issues

As mentioned in Section 3, in order to apply the *negate* operator, NFAs should be determinized. However, the determinization algorithm may result in an exponential blowup of the number of states. To keep the size of the automata small, we have used several optimization techniques such as reduction and minimization [19], adopting the implementation of these techniques from the BRICS package [30].

Although all generated automata are stored in memory and users do not need to use them directly, it is helpful to have an interface to allow viewing and debugging these automata. In our framework, we can store the generated automata in XML, and thus, enable displaying them in graphical automata editing tools such as JFLAP [31].

While Web services are terminating processes, they are meant to be repeatedly executed by different customers. In order to reuse the monitor for checking subsequent executions of the same Web service, we have implemented a resetting mechanism: As one execution terminates, an additional transition labeled `terminate`, added to all accepting states of the monitor, brings it back to the initial state.

BPEL supports the notion of *process instance*, so all messages include a process identifier as part of message header. Messages labeled with an existing process identifier are routed to the corresponding process; otherwise, a new process instance is started. By associating these process identifiers to our monitors, we can easily monitor multiple instances of the same process.

Because Web services are distributed and allow asynchronous message communication, messages may get delivered and received out of order. To handle out-of-order events, we annotate each event with two time stamps: one at invocation and the another at reception. When events arrive at the message queue of MonM, these time stamps are used to check if the invocation ordering is consistent with the reception ordering. If the orderings are not consistent, detected errors may be caused by network delays rather than incorrect conversations. Currently, all time stamps are generated by the same WebSphere Process Server.

We report monitoring results by displaying the status of each monitor in the *ActiveMonitors* view, tagging each property as satisfied, violated, or not yet conclusive.

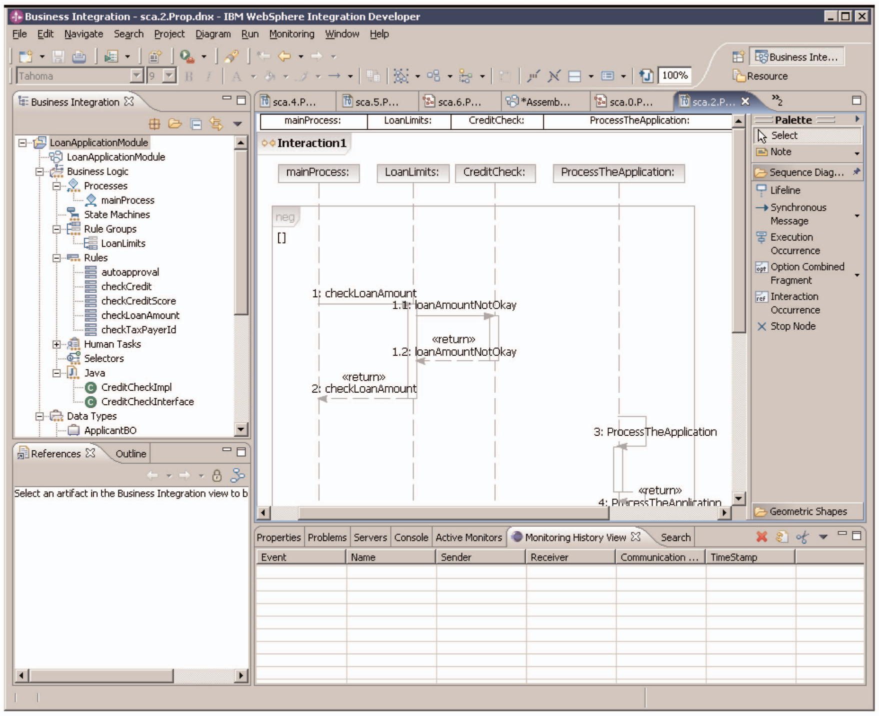

Fig. 17. Screenshot of the framework's user interface.

Clicking on satisfied or violated results displays a reason for the decision, in the Sequence Diagram editor. Table 4 gives a summary of the feedback provided by our framework as follows:

- For monitors for individual positive scenarios, if a given trace is accepted, the Sequence Diagram Editor shows the appropriate SD, with the observed trace highlighted. If the given trace is not satisfied by such a monitor, the answer to whether the system can exhibit such behaviors is inconclusive.
- Acceptance by a monitor for negative scenarios indicates that the appropriate safety property is violated, which is depicted by highlighting the appropriate trace (see Fig. 18a). Certainly, a failure to observe the violation on a given trace does not mean that every trace will satisfy the property; thus, in this case, the property is marked inconclusive.
- A monitor for a universal positive property, representing finitary liveness, is violated if the desired sequence has been started but has not been finished before the process terminated. This is indicated by the red line labeled "TERMINATE." If the sequence has been observed to completion, or if the process failed to terminate, no information about the satisfaction of this monitor can be given, deeming it inconclusive.

We also display the termination point in the case of individual positive scenarios, showing that the given trace is a prefix of an acceptable scenario.

TABLE 4
Summary (Answer/Feedback) of the Results from the Monitoring Framework

Property Type	Existential		Universal Positive
	Positive	Negative	
Violation		Violated/ highlight trace	Violated/ highlight termination location
Satisfaction	Satisfied/ highlight trace		

Empty cells indicate inconclusive results.

6 Experience

We have applied our framework to several Web services and report on results of monitoring them by running our tool on the WebSphere Process Server V6.0 (WPS) and WebSphere Integration Developer V6.0.1 (WID). Table 5 shows the details of the properties we specified and checked. In the table, column "Id" contains a unique identifier for each property; "Property" is the actual property to be checked; "# Part." corresponds to the number of partners involved in the corresponding SD; "# Events" is the number of events sent between partners in the SD; "# States" corresponds to the number of states in the

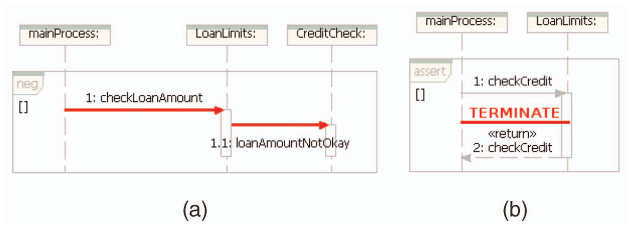

Fig. 18. Reporting errors: (a) A complete (negative) trace. (b) An incomplete sequence: violation of a liveness property.

TABLE 5
Properties and Sizes of Their Automata Representations

Id	Property	Pattern	# Part.	# Events	# States	# Trans.
	THE LOAN APPLICATION SYSTEM					
P_1	The credit score should always be valid, i.e., between 300 and 850	**Absence**	2	6	7	18
P_2	The credit score should eventually be checked if the loan amount is greater than zero	**Existence**	2	4	5	9
P_3	A loan cannot be granted if the loan amount is less than or equal to zero	**Absence**	5	8	8	23
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	**Absence**; Scope **After**	5	10	12	27
P_5	No one can get a loan without first going through a credit check	**Precedence**	5	8	28	95
	THE TRAVEL BOOKING SYSTEM					
P_6	The TB system should not reserve hotel room without checking the customer's credit first	**Absence**	3	6	5	14
P_7	A customer should be eventually notified about the status of his/her travel booking request, whether the reservation succeeds or not	**Existence**	2	4	5	9
P_8	A customer cannot make a reservation until his/her credit is checked	**Precedence Chain**	3	4	7	16
P_9	After the TB system receives a booking request, the customer will eventually receive a feedback message	**Response Chain**	2	6	6	15
P_{10}	If the customer's credit card is good, the TB system should attempt to make hotel, flight and car reservations	**Response**	5	10	31	89
P_{11}	The TB system should receive confirmation messages for all reservation attempts before generating a confirmation message	**Precedence**	5	8	21	48
P_{12}	No matter what happens after the customer submits a travel booking request, the customer will receive a feedback message	**Existence**	2	4	5	9
P_{13}	If the customer's credit is not good, the TB system should not make any charges to the customer's credit card	**Absence**; Scope **After**	5	8	8	22
P_{14}	If any reservations are made, the customer must be informed	**Response**	5	8	7	25
P_{15}	If a reservation cannot be made, inform the customer	–	–	–	–	–
	THE ONLINE SHOPPING SYSTEM					
P_{16}	A premium customer always gets a discount on his/her purchase	**Absence**	3	8	7	30
P_{17}	An order cannot be billed before being marked complete by the customer	**Absence**	3	6	6	23
P_{18}	A completed order will be eventually billed	**Existence**	3	4	5	15

corresponding automaton; and "# Trans." is the number of transitions in the automaton. Note that all of the constructed automata have fever than 100 transitions. While the system generates a large number of messages, our monitors receive just those within the scope of the automata; the rest are filtered. Furthermore, the intercepted events are never stored. Thus, enabling monitoring does not produce a significant performance overhead.

6.1 Monitoring the LA System

The LA system, introduced in Section 1.1, consists of six partners and six invocation-type activities, with the workflow shown in Fig. 1a. This application comes as part of the WebSphere Integration Developer v6.0.2. As it is a *sample* application, the original developers of the application have simplified some of the business logic, e.g., the `CreditCheck` component generates random credit scores rather than access the credit bureau.

We began by testing the system to see if the application was correctly deployed. To do this, we ran it on two different taxpayer ids and three different loan amounts, with the following specific input configurations:

$$c_1 = <\text{taxpayer id} = 1{,}234, \text{loan amount} = \$10{,}000>,$$
$$c_2 = <\text{taxpayer id} = 1{,}234, \text{loan amount} = \$60{,}000>,$$
$$c_3 = <\text{taxpayer id} = 1{,}888, \text{loan amount} = -\$1{,}000>.$$

As the system is supposed to generate random valid credit scores, we ran the system 10 times with each configuration. For configuration c_1, we expected to see some automatic approvals of the loan and some declines, based on whether the good or the bad score is generated. For c_2, we expected some manual approvals of the loan (the loan amount is above the automatic approval limit) and some declines. Finally, since the loan amount in c_3 is invalid, we expected to see only loan rejections.

For configurations c_1 and c_2, the behavior we observed was as expected: P_1, P_2, P_5 always held and P_3, P_4 held when the loan was granted. However, for all executions of c_3, the system automatically approved the loan, meaning that properties P_3 and P_4 were violated. For all executions of c_3, the system produced the following faulty execution trace:

$$FT = (\texttt{MnPs}, \texttt{ckCtSe}, \texttt{LnLt}), (\texttt{LnLt}, \texttt{ctSeOK}, \texttt{CtCk}),$$
$$(\texttt{MnPs}, \texttt{ckLnAt}, \texttt{LnLt}), (\texttt{LnLt}, \texttt{lnAtNO}, \texttt{CtCk}),$$
$$(\texttt{MnPs}, \texttt{ceLn}, \texttt{CeLn}),$$

where each triple $(Sender, m, Receiver)$ denotes partner $Sender$ sending a message m to partner $Receiver$. The $(\texttt{LnLt}, \texttt{lnAtNO}, \texttt{CtCk})$ triple in this trace indicates that the loan amount is less than or equal to zero. In other words, the `LnLt` component checked the predicate "loan amount is >0," and sent a `loanAmountNotOkay` (`lnAtNO`) message because the predicate did not hold. Therefore, this trace depicts a failure

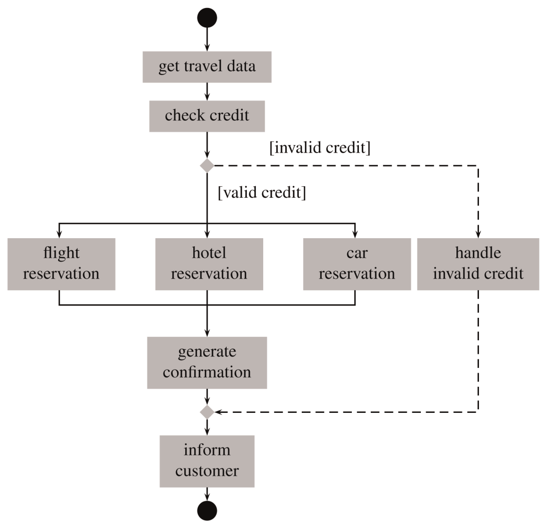

Fig. 19. The activity diagram of the TB system.

of P_3 because it includes an invalid behavior, the acceptance of the invalid loan, indicated by the subtrace

$$(\mathtt{MnPs}, \mathtt{ckLnAt}, \mathtt{LnLt}), (\mathtt{LnLt}, \mathtt{lnAtNO}, \mathtt{CtCk}), (\mathtt{MnPs}, \mathtt{ceLn}, \mathtt{CeLn}).$$

As P_4 is a scoped version of P_3, it also fails on this trace.

To identify the cause of the violations, we examined the BPEL diagram in Fig. 1a to see that the trace *FT* is produced if the LA system obtains the taxpayer's credit score, checks if the credit score is greater than 750 (`ScoreEvaluation`), checks if the loan amount is greater than zero (input validation), and checks if the loan amount is less than \$50,001 (`AutoApprovalTest`). The `ScoreEvaluation` should only occasionally be true, as the `CreditCheck` component generates random credit scores. However, we obtained trace *FT* every time the system was run with the taxpayer id 1888, i.e., the system always approved a negative loan.

We traced this behavior to two problems. The first, identified after looking at the BPEL code of the LA system, was that the application did not use the results of the input validation, allowing requests for negative loans to go through. The second problem was only identified after examining the source code for the `CreditCheck` partner. Instead of ignoring the taxpayer id and generating a random credit score, this component always returns a good credit score when the taxpayer id ends with "888." Combined, these two problems yielded the approval of the loan for configuration c_3 every single time.

Overall, our experience showed that the system can handle simultaneous failure of several monitors and allowed us to specify interesting properties, which led to the discovery of two real faults in the LA system.

6.2 Monitoring Other Applications

Additionally, we modeled and checked two other applications: the travel booking (TB) system and the Online Shopping (OS) System.

6.2.1 The Travel Booking System

TB acts as a broker offering its customers the ability to book all aspects of a trip. The workflow of TB system includes credit validation, flight/hotel/car reservation, and communication with the client. Customers can submit data about their desired travel plans and receive either a confirmation number or a failure message depending on whether the travel arrangements have been made successfully. The activity diagram in Fig. 19 shows high-level steps that are executed during the travel booking process.

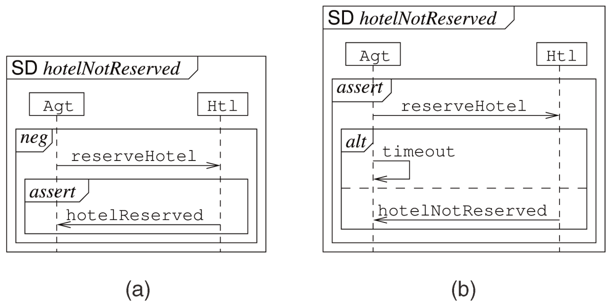

Fig. 20. Expressing property P_{15}: (a) using the existing alphabet of the TB system; (b) with additional events.

To fulfill its business goal, the TB system needs to interact with several partners: `CreditCardChecking` service, which validates the customer's credit card data; `FlightReservation` service, which books a flight; `HotelReservation` service, which reserves a hotel room; and `CarReservation` service, which makes a car reservation. In a typical scenario, an Internet customer begins an interaction with the TB system by entering data for his/her travel arrangements. The system then invokes the `CreditCardChecking` service, and if the credit card is valid, it tries to make hotel, flight, and car reservations. If all of the reservations are completed successfully, a confirmation number is generated and returned to the customer.

Table 5 lists properties we checked on this system (P_6-P_{15}). For example, P_6 includes six events among three partners and is represented by an automaton with six states and 23 transitions. Five properties, P_6, P_7, P_{12}, P_{13}, and P_{14}, are monitorable using patterns in the **Occurrence** hierarchy (see Fig. 9b). Four properties, P_8, P_9, P_{10}, and P_{11}, are monitorable using patterns in the **Order** hierarchy.

Property P_{15} can be expressed in UML 2.0 Sequence Diagram language but not in our specification language SD. The reason for this limitation is the chosen set of events of the TB system: Hotel reservations are handled only by two events: `reserveHotel` (the request) and `hotelReserved` (confirmation of the success). Thus, failure to reserve the hotel means that we were unable to receive the confirmation message. Since it is not clear how long the service should wait before declaring a failure, we have to express the property using an *assert* inside a *negate*, as shown in Fig. 20a, which is not allowed in our language (see Section 3.6.2). The problem can be fixed by adding two additional events to the TB system that give a reason why the hotel reservation fails: `timeout` (produced if a confirmation is not received by a certain time) and `hotelNotReserved` (produced if the reservation could not be obtained). With these, property P_{15} can be expressed as shown in Fig. 20b, which is within the SD language.

We checked properties P_6-P_{14} on two versions of the TB system: the complete system shown in Fig. 19 and a version where we removed the error handler for invalid credit cards (dashed links in Fig. 19). We did not detect any errors when running the complete system against these properties. When running the modified version of the system, the monitoring framework was able to detect a violation of the property P_7 when the user submitted a travel request with an invalid card, and reported this violation by showing that the event `displayResult` is missing. We believe that this feedback would have been useful for debugging of the Travel Booking System.

6.2.2 The Online Shopping System

This system implements a typical online shopping service and consists of four partners and 20 invocation-type activities. These activities are invoked via asynchronous or synchronous message passing. For a complete description of the system, see [32].

The first two properties, P_{16} and P_{17} in Table 5, are expressed using the **Absence** pattern. The remaining property, P_{18}, is expressed using the **Existence** pattern. We did not detect errors in the OS system when running it against these properties.

6.2.3 Summary

Overall, our experience showed that SD is a language expressive enough to capture a variety of properties of existing Web service applications, and all of the properties except one could be expressed using the pattern system. Expressing the remaining property required enriching the set of events in the corresponding system. Despite a potential exponential increase in the size of monitoring automata, we did not encounter it in examples we have tried, and thus, monitoring always yielded negligible overhead. Finally, the experience of encountering an error in an existing application, which resulted in a simultaneous failure of several monitors, allowed us to conclude that our framework can be used to facilitate effective debugging.

7 Related Work

The main contributions of our work are the definition of a runtime monitoring language and the creation of a dynamic runtime monitoring framework based on this language. Thus, we first summarize some work studying UML 2.0 Sequence Diagrams as a specification language. Afterward, we survey the research on runtime monitoring in the context of Web services.

7.1 Sequence Diagrams as a Specification Language

Like other partial-order scenario-based formalisms such as MSCs [11] and LSCs [12], UML Sequence Diagrams are enjoying an increasing usage as a specification language.

Lettrari and Klose [33] show how UML 1.3 Sequence Diagrams can be used to check properties of UML models. UML 1.3 SDs allow only simple event sequences, so the language formalized in [33] is a small subset of our specification language.

Ameedeen and Bordbar [34] show how a subset of UML 2.0 SDs can be transformed into Free Choice Petri nets, enabling the use of the corresponding analysis techniques. This SD subset is only used to specify possible system behaviors, and thus, does not include the *negate* and *assert* operators. This work also assumes that sending and receiving an event happen simultaneously. While this assumption works well for synchronous systems, it does not hold for most Web applications that rely on message queues for communication.

Autili et al. [18] propose a 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.

STAIRS [35] is a trace-based requirement specification methodology that also uses extended UML 2.0 SDs. Trace scenarios are classified into positive (mandatory and potential), negative, and inconclusive. Negative traces are captured using the *negate* operator. STAIRS does not use *assert* and instead defines a new mandatory choice operator, *xalt*, to express the requirement that both alternatives be present in a choice. In our work, we enable expression of mandatory and forbidden behaviors without extending the language.

Grosu and Smolka [17] interpret positive and negative UML 2.0 Sequence Diagrams as safety and liveness properties and give formal semantics for such diagrams using Safety and Liveness automata, respectively. Their approach does not use the *assert* operator and defines automata over infinite traces.

Harel and Maoz [16] define Modal Sequence Diagrams (MSD), an extension of UML 2.0 Sequence Diagrams. The semantics of *negate* and *assert* operators in MSD is given via the universal/existential distinction made by the LSCs [12]. In this formalism, diagrams, messages, and constraints can be defined as either *hot* (universal) or *cold* (existential), and the semantics of MSDs is given via alternating weak word automata (AFA). This formalism includes not only nondeterministic choices of NFA (the language into which we translated SDs) but universal choices as well [16]. Given that any AFA can be translated to an (exponentially larger) NFA [21], we believe that SDs and LSCs have the same expressive power. These languages, however, differ in their syntactic and usability properties. Specifically, LSCs are more succinct because they can freely combine nondeterministic and universal choices. However, SDs are easier to implement and use in a monitoring framework because of the existence of several efficient packages for manipulating NFAs. Moreover, unlike LSCs, the syntax of SDs conforms to UML 2.0, and hence, many existing UML tools can be used to capture and display these diagrams.

The same authors discuss how LSC specifications can be used to monitor the execution of the program using aspects [36] (a method used by several runtime monitoring frameworks described in the literature, e.g., [37]), but the exact translation from alternating automata into AspectJ and the resulting complexity of the approach are unclear. Finally, an existential, *constant*, subset of LSCs has been expressed in terms of NFA [38]. It is a strict subset of SDs, not allowing universal traces.

While we concentrated on specifying behavioral properties of interactions between partners, Bultan [39] identified Collaboration Diagrams (CDs) and Conversation Protocols (CPs) as more appropriate formalisms for specifying such properties as realizability and synchronizability, which he then checks using model-checking. These formalisms are simpler than UML 2.0 Sequence Diagrams and are appropriate for expressing such special-purpose properties.

7.2 Runtime Monitoring of Web Services

In this section, we compare our approach to existing runtime monitoring techniques. Online (offline) techniques analyze system events during (after) execution and the properties to check are determined a priori (a posteriori). Examples of the online techniques are [40], [41], [42], [43], [44], [45], and [46], [47], [48] are the offline techniques.

These techniques differ in the types of properties they can handle. *Global properties* allow the analysis of orchestrated obligations. These obligations are expressed from the point of view of the orchestrating service, but also include events from the other services involved in the conversation being monitored. *Local properties* are restricted to monitoring the events of a single service. Furthermore, some techniques concentrate on *state* properties, whereas others allow the user to express *sequences* of events. Like ours', the approach introduced by Pistore and Traverso [43] can be used to check global properties. In [43], properties are specified in LTL, which is more expressive than our specification language. However, specifying properties correctly in LTL can be challenging, especially when trying to specify sequences of events [1], whereas these are quite intuitive in our framework.

The frameworks described in [40], [41], [42], [44], [45] are restricted to local properties. Li et al. [45] specify properties using Interaction Constraints (ICs) [49]—a language based on Dwyer et al.'s Specification Pattern System [1]. Unlike our specification language, IC does not allow pattern nesting. Thus, new events must be introduced in order to reason about sequences of events. The rest of the local property frameworks check state formulas, specified using simple predicate logic. Specifically, Baresi et al. [41], [42] and Lohmann et al. [44] check service pre- and postconditions associated to external service invocations, while Lazovik et al. [40] check local assertions.

Offline techniques can handle both global and local properties. In the work of Mahbub and Spanoudakis [46], [47], properties are expressed using event calculus [50]. van der Aalst and Pesic [48] introduce DerSecFlow, a graphical language that can be used to express properties similar to our patterns, but without pattern nesting.

Various techniques are used for *checking* properties. The authors of [40] and [43] rely on planning techniques to create service compositions. Pistore and Traverso [43] analyze the application once the composition has been obtained, by instrumenting the system to include Java code that checks LTL monitors during runtime. Lazovik et al. [40] iteratively replace the violated service with another one, with weaker assertions, continuing the process until there are no more violations, or the composition is not possible.

In the case of service pre- and postconditions, the authors of [41], [42] modify the original BPEL diagram, introducing new BPEL activities that check the contract during external service calls. The authors of [44] propose a similar, but more intrusive framework, as JML contracts are integrated at the source code level. The authors of [46], [47] use temporal deductive databases to store and reason about events generated during runtime, while the authors of [48] analyze low-level event logs using an LTL checker.

Techniques used in the work of Li et al. [45] are the closest to ours. Like us, they take an automata-based approach for monitoring communications between partners and enable graphical display of violations.

As discussed before, the advantage of online techniques is that it is possible for the system to react once a problem has been detected. In [41], [42], BPEL exception handlers can be attached to the properties being checked. If such an exception handler is not provided, execution terminates when a violation occurs. As [43], [44] are Java-based, they can use Java's exception handling mechanism for recovery actions; however, this approach is highly intrusive. Li et al. [49] do not discuss recovery. Offline techniques like [46], [47], [48] instead *suggest* corrective actions, which can be tested during future executions.

8 Conclusion and Future Work

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. It is nonintrusive, running in parallel with the monitored system and intercepting interaction events during runtime. 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 safety and liveness properties. Liveness becomes finitary, where user-specified time limits or the process termination acts as the stopping event.

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. We have also created a library of such sequence diagram patterns and shown how patterns can be used to specify monitors for a number of interesting properties of several Web service applications. Finally, we reported on the implementation of our framework which allowed us to find bugs in real Web service applications.

8.1 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, in the LA system, 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. BPEL supports the notion of process instances and encodes a process ID in all message headers so as to identify which process instance is the intended recipient of the message. We believe that our framework accommodates this approach readily, by encoding these process IDs into the specification, the automata transition relation, and interaction events.

Currently, our framework permits the definition of properties that depend only on the order and occurrence of system events. By monitoring the actual *data* exchanged by conversation participants, we could check richer properties that depend on such data. We cannot use the existing automata translations for data exchange properties directly because the resulting automata would be too large to be useful for monitoring. Instead, we are currently investigating the use of Parameterized NFA [51] (PNFA) to create more succinct monitors, as single-PNFA transitions represent sets of NFA transitions.

Current BPEL recovery mechanisms are not suitable for developing self-healing Web services, as error handling and compensation mechanisms must be defined before deployment. As discussed in Section 1, online runtime monitoring techniques allow dynamic recovery, since recovery strategies can be applied as soon as errors are detected. Existing work [52], [53] focuses on the definition of recovery strategies for *local properties*, assuming that process definition is correct and errors are introduced only via interactions with external services. The recovery strategies are suggested "per message." Specifically, Baresi et al. [52] check external service pre- and postconditions to determine when a partner link should be modified, while Moser et al. [53] use QoS parameters. Our approach allows us to define recovery strategies suitable for *global properties*, i.e., define them "per conversation." We also want to study recovery strategies that dynamically modify the BPEL process definition [54].

We also plan to investigate techniques to help locate *causes* of errors (as opposed to places where a violation was detected) from observing results of successful and unsuccessful runs of the system. We will experiment with the techniques in [55], [56] for this task.

Finally, our work so far has assumed 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 [57] to turn a centralized monitor into a set of distributed ones, running in different process servers.

APPENDIX

Below, we continue the discussion of expressing property patterns in SD, started in Section 4.

k—Bounded Existence: Message 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 p does not occur, which corresponds to the **Absence** pattern (see Fig. 11c).

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

Response: Message p (stimulus) must be followed by message 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 is checked in an infinite *loop*, as there can be many stimulus/response pairs in one execution trace (see Fig. 11e).

Response chain: A sequence $p_1, \dots, p_n$ of messages must be followed by the sequence $q_1, \dots, q_m$ of messages in a given scope. We show two examples of this pattern: p responds to s, t (see Fig. 11f) and s, t responds to p (see Fig. 11g). This pattern has the same basic form as **Response**.

- p responds to s, t: 2 stimulus—1 response. The *critical* operator is used to enclose the message sequence s, t, to ensure atomicity of this sequence. An *assert* cannot be used since the stimulus sequence is optional.
- s, t responds to 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 $p_1, \dots, p_n$ of messages must precede the sequence $q_1, \dots, q_m$ of messages in a given scope. We show an example of this pattern, 2 cause—1 effect, p is preceded by s, t (see Fig. 11j). This pattern is implemented 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.

ACKNOWLEDGMENTS

The authors thank 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, MITACS Accelerate program, and NSERC. IBM and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the US, other countries, or both. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the US, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

REFERENCES

[1] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st Int'l Conf. Software Eng. (ICSE '99)*, pp. 411-420, May 1999.
[2] OASIS, *Web Services Business Process Execution Language Version 2.0*, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Jan. 2009.

[3] X. Fu, T. Bultan, and J. Su, "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services," *Proc. Eighth Int'l Conf. Implementation and Application of Automata (CIAA '03),* pp. 188-200, July 2003.

[4] X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services," *Proc. 13th Int'l World Wide Web Conf. (WWW '04),* pp. 621-630, May 2004.

[5] R. Kazhamiakin and M. Pistore, "A Parametric Communication Model for the Verification of BPEL4WS Compositions," *Proc. Int'l Workshop Web Services and Formal Methods (WS-FM '05),* pp. 318-332, 2005.

[6] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella, "Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step," *Proc. Int'l Workshop Web Services and Formal Methods (WS-FM '05),* pp. 257-271, 2005.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-Based Verification of Web Service Compositions," *Proc. 18th IEEE Int'l Conf. Automated Software Eng. (ASE '03),* pp. 152-163, 2003.

[8] N. Ghafari, A. Gurfinkel, N. Klarlund, and R. Trefler, "Algorithmic Analysis of Piecewise FIFO Systems," *Proc. Seventh Int'l Conf. Formal Methods in Computer-Aided Design (FMCAD '07),* pp. 45-52, Nov. 2007.

[9] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Trans. Software Eng. and Methodology (TOSEM),* vol. 3, no. 2, pp. 131-165, 1994.

[10] M. Smith, G. Holzmann, and K. Etessami, "Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs," *Proc. Fifth IEEE Int'l Symp. Requirements Eng. (RE '01),* pp. 14-22, Aug. 2001.

[11] ITU-TS, "ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96)," technical report, ITU-TS, Geneva, 1996.

[12] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *J. Formal Methods in System Design (FMSD),* vol. 19, no. 1, pp. 45-80, 2001.

[13] Object Management Group (OMG), *Unified Modeling Language (UML 2.0),* http://www.omg.org/spec/UML/2.0/, Jan. 2009.

[14] Object Management Group (OMG), *Object Constraint Language (OCL 2.0),* http://www.omg.org/spec/OCL/2.0/, Jan. 2009.

[15] R. Alur and M. Yannakakis, "Model Checking of Message Sequence Charts," *Proc. 10th Int'l Conf. Concurrency Theory (CONCUR '99),* pp. 114-129, 1999.

[16] D. Harel and S. Maoz, "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams," *Proc. Int'l Conf. Software Eng. (ICSE '06) Workshop Scenarios and State Machines (SCESM '06),* pp. 13-20, 2006.

[17] R. Grosu and S.A. Smolka, "Safety-Liveness Semantics for UML 2.0 Sequence Diagrams," *Proc. Fifth Int'l Conf. Application of Concurrency to System Design (ACSD '05),* pp. 6-14, 2005.

[18] M. Autili, P. Inverardi, and P. Pelliccione, "A Scenario Based Notation for Specifying Temporal Properties," *Proc. Int'l Conf. Software Eng. (ICSE '06) Workshop Scenarios and State Machines (SCESM '06),* 2006.

[19] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory Languages and Computation.* Addison Wesley, 1979.

[20] H. Störrle, "Assert, Negate and Refinement in UML 2 Interactions," *Proc. Unified Modeling Language (UML '03) Workshop Critical Systems Development with UML,* pp. 79-94, 2003.

[21] M. Vardi, "An Automata-Theoretic Approach to Linear Temporal Logic," *Proc. Eighth Banff Higher Order Workshop,* pp. 238-266, Aug. 1996.

[22] IBM, *IBM Rational Software Architect,* http://www.ibm.com/software/awdtools/architect/swarchitect, Jan. 2008.

[23] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Property Specification Patterns for Finite-State Verification," *Proc. Second Workshop Formal Methods in Software Practice (FMSP '98),* Mar. 1998.

[24] E. Clarke, O. Grumberg, and D. Peled, *Model Checking.* MIT Press, 1999.

[25] J. Yu, T.P. Manh, J. Han, Y. Jin, Y. Han, and J. Wang, "Pattern Based Property Specification and Verification for Service Composition," *Proc. Seventh Int'l Conf. Web Information Systems Eng. (WISE '06),* pp. 156-168, 2006.

[26] IBM, *WebSphere Business Integration Software,* http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint, Jan. 2009.

[27] IBM, *WebSphere Process Server,* http://www-306.ibm.com/software/integration/wps, Jan. 2009.

[28] IBM, *WebSphere Integration Developer,* http://www-306.ibm.com/software/integration/wid, Jan. 2009.

[29] P. Inverardi, H. Muccini, and P. Pelliccione, "CHARMY: An Extensible Tool for Architectural Analysis," *Proc. 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE '05),* pp. 111-114, Sept. 2005.

[30] A. Møller, *dk.brics.automaton,* http://www.brics.dk/automaton, Jan. 2009.

[31] Duke Univ., *JFLAP,* http://www.jflap.org, Jan. 2009.

[32] Y. Gan, "Runtime Monitoring of Web Service Conversations," master's thesis, Dept. of Computer Science, Univ. of Toronto, Mar. 2007.

[33] M. Lettrari and J. Klose, "Scenario-Based Monitoring and Testing of Real-Time UML Models," *Proc. Fourth Int'l Conf. Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML '01),* pp. 317-328, 2001.

[34] M.A. Ameedeen and B. Bordbar, "A Model Driven Approach to Represent Sequence Diagrams as Free Choice Petri Nets," *Proc. 12th Int'l IEEE Enterprise Distributed Object Computing Conf. (EDOC '08),* pp. 213-221, 2008.

[35] Ø. Haugen, K.E. Husa, R.K. Runde, and K. Stølen, "STAIRS: Towards Formal Design with Sequence Diagrams," *J. Software and System Modeling,* vol. 4, pp. 355-357, 2005.

[36] S. Maoz and D. Harel, "From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ," *Proc. SIGSOFT Conf. Foundations of Software Eng. (FSE '06),* pp. 219-230, 2006.

[37] F. Chen and G. Rosu, "MOP: An Efficient and Generic Runtime Verification Framework," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '07),* pp. 569-588, 2007.

[38] D. Harel and R. Marelly, *Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.

[39] T. Bultan, "Modeling Interactions of Web Software," *Proc. Second Int'l Workshop Automated Specification and Verification of Web Systems (WWV '06),* pp. 45-52, 2006.

[40] A. Lazovik, M. Aiello, and M.P. Papazoglou, "Associating Assertions with Business Processes and Monitoring Their Execution," *Proc. Second Int'l Conf. Service Oriented Computing (ICSOC '04),* pp. 94-104, Nov. 2004.

[41] L. Baresi, C. Ghezzi, and S. Guinea, "Smart Monitors for Composed Services," *Proc. Second Int'l Conf. Service Oriented Computing (ICSOC '04),* pp. 193-202, Nov. 2004.

[42] L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes," *Proc. Third Int'l Conf. Service Oriented Computing (ICSOC '05),* pp. 269-282, 2005.

[43] M. Pistore and P. Traverso, "Assumption-Based Composition and Monitoring of Web Services," *Test and Analysis of Web Services,* pp. 307-335, Springer, 2007.

[44] M. Lohmann, L. Mariani, and R. Heckel, "A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services," *Test and Analysis of Web Services,* pp. 173-204, Springer, 2007.

[45] Z. Li, Y. Jin, and J. Han, "A Runtime Monitoring and Validation Framework for Web Service Interactions," *Proc. 17th Australian Software Eng. Conf. (ASWEC '06),* pp. 70-79, 2006.

[46] K. Mahbub and G. Spanoudakis, "A Framework for Requirements Monitoring of Service Based Systems," *Proc. Second Int'l Conf. Service Oriented Computing (ICSOC '04),* pp. 84-93, 2004.

[47] K. Mahbub and G. Spanoudakis, "Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience," *Proc. Int'l Conf. Web Services (ICWS '05),* pp. 257-265, July 2005.

[48] W.M.P. van der Aalst and M. Pesic, "Specifying and Monitoring Service Flows: Making Web Services Process-Aware," *Test and Analysis of Web Services,* pp. 11-55, Springer, 2007.

[49] Z. Li, J. Han, and Y. Jin, "Pattern-Based Specification and Validation of Web Services Interaction Properties," *Proc. Third Int'l Conf. Service Oriented Computing (ICSOC '05),* pp. 73-86, 2005.

[50] M. Shanahan, "The Event Calculus Explained," *Artificial Intelligence Today,* pp. 409-430, Springer, 1999.

[51] J.A. Baier and S.A. McIlraith, "Planning with First-Order Temporally Extended Goals Using Heuristic Search," *Proc. 21st Nat'l Conf. Artificial Intelligence (AAAI '06) and the 18th Innovative Applications of Artificial Intelligence Conf. (IAAI '06),* July 2006.

[52] L. Baresi, S. Guinea, and L. Pasquale, "Self-Healing BPEL Processes with Dynamo and the JBoss Rule Engine," *Proc. Int'l Workshop Eng. of Software Services for Pervasive Environments (ESSPE '07),* pp. 11-20, 2007.
[53] O. Moser, F. Rosenberg, and S. Dustdar, "Non-Intrusive Monitoring and Service Adaptation for WS-BPEL," *Proc. 17th Int'l Conf. World Wide Web (WWW '08),* pp. 815-824, 2008.
[54] M. Reichert and P. Dadam, "ADEPTflex: Supporting Dynamic Changes of Workflows without Losing Control," *J. Intelligent Information Systems,* vol. 10, no. 2, pp. 93-129, 1998.
[55] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *SIGSOFT Software Eng. Notes,* vol. 27, no. 6, pp. 1-10, 2002.
[56] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error Explanation with Distance Metrics," *Int'l J. Software Tools for Technology Transfer,* vol. 8, no. 3, pp. 229-247, 2006.
[57] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili, "Synthesis of Correct and Distributed Adaptors for Component-Based Systems: An Automatic Approach," *Proc. 20th Int'l Conf. Automated Software Eng. (ASE '05),* pp. 405-409, 2005.

Jocelyn Simmonds received the BSc degree from the University of Chile in 2003 and the MSc degree from the Free University of Brussels in 2003. She also received a degree in computer engineering from the University of Chile in 2005. She is currently working toward the PhD degree under the supervision of Marsha Chechik. Her research interests are in software testing and automated verification.

Yuan Gan received the MSc degree from the Department of Computer Science at the University of Toronto in April 2007. She is currently a software developer at the IBM Toronto Lab, where she works on development of WebSphere BPM tools.

Marsha Chechik received the PhD degree from the University of Maryland in 1996. She is currently a professor in the Department of Computer Science at the University of Toronto. Her research interests are in the application of formal methods to improve the quality of software. She has authored more than 60 papers in formal methods, software specification and verification, computer security, and requirements engineering. In 2002-2003, she was a visiting scientist at Lucent Technologies in Murray Hill, New York, and Imperial College, London, United Kingdom. In 2003-2007, she was an associate editor of the *IEEE Transactions on Software Engineering*. She is a member of the IFIP WG 2.9 on Requirements Engineering, and regularly serves on program committees of international conferences in the areas of software engineering and automated verification. She was a cochair of the 2008 International Conference on Concurrency Theory (CONCUR), a program committee cochair of the 2008 International Conference on Computer Science and Software Engineering (CASCON), and a program committee cochair of the 2009 International Conference on Formal Aspects of Software Engineering (FASE). She is a member of the IEEE Computer Society

Shiva Nejati received the BSc degree from the Sharif University of Technology, Iran, in 2000, and the MSc and PhD degrees from the University of Toronto in 2003 and 2008, respectively. She is currently a research scientist at the Simula Research Laboratory in Norway. Her main research area is software engineering, with specific interests in model-based development, behavior analysis, requirements engineering, specification and design methods, and Web services.

Bill O'Farrell received the PhD degree in parallel computing from Syracuse University. He is a senior technical advisor in the area of business process management. He has been at IBM for 18 years, and has worked in a number of areas, including two positions (research associate and manager) within the Center for Advanced Studies. Besides BPM, his research interests include debuggers, concurrency, and object-oriented design.

Elena Litani received the bachelor's degree in computer science from York University, Toronto, Canada. She is an advisory software developer at the IBM Toronto Lab. She has been at IBM for seven years, working as a software developer and development manager on different projects, including Eclipse Modeling Framework (EMF) and Apache Xerces2 open source projects. She is currently a member of the Center for Advanced Studies at IBM.

Julie Waterhouse is an advisory software developer with 16 years of combined experience in software development and consulting with the IBM Toronto Lab. She is currently a member of the WebSphere Integration Developer SWAT Team, where she works with customers to help them be successful in building SOA-based integration solutions across WebSphere Process Server, WebSphere Enterprise Service Bus, and WebSphere Adapters.