

# Runtime Monitoring of Web Service Conversations

Yuan Gan, Marsha Chechik, Shiva Nejati  
Department of Computer Science, University of Toronto  
{ganyuan, chechik, shiva}@cs.toronto.edu

Jon Bennett, Bill O’Farrell, Julie Waterhouse  
IBM Toronto Lab  
{jdbennet, billo, juliew}@ca.ibm.com

## Abstract

For a system of distributed processes, correctness can be ensured by (statically) checking whether their composition satisfies properties of interest. In contrast, Web services are being designed so that each partner discovers properties of others dynamically, through a published interface. 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 behavioural 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 describe an implementation of our approach as part of an industrial system and report on preliminary experience.

## 1 Introduction

Recent years have seen an emergence of the field of web services, which use Service-Oriented Architectures (SOA) to dynamically discover and bind to services in order to increase the flexibility of business interactions. Each service consist of *components* and can discover other components using

© Copyright Yuan Gan, Marsha Chechik, Shiva Nejati, and IBM Canada Ltd., 2007. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

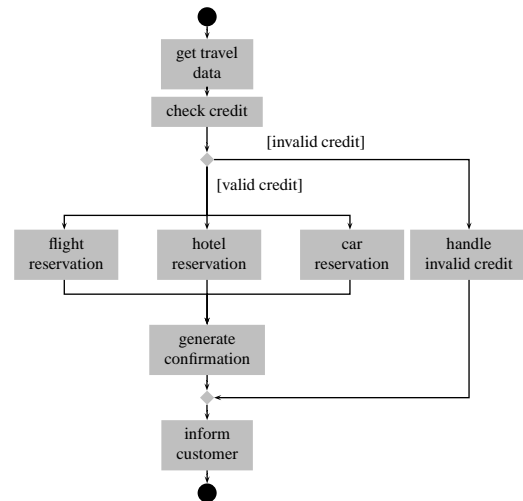


Figure 1: The Activity Diagram of TBS.

published interfaces. An SOA component can be written in a traditional compiled language such as Java<sup>TM</sup>, or in an XML-centric language such as BPEL [6]. An SOA *module* is made up of multiple SOA components.

For example, consider a Web-based travel booking system (TBS) that acts as a broker offering its customers the ability to book all aspects of a trip. The workflow of TBS 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 Figure 1 shows high-level steps that are executed during the travel book-



Figure 2: Interaction between Web Services.

ing process. To fulfill its business goal, TBS needs to interact with several partners: *CreditCardCheckingService*, which validates the customer’s credit card data, *FlightReservationService*, which books a flight, *HotelReservationService*, which reserves a hotel room, and *CarReservationService*, which makes a car reservation. In a typical scenario, an Internet customer begins an interaction with TBS by entering data for his/her travel arrangements. The system then invokes *CreditCardCheckingService*, and if the credit card is valid, then up to three reservations, for the car, hotel and flight, are made. If all of the reservations are completed successfully, a confirmation number is generated and returned to the customer.

Since the TBS system, like other web services, is a composition of several distributed business processes, its success depends on the correctness of its partners, and the interoperability between them is a major quality concern. For example, the system needs to guarantee that it processes travel reservations only for customers with valid credit, or that every request is acknowledged and none are lost or blocked indefinitely. 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., [12, 13, 26, 4, 11]. While static analy-

sis is very appealing – errors are discovered ahead of time and without the need to exercise the system, this approach has several major limitations. First of all, web services typically communicate via infinite-length channels, so the problem is decidable only under certain conditions. Further, realistic web services exchange many types of messages: some synchronous, some asynchronous, and some with acknowledgements and priorities. Finally, web services are typically heterogeneous. In our example, the TBS process is implemented using BPEL while other partners are written in Java and are invoked synchronously (see Figure 2). Static analysis approaches do not handle such features well.

Since web services are designed to discover properties of other partners dynamically (and thus the overall system may not even be available statically), we instead choose dynamic analysis via runtime monitoring. This approach has been explored by others as well [5, 32, 30]. Yet the goal of our work is to create a monitoring framework that is non-intrusive, allows the dynamic discovery of web services, and supports a variety of types of message exchange and partners implemented in different languages. In this paper, we describe the experience of implementing such a system, concentrating on a specification language for dynamic analysis of web services.

We aim to create an industrial-strength language for specifying temporal behaviour 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. We believe that such a language should have the following characteristics: (1) its notation should be *visual*; (2) it should allow specification of sequences of events; (3) it should have 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 graphical languages, such as GIL [9], Time Line Editor [34], Message Sequence Charts (MSCs) [24] and Live Sequence Charts (LSCs) [7], we have chosen a subset of UML 2.0 Sequence Diagrams (SDs) [36] as our specification language. SDs, used to capture in-

$P_1$	TBS should not reserve hotel room without checking customer credit first.
$P_2$	A customer should be eventually notified about the status of his/her travel booking request, whether the reservation succeeds or not.

Table 1: Some properties of TBS.

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 requirements specifications.

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* (nothing bad will ever happen) and *liveness* (something good will eventually happen) properties. For example, for the TBS system described earlier, possible safety and liveness properties are  $P_1$  and  $P_2$ , respectively (see Table 1). Liveness properties are not monitorable in general. However, since our work concentrates on web services with finitely terminating behaviours, we can monitor such properties in our framework, e.g., we can check whether the TBS process terminates without giving feedback to the customer.

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., [2, 16, 15]. 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.

The rest of this paper is organized as follows. After surveying related work in Section 2 and reviewing SDs in Section 3, we describe the semantics of our chosen subset of SDs in Section 4. In Section 5, we show how to selectively apply SD operations *negate* and *assert* in order to specify safety and liveness properties. We give semantics of the resulting subset of SDs using Safety and Liveness automata. We describe the implementation of this framework and report on preliminary experience in Section 6, and conclude the paper in Section 7 with a summary and an outline of future research.

## 2 Related Work

Monitoring systems at run time to ensure correctness has received a lot of attention, and many such systems have been developed. In this section, we

survey the research of runtime monitoring in the context of web services. We also summarize some work studying UML 2.0 Sequence Diagrams as a specification language.

**Runtime monitoring of web services.** Existing approaches to monitoring of web services differ in a number of aspects, including the specification language for expressing monitoring requirements/properties, the monitoring mechanisms of determining faults in service execution, the level of intrusiveness imposed on the monitored system, the system aspects being monitored, and the method of reporting results.

The approaches described in [5, 27] use instrumentation to check that web services correctly implement their interfaces. These approaches require access to the source code of web services. Other approaches [30, 29, 32], like ours, are complementary to [5, 27] and instead monitor correctness of conversations between partners. However, they use different specification languages and monitor different kinds of properties. Both Mahbub and Spanoudakis [30], and Robinson [32] concentrate on checking invariant and request/response properties, using, respectively, Event Calculus [33] and KAOS [8] for expressing properties. In our work, we translate properties to finite automata and can handle a variety of temporal properties checkable on finite traces. The work of Li et al. [29] is the closest to ours. Like us, they take an automata-based approach for monitoring communications between partners and enable graphical display of violations. However, they specify correct interactions using Interaction Constraints [28] – a language based on Dwyer’s Specification Pattern System [10]. Our specification language, SD, is expressive enough for a variety of safety and liveness properties and yet significantly more intuitive and thus more usable in an industrial context.

**Sequence Diagrams as a Specification Language.** Like other partial-order scenario-based formalisms such as MSCs [24] and LSCs [7], UML 2.0 Sequence Diagrams are enjoying an increasing usage as specification languages. Inverardi et al. [3] propose a Property Sequence Chart (PSC) language, which is an extended notation of a subset of UML 2.0 SDs. PSC enables expressing safety

and liveness properties by assigning attributes *fail* and *required* to messages. This is equivalent to applying operators *negate* and *assert* to individual SD message, respectively. The semantics of PSC is given using Büchi Automata, designed to operate on infinite execution traces. Since we consider only finite executions of web services, automata over finite words are sufficient and significantly easier to implement.

STAIRS [17] 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 properties without extending the language and also explore connections between our language and temporal logic.

Grosu and Smolka [15] 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.

### 3 Sequence Diagrams

*Sequence Diagrams* (SDs) [36] is a popular formalism for modeling behavioural scenarios by describing sequences of messages communicated between different objects over time. An example SD describing a scenario of the TBS system is shown in Figure 3(a). SDs have two dimensions: vertical, representing time, and horizontal, representing objects. Each object is illustrated by a rectangle with a vertical dashed line, called a *lifeline*. Lifelines are connected by horizontal arrows denoting messages that are sent from one object to another, synchronously or asynchronously. We refer to SDs with these features as *basic*. Basic SDs can be augmented by a number of operators to capture more sophisticated scenarios. Three classes of these operators included in UML 2.0 are described below:

**Compositional operators:** Operators *parallel* (*par*), *alternatives* (*alt*), *strict sequencing* (*strict seq*), and *weak sequencing* (*weak seq*) are used to put together two SDs via

various standard semantics of composition. The operator *loop* is used for repeating the scenario described by an SD multiple times.

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

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

The grammar for generating composite SDs is obtained by repeated application of compositional and alphabet changing operators, using Basic SDs as the building blocks:

$$\begin{aligned} SD & ::= BasicSD \mid unaryOp \ SD \mid \\ & \quad SD \ binaryOp \ SD \\ unaryOp & ::= consider_E \mid ignore_E \mid loop \\ binaryOp & ::= par \mid alt \mid strict \ seq \mid weak \ seq \end{aligned}$$

where *BasicSD*, *par*, *alt*, *strict seq*, *weak seq*, *loop*, *consider*, and *ignore* are terminal symbols, and *E* is a set of Sequence Diagram messages.

Since operators *consider* and *ignore* change the communicating alphabet of Sequence Diagrams, they need to take a set *E* of messages as an input argument. In what follows, we denote by SDs Sequence Diagrams generated by the above grammar. We discuss the *negate* and *assert* operators in Section 5.

### 4 Formalizing Sequence Diagrams

In this section, we provide a formal description of semantics of Basic SDs as well as their compositional and alphabet changing operators by adopting the automata-theoretic approach of [2].

**Nondeterministic Finite Automata.** Let  $\Sigma$  be a set of alphabets. We define a *trace*  $\sigma$  over  $\Sigma$  to be a finite sequence  $\sigma_0\sigma_1 \dots \sigma_n$ , where  $\forall i \cdot 0 \leq i \leq n, \sigma_i \in \Sigma$ . We denote by  $\Sigma^*$  the set of all finite traces over  $\Sigma$ .

**Definition 1 (Projection “ $\downarrow$ ”)** Let  $\Sigma' \subseteq \Sigma$  be an alphabet, and  $\sigma = \sigma_0 \dots \sigma_n$  be a trace over  $\Sigma$ . The projection of  $\sigma$  to  $\Sigma'$ , denoted  $\sigma \downarrow_{\Sigma'}$ , is defined as:

$$\sigma \downarrow_{\Sigma'} = (\sigma_0 \downarrow_{\Sigma'}) (\sigma_1 \downarrow_{\Sigma'}) \dots (\sigma_n \downarrow_{\Sigma'})$$

where  $\sigma_i \downarrow_{\Sigma'} = \sigma_i$  if  $\sigma_i \in \Sigma'$ , and  $\epsilon$  otherwise.

**Definition 2 (NFA [18])** A Non-deterministic Finite Automaton (NFA)  $A$  is a tuple  $(\Sigma, Q, \delta, Q_0, F)$ , where  $\Sigma$  is a set of input alphabets,  $Q$  is a finite set of states,  $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$  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  $\sigma = \sigma_0\sigma_1\dots\sigma_n$  is accepted by  $A$  iff there is a sequence  $q_0q_1\dots q_{n+1}$  of states s.t.  $q_0 \in Q_0$ ,  $q_{n+1} \in F$ , and for every  $0 \leq i \leq n$ ,  $(q_i, \sigma_i, q_{i+1}) \in \delta$ . The language of  $A$ ,  $L(A)$ , is the set of all traces accepted by  $A$ .

An NFA  $A$  receives a trace  $\sigma \in \Sigma^*$  as input and changes its current state according to its transition relation. A trace  $\sigma$  is considered *accepting* if after consuming it,  $A$  is in an accepting state. An example NFA over the alphabet  $\{!rH, ?rH, !rF, ?rF\}$  is shown in Figure 3(b). In cases where states do not have outgoing transitions for some symbols in  $\Sigma$ , like state  $q_1$  on  $?rF$  in Figure 3(b), it is assumed that this symbol causes a transition to a (non-accepting) dead-end state, which is usually not shown.

States in NFAs may have several outgoing transitions on the same input symbol, or may have transitions labeled  $\epsilon$ , indicating a *silent* move. *Deterministic* finite automata (DFAs) are NFAs where each state has a unique outgoing transition on each symbol. Every NFA can be converted into a DFA using the subset construction algorithm [18].

**Basic SDs.** We define basic SDs as follows.

**Definition 3 (Basic SDs [2])** 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 associated with 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  $i$ .

- $<$  is a partial order relation defined over  $E$ :  
 $< = \cup_{i \in \mathcal{I}} <_i \cup \{(s, f(s)) \mid s \in !E\}$ .

A Basic SD is shown in Figure 3(a). Here, the set  $\mathcal{I}$  of objects is  $\{Agt, Htl, Ftl\}$ , the set  $E$  of events is  $\{!rH, ?rH, !rF, ?rF\}$ , the total order  $<_{Agt}$  for object  $Agt$  is  $!rH <_{Agt} !rF$ , and the partial order  $<$  associated with the whole diagram is  $!rH < !rF$ ,  $!rH < ?rH$ , and  $!rF < ?rF$ . This partial order assumes that messages are communicated asynchronously. Partial order for synchronous communication is a subset of the above because of synchronization.

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 by [2], is as follows. Given a partial order  $<$  over  $E$ , let *cut*  $c$  be a subset of  $E$  that is closed with respect to  $<$ , i.e., if  $e \in c$  and  $e' < e$ , then  $e' \in c$ . 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. If a cut  $d$  equals the cut  $c$  plus a single event  $e$ , then there is a transition labeled  $e$  from  $c$  to  $d$ .

**Theorem 1** [2] 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  $\Sigma$  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  $\delta$  allows a transition from a cut  $d$  to a cut  $c$  on an event  $e \in E$  iff  $d$  is equal to  $c$  plus a single event  $e$ .

Since both the empty and the 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 Figure 3(a) is  $\{\emptyset, \{!rH\}, \{!rH, ?rH\}, \{!rH, !rF\}, \{!rH, ?rH, !rF\}, \{!rH, !rF, ?rF\}, \{!rH, !rF, ?rH\}, \{!rH, ?rH, !rF, ?rF\}\}$ . Note that the number of states of the corresponding automaton in Figure 3(b) is less than the number of the above cuts, because we reduced the states with the identical outgoing transitions to one state.

**Compositional operators.** The semantics of the compositional operators can be given in terms of

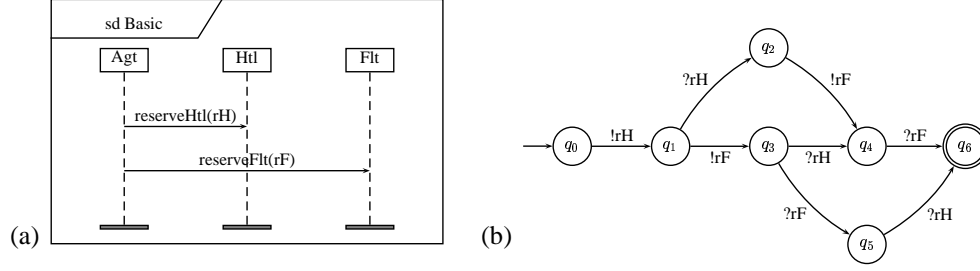


Figure 3: (a) a basic SD describing a scenario of the TBS example; and (b) its corresponding NFA.

the standard operations defined on NFAs (e.g., see [18]). In particular,

- *par* corresponds to the parallel composition operator;
- *alt* corresponds to the union operator;
- *strict seq* corresponds to the sequential composition operator;
- *weak seq* is a combination of parallel and sequential composition in which events associated with one particular object appear in the order indicated by the lifeline of this object, and other events are interleaved according to the semantics of parallel composition; and
- *loop* corresponds to the Kleene star operator.

The theorem below shows that the set of NFAs associated with SDs is closed under the compositional operators.

**Theorem 2** [18] *Let  $S, S_1$  and  $S_2$  be SDs, and let  $S = S_1 \text{ op } S_2$ , where  $\text{op}$  is a compositional operator. Then  $A_S = A_{S_1} \text{ op } A_{S_2}$ .*

**Alphabet changing operators.** Operators *consider* and its dual *ignore* are used to change the set of communicating alphabets of an SD. Both of them receive an SD  $S$  and a set of events  $E$  as input, but *consider* adds the elements in  $E$  to the set of events of  $S$ , whereas *ignore* removes the elements in  $E$  from the set of events of  $S$ . Formally, let  $S$  and  $S'$  be SDs,  $E$  be a set of events, and let  $A_S = (\Sigma, Q, \delta, q_0, F)$  be the automaton associated with  $S$ . For  $S' = \text{consider}_E S$ ,  $A_{S'} = (\Sigma \cup E, Q, \delta, q_0, F)$ , and for  $S' = \text{ignore}_E S$ ,  $A_{S'} = (\Sigma \setminus E, Q, \delta, q_0, F)$ . 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  $\Sigma$  of  $A_S$  without changing the transition relation  $\delta$

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.

## 5 Monitoring Properties

While quite expressive, the fragment of SDs presented in Section 4 cannot capture safety properties, e.g.,  $P_1$  in Table 1, because it does not have a mechanism for specifying undesirable scenarios. Neither can it be used for liveness properties, e.g.,  $P_2$ , because it cannot specify that a desirable scenario is mandatory for every behaviour of the system. In this section, we extend the SD fragment to enable expressing such properties, using *negate* and *assert* operators.

Various formal treatments of the semantics of these operators are given in the literature, e.g., [16, 15, 35]. These operators have a rich expressive power, and yet their arbitrary combinations are not well understood (e.g., does negating of an asserted trace mean that this trace is not required to occur or that a negation of the trace has to occur?), which is perhaps the reason why they have not found significant use in practice. We propose a simple fragment of SDs in which these operators can be used only once and cannot be intermixed. We show that this fragment is expressive enough to describe most safety and liveness properties. The simplicity of this fragment further reduces the complexity of our monitoring framework from both a conceptual and a computational perspective.

To describe a safety property, we enclose an SD  $S$  within a *negate* operator to indicate that the scenario represented by  $S$  is a forbidden one, and therefore, a safe system should never produce this scenario [15]. We call the resulting SDs *Safe*. For example, the property  $P_1$  in Table 1 is captured by a Safe SD in Figure 4(a). Similarly, we describe a liveness property by enclosing an SD  $S$  within an

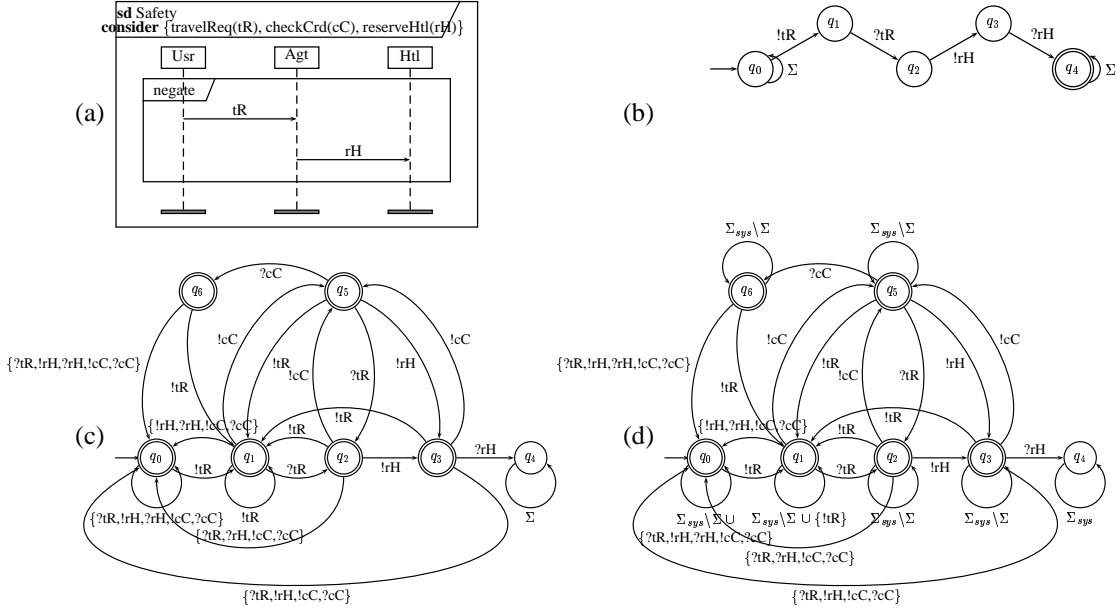


Figure 4: (a) A Safe SD describing the TBS property  $P_1$  and its corresponding NFAs: (b) before applying *negate*; (c) safety automaton after applying *negate*; (d) the resulting safety monitor.

*assert* operator to indicate that the scenario represented by  $S$  is the only valid continuation of any system behaviour [36]. We call such SDs *Live*. For example, the property  $P_2$  in Table 1 is expressed by a Live SD in Figure 5(a).

The grammar of SDs introduced in Section 3 is extended with two new rules:

$$\begin{aligned} \text{SafeSD} & ::= \text{negate SD} \\ \text{LiveSD} & ::= \text{assert SD} \end{aligned}$$

We discuss the semantics of Safe and Live SDs in Sections 5.1 and 5.2, respectively.

## 5.1 Monitoring Safety Properties

In this section, we show that Safe SDs can be translated into a particular class of NFAs called *Safety Automata* [1].

**Definition 4 (Safety Automaton [1])** An NFA  $A$  is a Safety Automaton iff every state of  $A$  is accepting.

For example, the automaton shown in Figure 4(c) is a safety automaton, since the state  $q_4$  is dead-end and thus can be removed. This automaton can detect all sequences satisfying the TBS safety property  $P_1$ . For example, it accepts the safe trace  $!tR.?tR.!cC.?cC$ .

$!rH.?rH$  (“a travel request is followed by a check credit card, followed by a hotel reservation”), and rejects the unsafe trace  $!tR.?tR.!rH.?rH$  (“a travel request is immediately followed by a hotel reservation”). Note that the *consider* operator used in Figure 4(a) changes the underlying alphabet of this SD to  $\{tR,cC,rH\}$ . Thus, symbols  $!cC$  and  $?cC$  are included in the alphabet of the automata in Figures 4(b)-(c) even though they do not explicitly appear in the SD.

We begin by making a slight modification to the NFA derived from an SD  $S$ , adding a self-loop transition labeled  $\Sigma$ , i.e., the underlying alphabet of this automaton, to its initial state in order to enable it to *guess* when a satisfying run begins. For example, Figure 4(b) illustrates the automaton corresponding to the Safe SD in Figure 4(a) before applying the *negate* operator. This automaton has a  $\Sigma$ -labeled self-loop in its initial state and thus accepts all traces containing the sequence  $!tR.?tR.!rH.?rH$  and starting with any arbitrary prefix in  $\Sigma^*$ .

We are now ready to describe our translation of a Safe SD to a safety automaton. Let  $A_S$  be an NFA associated to an SD  $S$  and obtained by the above construction, and let  $S^{safe}$  be a Safe SD obtained by enclosing  $S$  with a *negate*. The automaton corresponding to  $S^{safe}$  is the complement of  $A_S$  [15]

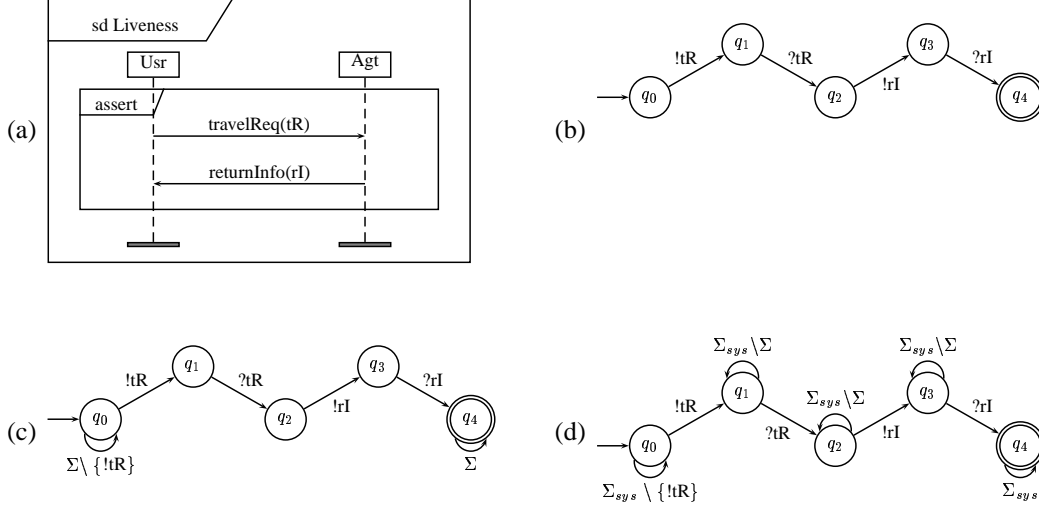


Figure 5: (a) A Live SD describing the TBS property  $P_2$ , and its corresponding NFAs: (b) before applying *assert*; (c) liveness automaton after applying *assert*; (d) the resulting liveness monitor.

and is denoted by  $A_S^{safe}$ . Note that  $A_S$  is an NFA, and hence it needs to be converted to a DFA before being complemented.

**Theorem 3** Let  $A_S = (\Sigma, Q, \delta, Q_0, F)$  be an NFA associated to an SD  $S$ , and let  $A_S^{safe}$  be the complement of  $A_S$ . Then  $A_S^{safe}$  is a safety automaton.

The proof of this theorem follows from noting that  $A_S$  has exactly one accepting state (see Theorem 1) and is available in [14].

The underlying alphabet of  $A_S^{safe}$  is equal to that of  $S$ . To be able to use  $A_S^{safe}$  for runtime monitoring, we need to extend the language of  $A_S^{safe}$  to handle system traces over alphabets larger than  $S$ . We do so by adding stuttering self-loops to the automaton's states. Semantically, this means that  $A_S^{safe}$  does not change its state when the input symbol is outside the alphabet of  $S$ .

**Definition 5 (Stuttering)** Let  $\Sigma_{sys}$  be the set of system events, and let  $A = (\Sigma, Q, \delta, Q_0, F)$  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.  $\Sigma_{sys}$ , where  $\delta' = \delta \cup \{(q, \Sigma_{sys} \setminus \Sigma, q) \mid \forall q \in Q\}$ .

The transformation of Definition 5 is language-preserving:

**Theorem 4** Let  $A = (\Sigma, Q, \delta, Q_0, F)$  be an NFA, and let  $\Sigma_{sys}$  s.t.  $\Sigma \subseteq \Sigma_{sys}$  be given. Let  $A'$  be

the stutter-closed form of  $A$  w.r.t.  $\Sigma_{sys}$  (see Definition 5). Then for every trace  $\sigma \in \Sigma_{sys}$ ,  $\sigma \in L(A')$  iff  $\sigma \downarrow_{\Sigma} \in L(A)$ .

We refer to the stutter-closed form of  $A_S^{safe}$  as a *Safety Monitor* and denote it by  $Monitor(A_S^{safe})$ . Since  $A_S^{safe}$  is a safety automaton, and further since the transformation in Definition 5 is language-preserving,  $Monitor(A_S^{safe})$  is also a safety automaton. For example, the safety monitor corresponding to the Safe SD in Figure 4(a) is shown in Figure 4(d). This monitor rejects any finite trace in “ $(\Sigma_{sys}/\{!tR\})^* !tR.(\Sigma_{sys}/\Sigma)^* ?tR.(\Sigma_{sys}/\Sigma)^* !rH.(\Sigma_{sys}/\Sigma)^* ?rH$ ”.

## 5.2 Monitoring Liveness Properties

Liveness properties constrain infinite traces of systems, and their standard characterization is given in terms of infinite automata [1]. In our work, however, we monitor web services that generate only finite traces. In this setting, a finite trace satisfies a liveness property if it can completely exhibit the liveness behaviour before being terminated. Using this insight, we define a notion of (*finitary*) *liveness automata* to characterize liveness properties over finite traces. We then show that Live SDs can be translated into this class of automata.

**Definition 6 (Liveness Automata)** Let  $\Phi_{live}$  be a set of finite traces  $\sigma_{live} = \sigma_0 \sigma_1 \dots \sigma_n$  represent-

ing liveness properties, and let  $\Sigma_{live}$  be the union of the underlying alphabets of traces  $\sigma_{live} \in \Phi_{live}$ . An NFA  $A = (\Sigma, Q, \delta, Q_0, F)$  is a liveness automaton w.r.t.  $\Phi_{live}$  iff for every  $\sigma \in L(A)$ , and every  $\sigma_{live} \in \Phi_{live}$ ,  $\sigma \downarrow_{\Sigma_{live}}$  includes  $\sigma_{live}$ .

Intuitively,  $A$  is a liveness automaton, if every trace recognized by it includes the live part completely.

**Definition 7** Let  $A_S = (\Sigma, Q, \delta, Q_0, F)$  be an NFA associated with an SD  $S$ , and let  $S^{live}$  be a Live SD obtained by enclosing  $S$  with an *assert* operator. The automaton corresponding to  $S^{live}$  is  $A_S^{live} = (\Sigma, Q, \delta', Q_0, F)$ , where

$$\begin{aligned} \delta' = & \{(q_0, \Sigma \setminus \sigma_0, q_0) \mid \exists \sigma \in L(A_S). \\ & \sigma_0 \text{ is the first symbol of } \sigma\} \cup \\ & \{(q_F, \Sigma, q_F) \mid F = \{q_F\}\} \cup \delta \end{aligned}$$

**Theorem 5** Let  $A_S = (\Sigma, Q, \delta, Q_0, F)$  be an NFA associated with an SD  $S$ , and let  $A_S^{live}$  be an automaton obtained by the translation in Definition 7. Then  $A_S^{live}$  is a liveness automaton w.r.t.  $L(A_S)$ .

Figure 5(b) shows the NFA corresponding to the Live SD in Figure 5(a) before applying the *assert* operator, and Figure 5(c) shows the corresponding liveness automaton. As shown in the figure, this automaton requires that the trace  $!tR?rI.!rI.?rI$  occur at least once.

Like Safe SDs discussed in Section 5.1, an automaton  $A_S^{live}$  is not immediately applicable to our runtime monitoring framework because its underlying alphabet does not include all system events. We modify  $A_S^{live}$  using the transformation in Definition 5 to obtain its stutter-closed counterpart, referring to it as *Live Monitor* and denoting it by  $Monitor(A_S^{live})$ . Since the transformation in Definition 5 is language-preserving (see Theorem 4),  $Monitor(A_S^{live})$  remains to be a liveness automaton. The liveness monitor corresponding to the Live SD in Figure 5(a) is shown in Figure 5(d). This monitor rejects any trace in which a travel request is not followed by a return information event. Formally, this automaton rejects the language “ $\Sigma_{sys}^* \cdot [\epsilon \mid !tR. (\Sigma_{sys}/\Sigma)^* \mid !tR. (\Sigma_{sys}/\Sigma)^* \cdot ?tR. (\Sigma_{sys}/\Sigma)^* \mid !tR. (\Sigma_{sys}/\Sigma)^* \cdot ?tR. (\Sigma_{sys}/\Sigma)^* \cdot !rI. (\Sigma_{sys}/\Sigma)^*]$ ”.

Note that *assert* is a universal operator: it requires a given sequence to occur in all executions of a system [16]. Without *assert*, the sequence is allowed to occur on some traces. Our monitors,

on the other hand, are just word automata that are unable to differentiate between universal and existential acceptance. Rather than using more complex monitors, we give semantics to the notion of acceptance: a liveness monitor checks all system traces, and if one is not accepted, a failure is reported. Monitors based on sequences without *assert* yield success if a given trace satisfies the sequence, but can never yield failure.

## 6 Architecture and Implementation

We have implemented our runtime framework within the IBM® WebSphere® business integration products [19]. In what follows, we describe the architecture of our solution and discuss some of the more challenging parts of the implementation. We also report on preliminary experience of using this framework to check correctness of web services. For more information, see [14].

### 6.1 Architecture

Our solution uses the WebSphere Process Server [21] and the WebSphere Integration Developer [20]. 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.

Our framework is shown in Figure 6. With the help of the *Property Manager* (PM), users create UML SD specifications for their web service applications. If monitoring is enabled, the *Monitoring Manager* (MonM) translates them into monitor automata using the techniques in Section 5. During the execution of the web service, *Message Manager* (MM) obtains interaction events from the *SCA Message Handler* (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. We describe these components below.

The Property Manager consists of a graphical tool for specifying interaction properties as UML SDs. Once users create a diagram and enable monitoring, PM loads the XML model of the specification diagram, checks that it uses the language

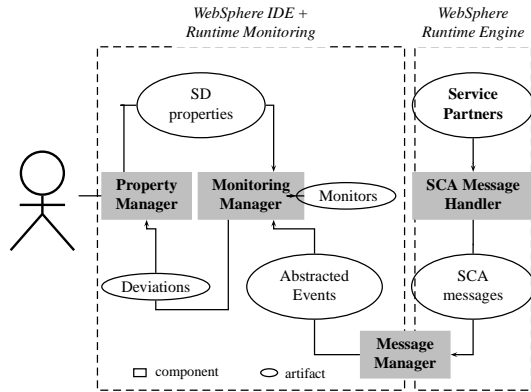


Figure 6: Architecture of the Framework.

subset described in Section 4, unwinds the partial order of the diagram into an NFA using the algorithm introduced in Section 4, and passes the NFA to MonM. In the case of a property failure, PM is also responsible for displaying errors to the user.

The SCA Message Handler sits in the process server and establishes a bridge through which our runtime monitoring framework communicates with the server to obtain information about the web service execution. In the process server, SCA is responsible for the invocation of native SCA service components and for the binding and interaction with external services. The SCA message handler 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.

The Message Manager is responsible for obtaining service request/response messages exchanged between business components from the SCA layer. MM, registered as a listener to the SCA message handler, 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 this information together with the timestamp of when the events were intercepted, and sends them to the message queue associated with MonM via a TCP/IP communication channel.

The Monitoring Manager is the central part of the framework, dealing with constructing monitor automata, processing events and keeping 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 4 and 5. 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 corresponds to taking the silent transition. 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.

## 6.2 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 design described in Section 6.1, we implemented four plug-ins. Figure 7 depicts the interactions and dependences between these, using double-headed lines.

The *MonitoringCorePlugin* is the component corresponding to the *MonitoringManager* in the architecture. It consists of a *MonitorCore* package, which acts as an entry point to *MonitoringCorePlugin*, a *Monitor* package that is responsible for receiving unfolded SD specifications and translating them into monitor automata, an *EventListener* package, which handles getting events from *MessageManagerPlugin* and forwards events to relevant monitors, and a *Utilities* package where automata-related manipulation is provided.

The *MessageManagerPlugin* implements the *Message Manager* in the architecture. It contains two packages: *EventAdaptor* and *EventForwarder*. The *EventAdaptor* package registers itself as a listener to the SCA Message Handler 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 *EventFor-*

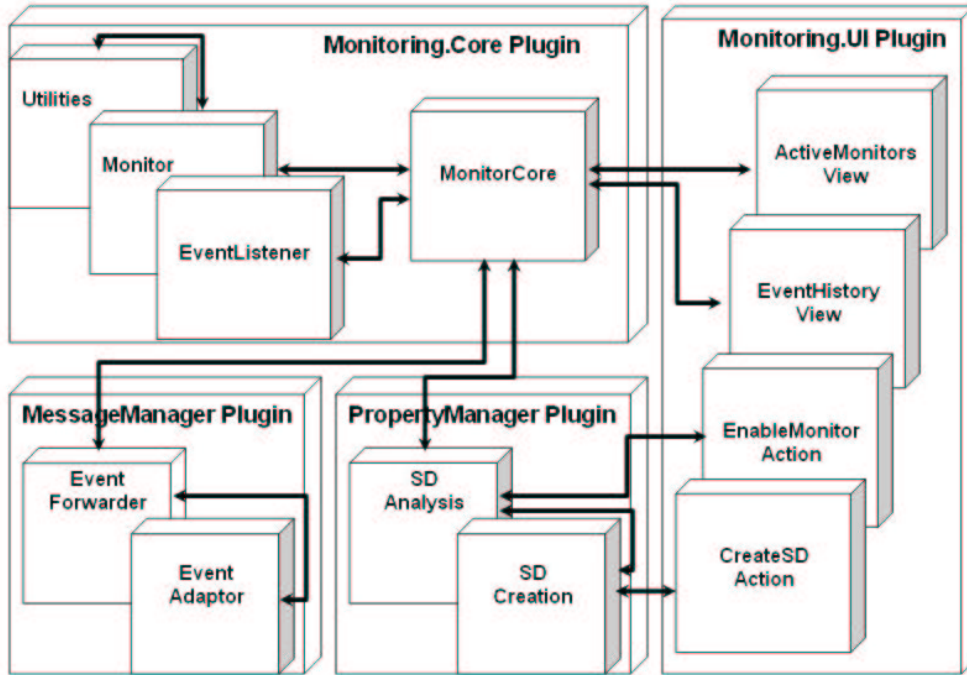


Figure 7: The Overview of Framework Plug-ins.

warder 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 the *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.

The *PropertyManagerPlugin* corresponds to the *Property Manager* 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 translations of Safe SDs, Live SDs, and all operations introduced in Section 3 except weak sequencing. In our framework, we adopted the implementation of compositional operations over automata from the Charmy project [23].

The *MonitoringUIPlugin* 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 a Sequence Diagram specification and then to enable it for monitoring. *ActiveMonitorsView* and *EventHistoryView* provide user windows to examine the satisfaction of monitored properties and the system execution history. Figure 8 shows the screenshot of the user interface of our runtime monitoring framework. In the Business Integration view in the top-left of the window, it lists the content of the implementation

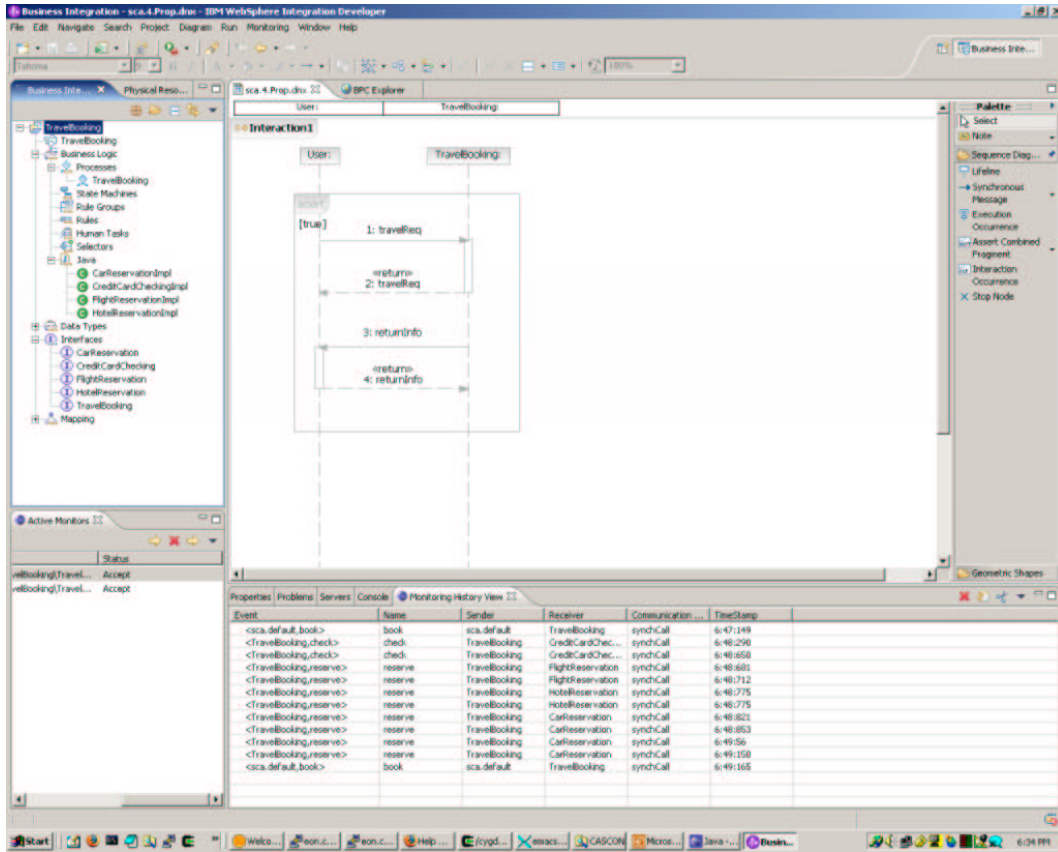


Figure 8: The Screenshot of the Framework User Interface.

Property	Web Service	Type	Involved Partners	Monitored Events	Num of States	Num of Transitions
$P_1$	TBS	Safety	3	6	6	23
$P_2$	TBS	Liveness	2	4	5	10
$P_3$	OSS	Safety	3	8	7	30
$P_4$	OSS	Safety	3	6	6	23
$P_5$	OSS	Liveness	3	4	5	15

Table 2: Properties and Sizes of Their Automata Representations.

of the TBS system. The panel in the middle of the window is the editor for creating Sequence Diagrams and viewing the monitoring results. The bottom two panels are GUIs that belong to the runtime monitoring framework. The left one is the Active Monitors view, which lists all monitor-enabled properties. The view also shows the acceptance status of the monitored properties. The right bottom panel is the Monitor History view from which users can trace the execution of web services.

### 6.3 Other Issues in Implementation

As mentioned in Section 5, in order to construct safety monitors, NFAs should be determinized.

However, the determinization algorithm may result in an exponential blow-up of the number of states. To keep the size of the automata small, we have used several optimization techniques such as reduction and minimization [18], adopting the implementation of these techniques available in [31].

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 [25].

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.

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 associate each event with a timestamp at the time of its invocation in the SCA layer. When events arrive at the message queue of MonM, they are reordered and processed according to their timestamps.

To report the results of monitoring to the users, we display the cause of violations in the SD editor. The violations of safety properties are reported by highlighting the unexpected event in the corresponding SD, and the violation of a liveness property – by marking the termination location indicating an incomplete sequence.

## 6.4 Experience

We have applied our framework to several web services and report on results of monitoring two of them by running our tool on the WebSphere Process Server V6.0 and WebSphere Integration Developer V6.0.1. The first is the TBS system introduced in Section 1 and consisting of five partners and seven invocation-type activities. Table 2 lists sizes of monitoring automata constructed from the TBS properties in Table 1. For example,  $P_1$  includes 6 events between 3 partners and is represented by an automaton with 6 states and 23 transitions.

The second system is an Online Shopping System (OSS) that 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. Two safety properties of OSS,  $P_3$  and  $P_4$  in Table 2, correspond to “*A premium customer always gets a discount on his/her purchase*” and “*An order cannot be billed before being marked complete by the customer*”, respectively. The liveness property of OSS,  $P_5$ , is “*A completed order will be eventually billed*”. Our initial experience indicates that safety and liveness properties can be expressed in our language, and the generated au-

tomata are quite small (e.g., 6 states and 23 transitions for property  $P_4$ ). Obviously, it remains to be seen whether the approach remains feasible for larger web service systems and for more complex properties.

We did not detect errors in running the systems against these properties (although our initial runs discovered errors in properties themselves!), but in order to exercise the monitoring framework, we manually introduced several errors into the web service implementation. For example, we modified the TBS system to remove the fault handler for dealing with invalid credit cards. The monitoring framework was able to detect a violation of the liveness property  $P_2$  when the user submits a travel request with an invalid card, and reported this violation by showing that the event *returnInfo* is missing. We believe that this feedback would have been useful for debugging the TBS implementation. In all cases, the overhead of using the monitoring system was negligible.

## 7 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 non-intrusive, running in parallel with the monitored system and intercepting interaction events during run time. Thus, it does not require any code instrumentation, does not significantly affect the performance of the monitored system, and enables reasoning about partners expressed in different languages. Furthermore, the use of a subset of UML 2.0 SDs as a specification language ensures that the framework is usable by practitioners to specify safety and liveness properties. Liveness becomes finitary, where user-specified time limits or the process termination act as the stopping conditions.

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, two hotels may want to communicate to share overflow customer

requests. We feel that the problem can be easily solved by encoding process IDs into the specification, the automata transition relation, and interaction events. We also plan to handle specification and monitoring of timing constraints and deal with checking correctness of message data being exchanged, although the latter may require a significant modification to the monitoring framework.

Our work so far has been built on a basis that all partners operate within the same process server and thus a centralized monitor is a viable option. In practice, most web services are distributed, requiring a distributed monitoring framework. We plan to investigate techniques used in the DESERT project [22] to turn a centralized monitor into a set of distributed ones, running in different process servers.

While we had little trouble expressing correctness criteria in the two systems we checked, creating *templates* for a variety of properties will significantly improve the usability of our framework and enable effective specification of more complex conversations. To this end, we intend to use the Specification Pattern System (SPS) [10] to build a library of sequence diagram templates. While our subset of SDs is less expressive than temporal logics, we believe that a variety of properties from SPS can be encoded in it. We also intend to conduct further case studies to assess the usability and effectiveness of our framework for checking web service compositions.

## Acknowledgements and Trade-marks

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

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

## About the Authors

**Yuan Gan** received an M.Sc. from the University of Toronto, Department of Computer Science in April 2007. She is currently a software developer at the IBM Toronto Lab, where she works on debugging tools in the WebSphere Business Integration family of products.

**Marsha Chechik** received her Ph.D. from the University of Maryland in 1996 and joined the department of Computer Science at the University of Toronto where she is an associate professor. Prof. Chechik's research interests are in the application of formal methods to improve the quality of software. She has authored over 50 papers in formal methods, software specification and verification, computer security and requirements engineering. In 2002-2003, Prof. Chechik was a visiting scientist at Lucent Technologies in Murray Hill, NY and at Imperial College, London UK. She is an associate editor of IEEE Transactions on Software Engineering, a member of 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.

**Shiva Nejati** received her B.Sc. from Sharif University of Technology in Tehran in 2000 and her M.Sc. from the University of Toronto in 2003. She is currently working toward her Ph.D. Her research interests are in model checking, model management, logic, and automata theory.

**Jon Bennett** is a Sr. Software Engineer at IBM's Toronto Lab. He holds degrees in Economics, Computer Science and Engineering. His time at IBM has been spent working on tools for customers who write enterprise applications. This has resulted in time developing C++ compilers, debuggers for IBM process integration products and, most recently, a focus on customer issues and product quality.

**Bill O'Farrell** is a senior manager in the WebSphere Business Integration tooling area, and is architect for WBI debuggers. Bill has been at IBM for 16 years, and has worked in a number of areas, including two positions within the Centre for Advanced Studies (research associate and manager). Besides debuggers, his interests include business-driven development, concurrency and object-oriented design. Bill has a Ph.D. in parallel computing from Syracuse University.

**Julie Waterhouse** is an Advisory Software Developer with 14 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.

## Disclaimer

The views expressed in this paper are those of the authors and do not necessarily reflect the views of IBM Corporation.

## References

- [1] B. Alpern and F. B. Schneider. "Recognizing Safety and Liveness.". *Distributed Computing*, 2(3):117–126, 1987.
- [2] R. Alur and M. Yannakakis. "Model Checking of Message Sequence Charts". In *Proceedings of CONCUR'99*, pages 114–129, 1999.
- [3] M. Autili, P. Inverardi, and P. Pelliccione. "A Scenario Based Notation for Specifying Temporal Properties". In *Proceedings of SCESM, ICSE'06 Workshop*, 2006.
- [4] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. "Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step". In *Proceedings of EPEW/WS-FM'05*, pages 257–271, 2005.
- [5] L. Baresi, C. Ghezzi, and S. Guinea. "Smart Monitors for Composed Services". In *Proceedings of ICSOC'04*, pages 193–202, 2004.
- [6] Business Process Execution Language for Web Services. [www-128.ibm.com/developerworks/library/specification/ws-bpe%1/](http://www-128.ibm.com/developerworks/library/specification/ws-bpe%1/).
- [7] W. Damm and D. Harel. "LSCs: Breathing Life into Message Sequence Charts.". *FMSD*, 19(1):45–80, 2001.
- [8] A. Dardenee, A. van Lamsweerde, and S. Fickas. "Goal-Directed Requirements Acquisition". *Science of Comp. Prog.*, 20:3–50, 1993.
- [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 TOSEM*, 3(2):131–165, 1994.
- [10] M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of ICSE'99*, pages 411–420, 1999.
- [11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. "Model-based Verification of Web Service Compositions". In *Proceedings of ASE'03*, pages 152–163, 2003.
- [12] X. Fu, T. Bultan, and J. Su. "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services.". In *Proceedings of CIAA'03*, pages 188–200, 2003.
- [13] X. Fu, T. Bultan, and J. Su. "Analysis of Interacting BPEL Web Services.". In *Proceedings of WWW'04*, pages 621–630, 2004.
- [14] Y. Gan. "Runtime Monitoring of Web Service Conversations". Master's thesis, Department of Computer Science, University of Toronto, April 2007.
- [15] R. Grosu and S. A. Smolka. "Safety-Liveness Semantics for UML 2.0 Sequence Diagrams". In *Proceedings of ACSD'05*, pages 6–14, 2005.
- [16] D. Harel and S. Maoz. "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams.". In *Proceedings of SCESM, ICSE'06 Workshop*, pages 13–20, 2006.
- [17] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. "STAIRS: Towards Formal Design with Sequence Diagrams". *Software and System Modeling*, 4:355–357, 2005.
- [18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- [19] IBM WebSphere Business Integration Software. [www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint](http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint).
- [20] IBM WebSphere Integration Developer. [www-306.ibm.com/software/integration/wid/](http://www-306.ibm.com/software/integration/wid/).
- [21] IBM WebSphere Process Server. [www-306.ibm.com/software/integration/wps/](http://www-306.ibm.com/software/integration/wps/).
- [22] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. "Synthesis of Correct and Distributed Adapters for Component-Based Systems: an Automatic Approach". In *Proceedings of ASE'05*, pages 405–409, 2005.
- [23] P. Inverardi, H. Muccini, and P. Pelliccione. "CHARMY: An Extensible Tool for Architectural Analysis". In *Proceedings of EFEC/FSE'05*, September 2005.
- [24] ITU-TS. "ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96)". Technical report, ITU-TS, Geneva, 1996.

- [25] JFLAP. [www.jflap.org/](http://www.jflap.org/).
- [26] R. Kazhamiakin and M. Pistore. “A Parametric Communication Model for the Verification of BPEL4WS Compositions”. In *Proceedings of EPEW/WS-FM’05*, pages 318–332, 2005.
- [27] A. Lazovik, M. Aiello, and M. P. Papazoglou. “Associating assertions with Business Processes and Monitoring Their Execution”. In *Proceedings of ICSOC’04*, pages 94–104, 2004.
- [28] Z. Li, J. Han, and Y. Jin. “Pattern-Based Specification and Validation of Web Services Interaction Properties”. In *Proceedings of ICSOC’05*, pages 73–86, 2005.
- [29] Z. Li, Y. Jin, and J. Han. “A Runtime Monitoring and Validation Framework for Web Service Interactions”. In *Proceedings of ASWEC’06*, pages 70–79, 2006.
- [30] K. Mahbub and G. Spanoudakis. “Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience”. In *Proceedings of ICWS’05*, pages 257–265, 2005.
- [31] A. Møller. Automaton Package. [www.brics.dk/automaton/](http://www.brics.dk/automaton/).
- [32] W. N. Robinson. “Monitoring Web Service Requirements”. In *Proceedings of RE’03*, pages 65–74, 2003.
- [33] M. Shanahan. “The Event Calculus Explained”. In *Artificial Intelligence Today*, pages 409–430. 1999.
- [34] M. Smith, G. Holzmann, and K. Etessami. “Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs”. In *Proceedings of RE’01*, pages 14–22, 2001.
- [35] H. Störrle. “Assert, Negate and Refinement in UML 2 Interactions”. In *Proceedings of CSD-UML’03*, pages 79–94, 2003.
- [36] Unified Modeling Language (UML 2.0). [www.uml.org/](http://www.uml.org/).