Model-Based Simulation of SOAP Web Services From Temporal Logic Specifications

Sylvain Hallé
Département d’informatique et de mathématique
Université du Québec à Chicoutimi, Canada
Email: shalle@acm.org

Abstract—This paper presents a methodology for generating a web service “stub” that simulates the behaviour of a real-world SOAP web service. The simulation is driven by a formal description of the original service’s input and output parameters, messages, and ordering constraints between messages, using an extension of Linear Temporal Logic called LTL-FO+. This logic is rich enough to express complex behaviours taken from real-world web services, where the structure of future messages and valid parameter values are interdependent. Given a history of previous interactions, a sound, symbolic algorithm is described that generates on-the-fly a new message that is a valid continuation of that history with respect to the LTL-FO+ specification. By providing a faithful placeholder for an actual third-party web service, this algorithm can be used as a development and testing tool. Empirical evaluation shows how such an approach outperforms a previous attempt that relied on a model checker to produce each new message.

Keywords—model-driven development; web services; temporal logic

I. INTRODUCTION

A major part of the interest created by so-called web services is their possibility to distribute processing capabilities across a number of loosely coupled functional entities communicating over standardized messages. Companies such as Google, Amazon or PayPal have been making an increasing part of their revenue by exposing and selling their functionalities as instances of services, reusable as background components by third-party application developers.

This appealing prospect, however, is mitigated by two factors. First, web service description languages, such as WSDL [1], describe the structure of each message, but not any eventual sequential dependencies between messages or data dependencies between parameters. This makes them particularly ill-suited to long running interactions where information about past requests matters in what constitutes a valid message. Second, the typical loose coupling of web services entails that all involved components seldom reside within the same organization. For example, some company’s web application might outsource its inventory management to the Amazon web services, while using the PayPal web service API for credit card transactions. This makes the debugging of any such composite application a non-trivial task, as some parts of the system are not under the developer’s control.

Web service “stubs” are small pieces of code that intend to help the developer by simulating the input-output behaviour of a real-world web service counterpart. When used in place of the actual service, they allow a third-party application to run in a simplified and closed environment suitable for testing. Some development tools allow stubs to be generated automatically from the WSDL specification of request and response messages of the original service, and provide templates for the generation of a simple set of stock response messages based on request types.

Yet, the behaviour of many well-known web services far exceeds request-response patterns independent of each other: as we shall see in Section II, the set of valid requests and parameter values sometimes depends on messages and values encountered earlier in the interaction. These constraints are called “data-aware” web service properties. An inventory of existing solutions will reveal that in this respect, most automatically generated web service stubs are closer to dummy placeholders than to reasonably faithful interacting partners.

In an earlier work [2], it was argued that a “universal stub” could be built, driven from a declarative specification of the service to simulate. Given a formula \(\varphi\) expressing the service’s behaviour in some logical language \(L\), the universal stub produces valid messages simulating the original service by finding a satisfying solution to \(\varphi\). Unfortunately, data-aware properties cannot be expressed easily in classical logics and rather belong to an extension of Linear Temporal Logic (LTL) with first-order predicates, called LTL-FO+, for which satisfiability solvers do not yet exist. This fact is illustrated in Section III.

The present paper attempts to bridge that gap and presents a universal web service stub based on LTL-FO+ specifications. Contrarily to an earlier work, which converted an LTL-FO+ specification back into classical LTL and used a model checker as a makeshift satisfiability solver [3], this paper presents in Section IV a purpose-built, symbolic satisfiability algorithm for LTL-FO+. It generates on-the-fly a new message that is a valid continuation of the current trace with respect to the input specification.

By natively handling LTL-FO+’s first-order quantifiers, this algorithm avoids the exponential blow-up associated...
with a translation to LTL. In this regard, an empirical evaluation of a prototype implementation, in Section V, shows significant improvement over the model checker solution by more than two orders of magnitude.

II. COMPLEX STUBS FOR WEB SERVICES

The relevance of developing complex web service stubs rests on the claim that many existing web services exhibit a behaviour that cannot be faithfully simulated by a mere dispatch of hard-coded responses. In this section, we present an example service that demonstrates some of this complex behaviour.

A. A Shopping Cart Service

Our example is inspired from a real-world web service called the Amazon E-Commerce Service (ECS) [4]. The ECS offers a public access for browsing Amazon’s inventory of items, as well as basic operations for creating and manipulating a virtual shopping cart made of these items. These operations are available through a web service API, by exchanging sequences of SOAP requests and responses with Amazon’s server.

Each request and response takes the form of a message in XML format. For example, searching an item based on some keyword “abc” can be done by sending the following message:

```xml
<ItemSearch>
  <Term>abc</Term>
</ItemSearch>
```

The server then replies with the list of item IDs corresponding to the keyword:

```xml
<ItemSearchResponse>
  <Items>
    <ItemID>123</ItemID>
    <ItemID>456</ItemID>
    ...
  </Items>
</ItemSearchResponse>
```

Similarly, a shopping cart can be created with a CreateCart message, whose response provides a CartID, say “ID123”. Subsequently, item IDs can be added to that cart through a message of the following form:

```xml
<CartAdd>
  <CartID>ID123</CartID>
  <Items>
    <ItemID>123</ItemID>
    <ItemID>456</ItemID>
    ...
  </Items>
</CartAdd>
```

Items can be removed using a similar CartRemove message, and the contents of the cart can also be wiped out using CartClear, each time passing the relevant CartID as a parameter.

We finally suppose that a user can send an empty RequestPayment/ message to obtain a PaymentForm response giving instructions regarding payment of cart items, and an empty Logout/ message to finish the transaction. Any attempt by the user to logout from the system is replied with the payment form instead of the logout confirmation if the shopping cart is not empty.

B. Constraints on the Shopping Cart Service

From the set of all possible sequences of messages taken from the description above, not all of them constitute plausible interactions with the shopping cart service. Indeed, even such a simple web service is subject to a number of constraints in the way its various operations can occur.

First, we find constraints related to the structure and values inside each message. For example, Term is the only allowed element inside an ItemSearch; CartID must appear in CartAdd, but not in CartCreate, and so on. In addition, each parameter contains values taken from some domain: the search term “abc” does not constitute a valid CartID. Second, all requests must be replied with their appropriate response operation. Hence, it does not make sense for the service to reply to an ItemSearch request with a CartCreateResponse.

A correct invocation of this service involves a number of additional constraints linked to the semantics of the operations involved. As an example, we list four of them:

1) Cart operations must begin with a CartCreate message.
2) Once a cart is created, the same CartID must be passed in all requests and responses.
3) The same item cannot be added via CartAdd twice to the same shopping cart: one must use the CartModify operation to change the quantity of an existing item.

The reader is referred to [5] for a more in-depth presentation of such message constraints on this and other services.

C. A Case for Web Service Stubs

As one can see, the shopping cart management functionalities of this example could be reused by a third-party application developer. However, there exist several reasons why the developer might not want to interact with the actual service before its launch into production.

A first obvious reason is to prevent actual transactions from being performed during the testing phase. Although shopping cart manipulations are relatively harmless until checkout is attempted, any operation with unrecoverable side effects (financial transactions, shipping, actual creation or deletion of objects) should be prevented. Ironically, those sensitive operations are often those with the most complex semantics and hence require more testing and debugging.

To this end, some providers offer a copy of their actual services running in a closed environment for developers to test with. Amazon Web Services [6] and PayPal [7] provide such “sandboxes”. Once an application ends its testing phase, the URL endpoint of the sandbox is replaced
by the URL of the real service, which is supposed to work in the exact same way. While such a principle allows the highest degree of faithfulness, there exist cases where such environments are not available for the service in question. Moreover, to debug a particular piece of code only reachable through a particular sequence of responses, one must find a way for the service to respond that particular sequence —yet the sandboxed service is not under the developer’s control.

A second solution consists in replacing the actual web service by a simple “stub” that stands as a placeholder that receives and returns messages similar to those of the actual service. By analogy with programming languages, one can think of a stub as a simple local function returning a value of the same type as some more complex call to an external library. This simpler function allows the developer to build and debug their program in a closed environment before dealing with the actual link to the real one. In the following, we survey the existing approaches suitable for the development of web service stubs.

1) Web Service Stubs: A web service stub can be written by hand using a simple scripting language and a couple of hard-coded responses. This approach, however, becomes quickly inappropriate and tedious if one wishes to achieve a higher degree of faithfulness with respect to the service to simulate. To this end, a few commercial and experimental tools have been developed to help generate web service stubs from some input specification. For example, a commercial development tool for SOAP web services, called soapUI [8], allows a user to create “mock web services”. The WSDL document declaring the structure of each XML request and response is automatically parsed, and a boilerplate message for each is given to the developer, who then fills each element with some hard-coded values.

Another solution consists in building a script that produces random, yet WSDL-compliant messages upon request; this can be used, for instance, to generate test cases for web services [9]. A tool called TAXI (Testing by Automatically generated XML Instances) has been developed to automatically produce valid instances of XML Schemas [10]. It has been used in the field of web services, to automatically generate XML documents with a given structure to be sent as an input to a web service to test. A similar tool called WebSob [11], when used in conjunction with unit test generators such as JCrasher [12], can generate random WSDL-compliant requests and discover incorrect handling of nonsensical data on the service side.

2) Model-Based Simulation: The previous solutions treat request-responses as atomic patterns independent of each other, and abstract away any ordering dependencies that might exist between different messages. Yet, as we have seen earlier, this does away with a wide number of constraints exhibited by some real-world web services. A second line of solutions exploit richer declarative models of systems to dynamically generate valid, long running sequences of requests and responses. One notable example is the recent development of the Play-Engine [13]. The engine uses specifications expressed as Live Sequence Charts (LSC) [14], and is composed of two parts: “Play-In” is used to capture execution scenarios and to infer the appropriate LSC; on the opposite the “Play-Out” plays back an LSC specification, by dynamically selecting the transitions to fire according to the current state of the system.

Formal grammars have also been used to generate test data [15]–[18]. The principle has been extended into interface grammars, which have been used to represent the possible sequences of messages in SOAP web services. [19]. The approach takes into account message structure and relationships between message elements [5]. Yet, grammars are monolithic; the same production rule often plays a role in more than one constraint. This makes it hard to take away or to add a new requirement without rewriting substantial parts of the specification. In addition, grammars do not provide an easy way to constrain request parameters.

3) A Model Checker-based Stub: Closer to the problem at hand is an earlier work describing a universal stub engine for web services [3]. The stub takes as input a description of message structures, parameters, and possible values, along with a list of formulæ expressed in an extension of Linear Temporal Logic with first-order quantification on message parameters, called LTL-FO+. As we shall see in Section III, this logic is rich enough to express data-aware constraints of the kind described above, and hence allows for a description of complex dependencies between messages.

Since requirements are expressed using a logical formalism, the production of valid sequences of messages according to the specification becomes a problem of satisfiability solving. More precisely, each message of a trace is associated to one “state” of a purpose-built finite state machine $M$, with various state variables encoding the message’s parameters and values. The set of LTL-FO+ specifications $\varphi$, giving constraints on messages, is translated back into classical LTL assertions $\varphi'$ on the traces of the state machine. The NuSMV model checker [20] is then asked in the background to verify $\varphi'$ against $M$; if a counter-example is found, its contents can be converted back into a new message, which, by construction, satisfies $\varphi'$ and hence is a valid continuation of the current trace.

In this solution, the translation of LTL-FO+ into LTL and the use of an LTL model checker acts as a makespan satisfiability solving procedure, for lack of a “native” LTL-FO+ solver. It was suggested that NuSMV’s highly optimized exhaustive state space search, combined with the relative small size of the problem to solve, would outperform any home-brewed constraint solver specific to LTL-FO+. Preliminary experimental results indicated that the use of a model checker as a back-end engine to produce message traces compliant with these formulæ is feasible for modest domain sizes.
III. A Formal Specification of SOAP Web Services

Several observations can be made from the constraints of the shopping cart service. First, these constraints go beyond single request-response patterns and express relationships between multiple requests and responses. For example, property 1 stipulates that a user cannot invoke any of the CartAdd, CartRemove, CartClear or CartModify before a CartCreate has been requested. Second, the constraints correlate values of elements across multiple messages: hence property 2 states that the value of CartID returned by CartCreateResponse must be passed on to any future cart request or response. Third, occurrences of values in past messages can constrain what messages and values can be sent in the future: property 3 states that any ItemID appearing in a CartAdd request must not reappear in any future CartAdd request. This last type of constraint is called "data-aware", since the sequence of messages and the valid future CartAdd request. This last type of constraint is called "data-aware", since the sequence of messages and the valid parameter values inside messages are interdependent.

In the present section, we describe a formal specification of these kinds of properties and briefly describe the logic LTL-FO+.

A. Message Structure and Element Domains

Messages exchanged with the web service typically consist of XML documents sent through the SOAP protocol. A first step consists in providing a formal representation of these messages.

Let $E = \{e_1, e_2, \ldots, e_n\}$ be a finite set of message elements. We denote as $E^*$ the set of all finite sequences of elements, called paths. In the CartAdd message shown earlier, we have $E = \{\text{CartAdd}, \text{CartID}, \text{Items}, \text{ItemID}\}$. A message schema is a set of prefix-closed paths. Intuitively, the schema defines the structure of the message by giving the list of all possible paths from the root of the message to its leaves. For example, CartAdd/Items/Item is part of the schema while CartAdd/CartID/Item is not. A compact notation for expressing this schema is as follows:

\[
\text{CartAdd[}
\begin{align*}
\text{CartID},
\text{Items[Item*]}
\end{align*}
\text{]} = D
\]

Let $D$ be a finite set of values. The element domain is a mapping $\text{Dom} : E^* \rightarrow P(D)$; for $\pi \in E^*$ a path, $\text{Dom}(\pi)$ assigns a set of possible values. Given a simple message schema $S = \{\pi_1, \ldots, \pi_n\}$, a message instance is a mapping $m : S \rightarrow 2^D$ which assigns, for each path $\pi_i \in S$, a subset of $\text{Dom}(\pi_i)$. Hence, the particular instance $m$ of the CartAdd message schema shown earlier is such that $m(\text{CartAdd}/\text{CartID}) = \{\text{ID123}\}$, and $m(\text{CartAdd}/\text{Items}/\text{Item}) = \{123, 456, \ldots\}$. We denote by $M$ the set of all such messages.

\[
\begin{align*}
\overrightarrow{m} \models \alpha & \equiv m_0 = (a, \ast) \text{ and } a = \alpha \\
\overrightarrow{m} \models p = v & \equiv m_0 = (a, \ast) \text{ and } \bullet(p) = v \\
\overrightarrow{m} \models \neg \varphi & \equiv \overrightarrow{m} \not\models \varphi \\
\overrightarrow{m} \models \varphi \land \psi & \equiv \overrightarrow{m} \models \varphi \text{ and } \overrightarrow{m} \models \psi \\
\overrightarrow{m} \models \varphi \lor \psi & \equiv \overrightarrow{m} \models \varphi \text{ or } \overrightarrow{m} \models \psi \\
\overrightarrow{m} \models \varphi \rightarrow \psi & \equiv \overrightarrow{m} \not\models \varphi \text{ or } \overrightarrow{m} \models \psi \\
\overrightarrow{m} \models \Box \varphi & \equiv m_0 \models \varphi \text{ and } \overrightarrow{m} \not\models \varphi \text{ or } \overrightarrow{m} \models \psi \\
\overrightarrow{m} \models \varphi \bigoplus \psi & \equiv m_0 \models \varphi \text{ and } \overrightarrow{m} \not\models \varphi \text{ and } \overrightarrow{m-1} \models \varphi \\
\overrightarrow{m} \models \varphi \bigodot \psi & \equiv m_0 \models \psi, \text{ or both } m_0 \models \varphi \text{ and } \overrightarrow{m-1} \models \varphi \psi \not\models \varphi \bigoplus \psi \\
\overrightarrow{m} \models \exists_x : \varphi & \equiv \overrightarrow{m} \models \varphi[x/v] \text{ for all } v \in m_0(\pi) \\
\overrightarrow{m} \models \forall_x : \varphi & \equiv \overrightarrow{m} \models \varphi[x/v] \text{ for some } v \in m_0(\pi)
\end{align*}
\]

Table I

<table>
<thead>
<tr>
<th>SEMANTICS OF LTL-FO+</th>
</tr>
</thead>
</table>

B. A First-Order Linear Temporal Logic: LTL-FO+

A trace of messages $\overrightarrow{m}$ is a sequence $m_0, m_1, \ldots$ such that $m_i \in M$ for all $i \geq 0$. Assertions over message traces are expressed in an extension of Linear Temporal Logic, called LTL-FO+, already presented in e.g. [21]. Its building blocks are atomic propositions, which are of the form $x = y$, where $x$ and $y$ are either variables or constants.

These atomic propositions can then be combined with Boolean operators $\land$ ("and"), $\lor$ ("or"), $\neg$ ("not") and $\rightarrow$ ("implies"), following their classical meaning. In addition, LTL temporal operators can be used. The temporal operator $G$ means "globally". For example, the formula $G \varphi$ means that formula $\varphi$ is true in every message of the trace, starting from the current message. The operator $F$ means "eventually"; the formula $F \varphi$ is true if $\varphi$ holds for some future message of the trace. The operator $X$ means "next"; it is true whenever $\varphi$ holds in the next message of the trace. Finally, the $U$ operator means "until"; the formula $\varphi U \psi$ is true if $\varphi$ holds for all messages until some message satisfies $\psi$.

Finally, LTL-FO+ adds quantifiers that refer to parameter values inside message. Formally, the expression $\exists_x : \varphi(x)$ states that in the current message $m$, there exists a value $v \in m(\pi)$ such that $\varphi(v)$ is true. Dually, the expression $\forall_x : \varphi(x)$ requires that $\varphi(v)$ holds for all $v \in m(\pi)$. The semantics of LTL-FO+ are summarized in Table I, where $\overrightarrow{m} = m_0, m_1, \ldots$ is a sequence of messages; the reader is referred to [21] for a deeper coverage of LTL-FO+ in a related context.

Using this logic, it is possible to convert the schema definition into a set of LTL-FO+ formulae. It suffices to exploit the fact that the expression $\exists_x : \top$ returns true if path $\pi$ exists in the current message, and false otherwise. For example, the fact that no CartAdd message must occur before a CartCreate can be written in a straightforward
manner as:

\[ (\neg \exists \text{CartAdd} \land \top) U (\exists \text{CartCreate} \land \top) \]

Similarly, the fact that the CartID returned in CartCreateResponse must appear in all subsequent messages is formalized as follows:

\[ G (\forall \text{CartCreateResponse}/\text{CartID} : X G \forall \text{CartAdd}/\text{CartID} y : x = y) \]

This formula stipulates that globally, any CartID element \( x \) appearing in a CartCreateResponse is such that, from the next message on, any CartID element \( y \) appearing in a CartAdd message is such that \( x = y \). The same property can be repeated for other Cart message types.

For the sake of completion, we also give the formalization of constraint #3, asserting that ItemIDs appearing in a CartAdd must not reappear in a future CartAdd:

\[ G (\forall \text{CartAdd}/\text{ItemId} x : X G ((\exists \text{CartAdd} \land \top) \rightarrow \exists \text{CartAdd}/\text{ItemId} y : x = y)) \]

IV. AUTOMATED GENERATION OF MESSAGE SEQUENCES

One can see how, by means of a couple of simple LTL-FO\(^+\) formulæ, a somewhat faithful encoding of the service’s behaviour can be achieved. That is, any trace of XML messages fulfilling these formulæ must be such that, for example, cart operations will be mimicked with relative precision, the CartAdd mechanism will faithfully remember all existing items in the cart. Consequently, a procedure that can generate a sequence of such requests, according to the temporal specifications, can form the basis of a web service stub for the shopping cart service. To this end, this section presents a satisfiability procedure for LTL-FO\(^+\).

A. An On-the-Fly Runtime Monitoring Algorithm

Since domains for each message element are finite, a naive strategy consists in systematically enumerating each possible message and comparing it against the specification, until finding a satisfying assignment of values for each element. Such a procedure is sound and complete, yet inelegant and of an unacceptable complexity. In general, assuming \( d \) message elements each having \( d \) possible values, there exist \( O(2^{nd}) \) possible messages. Clearly, significant savings can be achieved by generating messages from the ground up, using the specification as guidelines.

To this end, algorithms for the runtime monitoring of specifications can be put to use. In runtime monitoring, a sequence of messages is progressively fed to a procedure (the “monitor”) that updates its internal state after each message, and that can answer on-the-fly whether the property to watch is being violated. By making appropriate modifications, such a procedure can be adapted to work “in reverse” and to produce, rather than verify, a message based on its current internal state.

The LTL-FO\(^+\) satisfiability procedure is built on top of a runtime monitoring algorithm for LTL developed by Gerth et al. [22]. Starting from an initial formula, it applies a set of decomposition rules that produce a tree, each node of which is of the form \( \Gamma \models \Delta \). The left-hand side of each node, \( \Gamma \), contains a list of formulæ expressing constraints that must be true in the current state (i.e. message) of the trace. The right-hand side of the node, \( \Delta \), contains a list of formulæ that must be true in the next message of the trace. The resulting node shows how \( \varphi \) is kept in \( \Gamma \), while \( G \varphi \) is copied to \( \Delta \). On some occasions, the decomposition produces two alternatives: hence the formula \( F \varphi \) can be fulfilled either by \( \varphi \) in the current message, or by \( F \varphi \) being pushed to the next state. The decomposition continues until no rule can be applied.

B. Adaptation to LTL-FO\(^+\) Satisfiability

This original algorithm for LTL must be adapted to the present context. First, since the original algorithm was meant to verify a trace, it must be turned into a procedure that produces messages.

This seems straightforward at first glance: the left-hand side of a decomposition node contains a list of formulæ that must be true in the current message to process. Therefore, they can simply be interpreted as assertions on the content.
of the message, whose combination can be used to infer what to put into the message to generate. The various nodes produced by the decomposition rules can be seen as a set of alternatives for the creation of a message. For instance, the decomposition rule for $\varphi \lor \psi$ branches into two possibilities: create a message that satisfies $\varphi$, or create a message that satisfies $\psi$.

Yet, this simple interpretation leaves out possible solutions. When verifying a message against a property of the form $\varphi \lor \psi$, it is indeed sufficient to make sure that either one of $\varphi$ and $\psi$ is satisfied. However, when generating a message satisfying $\varphi \lor \psi$, these two alternatives entail that either a message satisfying $\varphi$ or a message satisfying $\psi$ will be generated—excluding the generation of a message where both $\varphi$ and $\psi$ are true. Similarly, the original decomposition of $\mathbf{F} \varphi$ excludes the generation of a trace where $\varphi$ is true in the current message, and $\varphi$ still applies to some message in the future. A third case must therefore be added to all branching rules, resulting in the modified set in Figure 2.

The second modification to the original algorithm is the addition of extra rules for the first-order quantifiers present in LTL-FO+. Again, quantifiers can be interpreted as assertions on the content of a message. An existential quantifier of the form $\exists x : \varphi(x)$ asserts that there must exist in the message to output an element with value $b$ at the end of path $\pi$, for some $b \in \text{Dom}(\pi)$, and such that $\varphi(b)$ is true. A possible alternative is therefore to add $b$ at the end of path $\pi$ (denoted in shorthand as $\varpi b$), and to make sure that $\varphi(b)$ is true. There exists one such possible alternative for each $b \in \text{Dom}(\pi)$; this is what the decomposition rule in Figure 2 shows.

Similarly, the universal quantifier $\forall x : \varphi(x)$ asserts that every value $x \in \text{Dom}(\pi)$ present in the message, if any, must satisfy $\varphi(x)$. If $S_\pi \subseteq \text{Dom}(\pi)$ is the set of all values $b$ such that $\varpi b$ is in $\Gamma$, then $\varphi(b)$ must be added to $\Gamma$. However, one can also add other values from $\text{Dom}(\pi)$ and assert that $\varphi$ is true for these new values as well. Therefore, for every $S \subseteq \text{Dom}(\pi) \setminus S_\pi$, we can create a new alternative that asserts $\varphi(b)$ and $\varpi b$ for each $b \in S$.

Starting from an initial LTL-FO+ formula, the repeated application of these new decomposition rules creates a tree whose all branches end when the left-hand side of a node contains only expressions of the form $\varpi b$ or $\neg \varpi b$, for various paths $\pi$ and values $b$. The list of assertion in each such leaf provides a straightforward “recipe” for creating a new instance of a message. It suffices to take each of the $\varpi$ terms one by one, and to create the branch and value stipulated by the expression. Figure 3 shows an example for the three terms $\varpi \text{Message/ID}1234$, $\varpi \text{Message/Items345}$, $\varpi \text{Message/Items678}$.

Finally, some leaves of the tree may have to be discarded for two reasons. If the node contains both $\varpi b$ and $\neg \varpi b$ for the same path $\pi$ and the same value $b$, then the conditions on the message to generate are contradictory, since they require both that some value $b$ be present and absent from the message at the end of the same path. Therefore no message can be produced that satisfies both conditions.

Another possibility is when the list of $\varpi$ terms contradicts the schema declaration. For example, if a schema does not allow the Items element to have more than one value, then the message created from the terms in Figure 3 must be discarded, as it assigns two leaves to this element.

C. Soundness and Completeness

The leaves remaining after this pruning step each represent a possible new message to create. From this set of candidate solutions, it suffices to pick one arbitrarily and instantiate the appropriate message. All the formulæ appearing in the right-hand side of the chosen node (i.e. $\Delta$) represent the conditions that must be true in the next message. A new start node can be created by copying these formulæ to the left-hand side, and the algorithm is ready for a new cycle.

For such an algorithm to be sound, we must show that any message it produces is indeed a valid extension of a trace satisfying the original LTL-FO+ formulæ. This is summarized in the following theorem.

**Theorem 1.** Let $N = \Gamma \models \emptyset$ be some start node, and $N' = \Gamma' \models \Delta'$ be some leaf node obtained by the application of the decomposition rules shown in Figure 2, and such that along the branch from $N$ to $N'$, for every path $\pi$, no term $\varpi b$ has been added after a universal quantifier of the form $\forall_x \varphi$ is decomposed. Let $\varphi$ and $\varphi'$ be the conjunction of the formulæ in $\Gamma$ and $\Delta'$, respectively. Finally, let $n'$ be the message created from the $\varpi$ terms in $\Gamma'$. Then, for every message trace $\overline{m}$, $\overline{m} \models \varphi$ if and only if $\overline{m}n' \models \varphi'$.

**Proof:** We only sketch the proof, which is done in two steps:

1) Let $N = \Gamma \models \Delta$, be a node, and $N_i = \Gamma_i \models \Delta_i$ for $i = 1, \ldots, n$ be the nodes resulting from the application of a decomposition rule. Define $\Psi(N) = \bigwedge \Gamma \land X \land \Delta$. Show that for every $i$, $\Psi(N_i) \rightarrow \Psi(N)$.

2) Let $N = \Gamma \models \emptyset$ the root node of a decomposition cycle, and $N' = \Gamma' \models \Delta'$ one of the leaves of the resulting tree. Then $\Gamma'$ is a list of terms of the form
Figure 2. Modified decomposition rules for the generation of a new message in LTL-FO+

\[ \oplus \pi \theta \; \text{let} \; m \text{ be the message built from these terms, and} \; \overline{m} \text{ be any trace of messages. Show that} \; \overline{m} \models \bigwedge \Delta' \text{ entails} \; m \overline{m} \models \bigwedge \Gamma. \]

We shall highlight the fact that this theorem requires an additional condition on the decomposition rules, namely that a universally quantified formula \( \phi \) for some path \( \pi \) be evaluated after the addition of any \( \oplus \) term with that same path \( \pi \). Otherwise, some \( \oplus \) term might introduce a new value that invalidates \( \phi \) after the quantifier has been handled. Consider for example the set of expressions \( \forall \pi x : x > 0 \), \( \exists \pi x : x = 0 \). Handling the first one will add any number of positive values at the end of path \( \pi \). Then, the existential quantifier will require the presence of 0 at the end of path \( \pi \). This invalidates the first quantifier, which has already been decomposed.

However, while this condition preserves the soundness of the algorithm, it also makes it incomplete. That is, there exist cases where the application of the decomposition rules refrains from generating a message because the condition is violated, even though a valid message might still exist. A decomposition strategy consists in delaying the processing of universal quantifiers as much as possible. In the shopping cart scenario, this strategy is sufficient to avoid invalidating the theorem’s condition. Every time the algorithm does output a message, this message is a valid continuation of the current trace with respect to the specification.

V. EXPERIMENTS

To provide first insights regarding the viability of the proposed solution, we implemented the LTL-FO\(^+\) satisfiability algorithm described in the previous section and developed a universal stub based on it. This stub was then compared, in various experiments, to the NuSMV-based stub described earlier. It shall be noted that a comparison of this stub with other related work described in Section II-C is not directly possible for lack of one or more of the characteristics required by LTL-FO\(^+\). Boilerplate stubs [8]–[11] cannot generate long-running sequences dependent on user input, while existing model- and grammar-based approaches [5], [13], [19] do not allow for existential quantification over nested message elements.

A. Prototype Implementation

The prototype stub consists of a set of Java packages for the symbolic manipulation of LTL-FO\(^+\) expressions. The code base for the stub is freely available under an open source license.\(^1\) The stub itself is a stand-alone class that takes as input a configuration file defining the structure of each possible message to simulate, the range of available values for message elements, and finally a list of LTL-FO\(^+\) formulæ, written in a text-only syntax. Table II shows an example of such a file.

Once instantiated, the stub can be asked to produce a new message based on the current state of its specification through the \texttt{generate()} method, which outputs an XML message produced by the LTL-FO\(^+\) satisfiability procedure. In addition, if the soundness condition is violated during the decomposition, a special flag can be queried to warn the stub’s consumer.

The experiments described here consider a web service with a single message with structure \( m[p*] \), and a single LTL-FO\(^+\) formula expressing the fact that the \( p \) element contains all values that previously appeared:

\[ G \left( \forall m/p x : X G (\exists m/p x' : x' = x) \right) \]

This formula is a generalized version of a constraint found in our earlier example: in the Amazon ECS, a shopping cart must contain the list of all items previously sent in an \texttt{add} message. It requires the stub to dynamically store a set of values having appeared in the \texttt{the} trace. This behaviour is unique to data-aware properties (propositional temporal logic alone cannot easily express these constraints) and is hence considered a good representative of the complexity of such properties.

\(^1\)http://beepbeep.sourceforge.net
We then compared the running times of the symbolic stub with the model checker stub in three experiments, detailed in the following. These experiments can be summarized coarsely as follows. For $a$ the message arity, $d$ the domain size and $\ell$ the length of the current trace, the NuSMV stub runs in time proportional to $\ell \cdot 2^{ad}$, while the symbolic stub introduced in this paper runs in time proportional to $2^d$.

B. Impact of domain size

The first experiment consisted in generating traces of 20 messages from this property, with messages of arity 10 and $\text{Dom}(p) = \{1, \ldots, n\}$, for $n$ ranging from 1 to 10. We then took the total running time for generating the 20 messages for both stubs. The results are shown in Figure 4. One can see that, despite a similar exponential growth in the size of the domain, the symbolic stub is on average more than 500 times faster, taking about 0.4 seconds to generate a trace where the NuSMV stub requires more than 6 minutes. Indeed, since the NuSMV model checker only supports propositional LTL (in addition to CTL) for its specification language, the NuSMV stub must translate the original LTL-FO$^+$ formula back into a propositional LTL formula. Moreover, the number of message elements must be bounded so that each message can be encoded into a finite set of state variables. Each message element is represented as a name-value pair of the form $p_i = v_i$, where $p_i$ is a state variable encoding the element’s “name path” from the message’s root, and $v_i$ is a state variable containing the element’s value. If $k$ is the maximum number of message elements, then LTL-FO$^+$ quantifiers are translated as follows:

- $\exists p x : \varphi \equiv \bigvee_{p \in \text{Dom}(p)} (\forall_{i=1}^k p_i = p \land v_i = v \land \omega(\varphi[x/v]))$
- $\forall p x : \varphi \equiv \bigwedge_{p \in \text{Dom}(p)} (\forall_{i=1}^k (p_i = p \land v_i = v) \rightarrow \omega(\varphi[x/v]))$

The formula hence becomes exponential in the size of the domain, and can indeed become very large: the expanded formula corresponding to the rightmost point in Figure 4 is 264,000 characters long. Yet, it has been observed that for two equivalent problems, model checkers tend to be much more efficient at handling very large finite-state machines than very large formulæ [23]; hence the symbolic algorithm, while still being exponential in the domain size, performs much better.

C. Impact of message arity

The second experiment consisted in settling on a domain size, and varying the arity of the messages. The results are shown in Figure 5. One can see that the arity only has an effect on the NuSMV stub, since the translation into a NuSMV finite state model requires a fixed bound on the number of state variables for encoding each message. In turn, as described above, the translation of an LTL-FO$^+$ into LTL is exponential in the arity of the messages, explaining the roughly exponential growth of running time. On its side, the symbolic stub does not require any fixed bound on message arity, and hence its running time is identical regardless of this parameter.

D. Time per message

The third experiment consisted in measuring the time to generate each successive message along a trace, for a fixed domain size and fixed message arity. That is, the stub was repeatedly asked to extend the current trace by one more message, and the elapsed time to generate each of them is plotted in Figure 6.

One can see how the time to generate one message, in the NuSMV stub, is increasing as the trace lengthens. This is consistent with findings in our earlier study, which blamed
part of that behaviour on the fact that the model checker is not called incrementally and has to repeat its analysis from scratch at every new message. However, this is no longer the case for the symbolic algorithm, where each new message is built upon the current state of the generator. Consequently, the time to generate each message, in addition to being shorter, more importantly does not increase with trace length.

E. Choice of solution

Another difference between the NuSMV and the symbolic stub, not represented in the previous quantitative analysis, consists of the way in which each new message is generated. In the NuSMV stub, the whole problem is sent to the NuSMV model checker, which is considered as a “black box”. Since it is asked to find a counter-example to the negation of the actual specification, its process ends as soon as one such counter-example is found. Therefore, there is no easy way, if at all, to provide additional criteria for the selection of the returned message.

On the other hand, the present algorithm is implemented to generate the complete decomposition tree from the current state; only then is a solution picked at random from the set of all available conditions. This would make it much easier, in a longer term, to parameterize the stub with additional criteria regarding the selection of the message to return. Incidentally, it also places the symbolic stub at a disadvantage, in terms of performance, with the NuSMV stub, since it does not stop at the first solution it encounters.

VI. CONCLUSION

The interest of the automated generation of a web service stub based on a logical specification of its behaviour is closely dependent on the faithfulness of the simulation that can be achieved. As we have shown, this, in turn, requires the use of logical languages that are typically more expressive than classical Boolean logic or even Linear Temporal Logic, such as LTL-FO$^+$. While the problem of generating sequences of messages satisfying an LTL-FO$^+$ specification can be temporarily circumvented through a translation into LTL and the use of a model checker, its definitive solution requires the development of a sound and dedicated procedure where first-order quantifiers are handled symbolically. The present paper described such a procedure and experimented a universal web service stub prototype based on it.

The experiments highlighted the fact that the reliance on a model checker turns out to be a double-edged sword. On the positive side, the implementation of the model checker stub only requires to provide a translation of the original problem into an equivalent input file for the model checker, and a way to parse its response back into a message. However, it turned out that the “hijacking” of a model checker to use as a model finder, while conceptually correct, still has adverse effects on the performance of the procedure. First-order quantifiers are blown up into an exponential number of copies of the original formula, where each variable is replaced by all possible combinations of its values. This exponential growth has a greater impact than expected, compared to the symbolic manipulation of quantifiers in the algorithm proposed here. In addition to greatly improved running times with respect to domain size, the symbolic algorithm is also shielded from any assumptions on the arity of the messages. Consequently, even though the universal stub is still a proof-of-concept, Java-based prototype, its performance on the same LTL-FO$^+$ specification is in some cases more than 500 times better.

Yet, the proposed algorithm imposes restrictions on the decomposition of the input specification to ensure its soundness. This is made at the price of completeness, which entails that in some cases, the stub may refrain from returning a message when it cannot guarantee its compliance to the specification. An improvement in the decomposition strategy to restore completeness is hence considered as important future work.

REFERENCES


