Cooperative Runtime Monitoring of LTL Interface Contracts

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

Abstract—Requirements on message-based interactions can be formalized as an interface contract that specifies constraints on the sequence of possible messages that can be exchanged by multiple parties. At runtime, each peer can monitor incoming messages and check that the contract is correctly being followed by their respective senders. We introduce cooperative runtime monitoring, where a recipient “delegates” its monitoring task to the sender, which is required to provide evidence that the message it sends complies with the contract. In turn, this evidence can be quickly checked by the recipient, which is then guaranteed of the sender’s compliance to the contract without doing the monitoring computation by itself. A particular application of this concept is shown on web services, where service providers can monitor and enforce contract compliance of third-party clients at a small cost on the server side, while avoiding to certify or digitally sign them.

Keywords—runtime monitoring; temporal logic; distributed computing; web services

I. INTRODUCTION

Message-based communications can be found in a wide variety of domains and applications. Recently, the rise of web services as a paradigm for decoupling IT systems into independent functional units further confirmed the appropriateness of messaging as an interaction model. Even platforms such as the iPhone OS or Google Wave’s plugin architecture, which allow third-party applications to consume system resources by calling appropriate methods, can be likened to a form of messaging.

Although in each of these cases, a “message” is conveniently regarded as an atomic unit of data, in many practical contexts, this assumption does not hold. The semantics of the underlying operations impose that these messages be sent and received following some agreed-upon sequence or contract. Various techniques have been suggested to ensure that a “client” (in its most general meaning) correctly accesses the “server’s” resources following its contract.

For example, the extensive testing of a potential client, followed by the generation of a “digital signature” that is checked at execution time, can ensure a server that only pre-authorized (and supposedly compliant) clients are allowed to interact. However, pre-certification is cumbersome, and in many situations, does not even prevent a pre-validated client from being compromised. This is especially true of Ajax web applications, which run on a remote machine and whose plain-text source code can easily be tampered with. The only safe alternative is for the server to check contract compliance dynamically, as the message exchange unwinds, through a process called runtime monitoring. Yet, traditional runtime monitoring lacks the appeal of pre-validation, in particular since the whole compliance checking must be done by the server itself at runtime.

This paper introduces a novel approach to contract compliance checking called cooperative runtime monitoring. In this framework, the client performs the same computation the server would do on its side when receiving a new message, and sends the result. To make sure that the client can be trusted, a proof that the message is a valid continuation of the current transaction is appended to the message. The server merely checks the proof to be ensured of the client’s “good faith”.

This approach presents numerous advantages. First, it shifts the monitoring load on each client, which in turn only need to monitor their particular instance of the contract. Second, it still ensures the server that a client follows the contract, message by message, without requiring a rigid and cumbersome a priori certification process. We provide initial results on a proof-of-concept system that uses Linear Temporal Logic as its specification language, and show that the same compliance guarantee can be ensured on the server side for less than 10% of the computational cost of classical monitoring.

II. MONITORING OF MESSAGE-BASED CONTRACTS

Although message-based communication scenarios abound, perhaps the most fertile is the context of web services and web-based applications. In a web service interaction, two independent entities expose their functionalities as a set of possible requests and responses, which can be sent and received through XML documents (“messages”) containing element names nested within each other. A web application is a particular case of this scenario where one of the partners is a web browser running a client-side Javascript application.

A. A Motivating Example

To illustrate the need for compliance checking of message-based contracts, we provide an example taken from
an actual web service, the Amazon E-Commerce Service (ECS). The ECS is a free service that exposes Amazon’s product data, and provides a number of operations to search for items in Amazon’s catalogue, create and manipulate the contents of a shopping cart that can eventually be “checked out” for payment directly on Amazon’s web site. Any third-party developer can obtain a free Amazon account and create a web application that interacts in the background with the ECS, through SOAP request-response messages.

Although each of these operations is intended as a simple request-response pattern of interaction, the semantics of each operation is such that their ordering is also important. For example, it does not make sense (and is actually an error, according to Amazon’s documentation) to add contents to a shopping cart if no shopping cart has first been created. Similarly, one cannot delete an item from a shopping cart before adding anything to it. We list a few constraints on message sequences derived directly from Amazon’s online documentation:

P1. The CartCreate message must precede any occurrence of CartAdd, CartModify or CartRemove.

P2. CartModify and CartRemove must occur after one CartAdd.

P3. If CartClear is invoked, no CartAdd, CartModify or CartRemove can occur before a new CartCreate.

These are simplifications of the actual constraints that do not take into account finer conditions relying on the data inside the messages. The reader is referred to [1] for a detailed description of sequential constraints for the ECS.

The Amazon ECS is not an exceptional example. Other contexts where the sequence of messages must be taken into account have been described [2], [3], including the PayPal web service [4] and the “channel contracts” in the Singularity operating system [5].

B. Compliance to Message-based Contracts

In addition to the pre-defined structure of each XML message, specified in a WSDL document that accompanies each web service, the conjunction of all sequential constraints on these messages describes an interface contract. However, while checking that each message has a valid structure is straightforward, making sure that each message is the continuation of a valid sequence is much harder. No standard exists at the moment to formalize sequential contracts in the way WSDL does for message structure. It is therefore left to each application developer to peruse the plain-text documentation of a given web service, and make sure that an Ajax application interacting with this service follows all the sequential constraints scattered across that documentation.

Worse yet, even though a developer is required to have a valid Amazon user ID to use the Amazon web services, this ID does not provide any assurance that the application she develops follows the contract. This issue is not specific to Amazon, or even to web services in general. Any platform allowing third-party applications to use its resources is faced with the same problem. As soon as some form of “dialogue” is implied between a client and a server, the question of making sure that this dialogue is followed by both parties is an open problem. A couple of remedies have been explored, which we describe below.

1) A priori Certification: A possible solution is for the server’s organization to certify each third-party client beforehand for contract compliance. Ways of doing so include thorough testing in a variety of possible use cases, or static analysis such as model checking. Once the organization deems the client good enough, it grants a digital signature, possibly computed from the client’s code, or some other form of authorization. The server can then require each client to provide a valid signature in order to start interacting. This is the approach followed, to different degrees, by many current application platforms, such as the AppStore for the Apple iPhone, or the plugin architecture for Google Wave [6].

However, this process requires resources (testing teams or powerful static analysis tools) dedicated to this certification. Moreover, numerous examples show that this barrier can be bypassed (e.g. “jail-breaking” of iPhones [7]). In the web realm, this approach is simply null and void, since the plain-text Javascript code of Ajax applications can be subverted at runtime through techniques such as prototype hijacking [8]. Not surprisingly, the a priori certification of web applications has never been seriously considered.

2) Server-side Runtime Monitoring: A possible way to escape this issue is to monitor the message exchange between a client and a server at runtime. In particular, the server, requiring compliance from its clients, follows the message exchange and constantly compares it to the contract specification. This can be integrated directly in the server’s application logic, or as an independent agent at the server’s interface that performs a first compliance check before relaying the message to the application proper, as explored in [9].

No trust assumption is required on the clients, since any non-compliant behaviour is anyway checked (and prevented) by the server itself. This is shown in Figure 1(a).

Although this approach ensures contract compliance from clients, its downside is to still let any message reach the server itself. The application is protected from erratic message exchanges, but the computational burden of this protection is carried solely by the server, without requiring anything from the clients.

3) Client-side Runtime Monitoring: A sensible workaround to this issue would be to have each client monitor the message exchange with the server, as shown in Figure 1(b). This way, instead of having a centralized monitoring of every transaction on the server side, the same computational load is distributed among all clients, which each follow only one transaction (i.e. theirs). Moreover, no time is wasted on the server side to weed out
bad sequences of messages, as each client censors itself through its monitoring “guard dog”. Client-side monitoring was studied in [10], where a runtime monitor was placed atop an Ajax web application and run by the user’s web browser. Studies on a real web application using the Amazon web services showed that such an approach does not put undue load on the client web application, while at the same time makes sure that the contract is followed.

C. Cooperative Runtime Monitoring

There is, however, one caveat to this approach: how can a server be guaranteed that a client monitors the right contract (or any contract at all) when interacting with it? Clearly, a server cannot let all its guards down without “trusting” the other end of the exchange. We therefore introduce an approach that intends to be a middle-ground solution between not trusting the client (server-side monitoring), and completely trusting the client (client-side monitoring). In cooperative runtime monitoring, the task of asserting the compliance of a message to some sequential contract is shared between the client and the server.

The process is shown in Figure 2. First, the sender computes (with a function $\gamma$) a “proof” that its message complies with the current state of the contract (dashed rectangle) and includes this proof along with the message to send. Upon reception of this message, the server checks that this proof is coherent with the message it accompanies (function $\mu$), and then checks the proof itself (function $\nu$). The message is relayed to the server if the proof is deemed correct with respect to the contract specification. This way, the burden of proving compliance is shifted to the sender, who has to compute and produce a “compelling” piece of evidence that the receiver can then simply verify.

For such a scheme to work, however, three basic requirements must be fulfilled:
1) The proof must be equivalent to the monitoring computation—that is, the proof should be judged correct if and only if the message that accompanies it follows the contract.
2) The proof must be unspoofable. This means that any arbitrary proof, aimed at fooling the receiver into accepting a bad message, should be detected as such.
3) Checking the proof should be tractable. It is generally accepted that a “tractable” algorithm runs in time polynomial with respect to the size of its input.

Remark that nothing is said about the hardness of building the proof; this part is delegated to the sender of a message. Obviously, the interest of this method is for the “verifying” part ($\mu$ and $\nu$) to be easier than the “proving” part ($\gamma$).

The tractability of checking the result of that computation corresponds to a familiar concept, that of the NP complexity class. Formally, a decision problem $P$ is in the NP complexity class if, for a potential solution $x$, it can be verified that $P(x)$ holds in time polynomial to the size of $x$ [11]. From this observation, it follows directly that:

**Theorem 1.** Cooperative runtime monitoring requires that both $\mu$ and $\nu$ be in NP.

This principle might seem paradoxical, since any error in the client implementation resulting in an invalid continuation of the current message exchange should be intercepted by $\gamma$, which could not produce a proof of compliance and hence not send the message to the server. Actually, the inclusion of the proof acts as a deterrent: it coerces clients to check their messages and provides ground for the server to reject them in the presence of a faulty (or nonexistent) proof.

III. A FORMAL MODEL FOR COOPERATIVE RUNTIME MONITORING

Based on the informal model described above, we devise a formal model for cooperative runtime monitoring based on Linear Temporal Logic (LTL). LTL is an extension of classical propositional logic that expresses properties over message traces. Many major model checking tools such as SPIN [12] verify temporal formulæ expressed in LTL. Other approaches for specifying contracts include star-free regular expressions, Hennessy-Milner Logic, $\mu$-calculus, PSL [13], LTL-FO [14], and LTL-FO$^+$ [9], DecSerFlow [15], Logscope [16], Eagle and RuleR [17], which all subsume LTL. The reader is referred to [18] for a deeper coverage of LTL and other temporal logics.

A. Linear Temporal Logic

A message trace $m_0m_1 \ldots$, noted $\pi$, represents a sequence of incoming and outgoing messages at a peer’s interface over a period of time. The basic building blocks of LTL formulæ are propositional variables $p, q, \ldots$, expressing
Boolean conditions on particular messages of this trace. More precisely, in the present context, each propositional variable stands for a simple XPath expression denoting a particular path inside the XML document. The expression evaluates to true if the path can be found in the current message, and to false otherwise. For example, the expression /CartCreate/Cart/CartID/123 is true on a CartCreate message whose CartID element, nested inside a Cart element, contains the value “123”.

On top of these propositional variables, LTL allows Boolean connectives ∨ (or), ∧ (and), ¬ (not), bearing their usual meaning and temporal operators to express constraints on the sequence of messages. The temporal operator G means “globally”; the formula G ϕ means that formula ϕ is true in every message of the trace, starting from the current message. The operator F means “eventually”; the formula F ϕ is true if ϕ holds for some future message of the trace. The operator X means “next”; it is true whenever ϕ holds in the next message of the trace. Finally, the U operator means “until”; the formula ϕ U ψ is true if ϕ holds for all messages until some message satisfies ψ. A finite trace m satisfies a formula ϕ if it can be extended into an infinite trace m′ that satisfies ϕ.

Using this language, the properties described in Section II-A can be formalized as LTL formulae. For example, property P1 becomes ¬/(/CartModify) U/CartCreate. This formula states that, until some message contains a “CartModify” element, a message cannot contain a “CartModify” element. This is indeed equivalent to the first part of property P1; a similar formula can be written to state the same condition with CartAdd and CartRemove messages.

In the same way, a translation of P2 is ¬/(/CartModify) U/CartCreate. Similarly, P3 becomes G (¬/CartClear ∨ (¬/CartAdd U/CartCreate)). The reader is referred to [9], [10] for more formalizations of message properties in LTL, and to [4] for further examples of temporal message properties in actual web services.

B. Classical LTL Runtime Monitoring

The runtime monitoring of an LTL formula over some message trace m consists in processing each message one at a time, and outputting an intermediate result regarding the validity of this trace relative to the formula. This should be contrasted to offline methods such as [19], which analyze a complete trace without producing intermediate results after each message.

An “on-the-fly” runtime monitoring algorithm for LTL has been developed by Gerth et al. [20], and implemented as a string-re-writing algorithm in the Maude language by Rosu and Havelund [21]. The algorithm works as follows: the formula ϕ to monitor is first placed into the right-hand side of a node of the form G ϕ. Intuitively, the left part of the node represents the formulae that must be true in the current message, and the right part represents the formulæ that must be true in the next message to be read. Since no message has yet been processed, the specification goes into the right-hand side.

Once a message must be processed, the right-hand side of the last generated node is shifted to the left-hand side, which is then decomposed using the rules described in Figure 3. These rules successively break down the formula on the left-hand side into a list of smaller formulæ.

Some of these rules send some contents to the right-hand side in the process, such as the decomposition for the X operator; indeed, for X ϕ to be true in the current message, then ϕ must be true in the next message. Similarly, if G ϕ is true in the current message, this means both that ϕ is true in the current message (ϕ therefore appears on the left-hand side), and G ϕ must still be true in the next message (and G ϕ appears on the right-hand side of the node). Finally, some decompositions introduce two children; hence F ϕ can hold if ϕ is true in the current message, or otherwise if F ϕ holds in the next. The decomposition rules for other operators follow the same intuitive approach.

No further decomposition can be applied when each formula has been broken down into individual propositional variables, or negations of variables. These “atoms” are then evaluated against the current message (they evaluate to true
or false, depending on the XPath expression they stand for, and whether they are negated or not). An atom evaluating to false immediately spawns the node ⊥, while an atom evaluating to true is simply removed from the node. The current message is a valid continuation of the trace if this decomposition spawns at least one non-⊥ leaf. In such a case, the set of such non-⊥ leaves is then taken as the roots of new decomposition trees, awaiting for the next message, and the process starts over. A more detailed description of this algorithm can be found in [9].

Figure 4 shows a sample decomposition for a message where \( p \) is true, \( q \) is true and \( s \) is false. One can see that this message is a valid continuation of the trace from state \( ⊩ G (p ∧ (X q ∨ F s)) \), since the resulting decomposition spawns at least one leaf node which is not ⊥.

C. Theoretical Consequences

The overall complexity of the previous algorithm has been established:

**Theorem 2** (From [22]). LTL runtime monitoring is PSPACE-complete.

The combination of this result with Theorem 1 leads to a result with important consequences:

**Theorem 3.** If LTL monitoring can be used for cooperative runtime monitoring, then \( P = NP \).

**Proof:** By Theorem 1, the cooperative monitoring scheme imposes that verifying that a trace of messages complies with an LTL specification must be an NP problem. But by Theorem 2, LTL monitoring is PSPACE-complete. Since \( P \subseteq NP \subseteq PSPACE \), these two facts can only be reconciled if \( P = NP \).

Since the conclusion of Corollary 3 is deemed very unlikely (it actually requires that three complexity classes collapse into one), we can safely consider the problem of cooperative LTL monitoring impossible. More precisely, any method of checking that some evidence provided by the sender follows the contract will be as complex for the server as if it simply monitored the whole conversation by itself. Hence, “outsourcing” the monitoring computation to the client offers no gain unless it can be trusted.

It is important to note that this corollary applies not only to LTL, but to any specification language at least as expressive as LTL. This includes all the languages mentioned at the beginning of this section.

IV. COOPERATIVE RUNTIME MONITORING IN LTL

To use LTL for cooperative runtime monitoring, a compromise must therefore be made regarding the input language. In particular, LTL can be restricted to a subset of its possible formulæ. Such a fragment, if well chosen, could then ensure us that the decompositions produced by the on-the-fly algorithm can always be checked in polynomial time. In this section, we survey this approach.

The definition of the \( μ \) function is straightforward. It consists of checking that each propositional variable has
the proper truth value with respect to the message sent. Since each propositional variable stands for a simple path in the current message, which is either present or not, we can safely assume that checking this property is polynomial with respect to message length.

A. Proofs in LTL

First, we must define what constitutes a proof that a message is a valid continuation of a trace, according to some LTL formula. In Section III-B, we have shown an algorithm which, from a given state and a set of propositional variables, produces a derivation tree. The message is a valid continuation if this tree contains at least one leaf that is not \( \bot \), as shown in Figure 4.

Then, the sequence of decomposition rules, applied from the start state and leading to the non-\( \bot \) leaves, can be seen as witnesses that the current message is valid. In Figure 4, there are two such witnesses: the paths from the root to the leftmost and rightmost leaves, respectively. These witnesses can be described in a shorthand notation, by simply giving the sequence of decomposition rules applied at each derivation step. For example, the transition from the root node to its immediate child is obtained through the decomposition rule for the \( G \) operator; similarly, the last transition leading to the rightmost leaf is obtained by taking the right-hand side of the decomposition rule for the \( F \) operator.

Using this notation, one can succinctly describe the two witnesses in Figure 4 as follows:

\[
\begin{align*}
G, \land, p, \lor_1, X : & \{ q, G (p \land (X q \lor F s)) \} \\
G, \land, p, \lor_2, F_2 : & \{ F s, G (p \land (X q \lor F s)) \}
\end{align*}
\]

Each witness has a length in linear proportion with the size of the original formula; this is true since, at any derivation step, at least one symbol of the original formula is removed to produce one of the children nodes.

From a known start state, the receiver can easily take one of the witnesses, compute the sequence of derivation rules and check that the resulting end state corresponds to that sequence of derivations. One can devise a polynomial-time algorithm \( \nu \) which, given a start state and a sequence of derivation rules, checks that sequence of derivations can indeed be carried to the end. This procedure has the advantage that “dead-end” branches need not be expanded by the receiver. It therefore presents the potential of making the verification of a given derivation simpler than its computation, in the case where proportionally few branches end up in non-\( \bot \) states compared to the size of the whole derivation tree.

Note that in this context, a malicious client can still tamper with a proof before sending it to the server. However, since the server checks that proof against the message, it can be shown that any tampering is either detected by the function \( \mu \), or does not change the outcome of function \( \nu \). That is, one cannot create a “fake proof” that fools the receiver in accepting a message that it would otherwise reject. The demonstration of this claim is omitted due to lack of space.

B. An NP-Complete Fragment of LTL

In some situations, the number of branches to develop might amount to a significant proportion of the derivation tree, and even to the whole tree in the case where none of the branches ends up with \( \bot \). In such a case, checking the proof amounts to reproducing the whole derivation on the receiver side. This is consequent with Theorem 3, which predicts that a proof that a message complies with an LTL formula can, in some cases, be as long as its computation. As a first workaround to this consequence, we are interested in restricting the input language to a subset of LTL formulae where a derivation is easier to check than to compute. More precisely, one wants to find fragments of LTL that are in NP, in line with Theorem 1.

To this end, we introduce a new subset of LTL, called the non-branching fragment. We call a formula operator-free when it does not contain any temporal operator. We then define non-branching LTL as follows:

**Definition 1.** An LTL formula \( \varphi \) is non-branching if it follows these rules:

1) In \( \varphi \land \psi \), one of \( \varphi \) or \( \psi \) is operator-free
2) In \( F \varphi \), \( \varphi \) is operator-free
3) In \( \varphi U \psi \) and \( \varphi V \psi \), both \( \varphi \) and \( \psi \) are operator-free

The next theorem shows that a non-branching formula always produces a proof that can be checked in polynomial time.

**Theorem 4.** Let \( \varphi \) be a non-branching LTL formula, and \( c \) be the proof obtained from the on-the-fly decomposition algorithm for some message \( m \). The length of \( c \) is linear in the length of \( \varphi \).

**Proof:** The proof is done in two steps. We first show that the derivation of \( \varphi \) produces a single branch for any message \( m \). Then we show that the resulting leaf node contains only non-branching LTL formulae.

Step 1: we proceed to show that the derivation of \( \varphi \) does not produce any branching. We first remark that the derivation of any operator-free formula \( \psi \) does not produce branching. Indeed, in such a case, \( \psi \) is a propositional formula whose truth value is completely determined by the contents of the current message. Its derivation will produce branches that will all end either with the \( \bot \) node, or with a node of the form \( \emptyset \models \Delta \), with an empty left side. In the first case, this branch is not a witness and does not need to be included to the proof. In the second case, \( \varphi \) is true, and any one such branch can be chosen as the witness.

\[\text{Remark that all witnesses must be kept, since the validity of the next message requires that any one of them spawns a non-\( \bot \) node in the next round of decomposition.}\]
If the current operator to decompose is $\mathit{G}$, $\mathit{X}$ or $\land$, then no branching ever happens. We study the remaining operators one by one:

- $(\lor)$: since $\varphi$ is non-branching, then $\varphi = \psi \lor \psi'$ and we can suppose, without loss of generality, that $\psi$ is operator-free. By the previous remark, $\psi$ does not produce any branching.
- $(\mathit{F} \varphi)$: by definition, $\varphi$ is operator-free. Therefore, the left-hand side of the decomposition rule for $\mathit{F}$ is a propositional formula; it does not produce any branching. Only if all the nodes produced by this side of the decomposition end up with $\bot$ can the right-hand side of the $\mathit{F}$ rule be expanded. In such a case, the whole left side can be discarded, and again no branching is produced.
- $(\varphi \mathit{U} \psi)$: similarly to the previous cases, one can see that the decomposition of $\mathit{U}$ branches into two nodes with formulæ that are, by definition, operator-free. By the above remark, any one branch ending with a non-$\bot$ node is an appropriate witness, and all others can be discarded. A similar reasoning can be applied for $\mathit{V}$.

Step 2: by simple inspection of all decomposition rules, one can see that if the topmost formula is non-branching, then all decomposed subformulæ are also non-branching.

As an example, the formula $\mathit{G} \neg(p \to \mathit{F} q)$ is a non-branching formula; however, the formula $\mathit{G}(p \land (\mathit{X} q \lor \mathit{F} s))$ is not a non-branching formula. Indeed, Figure 4 shows that some of its proofs produce more than one branch.

C. Relative Strength of Non-Branching LTL

The non-branching condition is very strong. If forces any derivation in a trace to produce at most one witness, regardless of the total size of the derivation tree. By a previous remark, since a witness is itself linear in the size of the original formula, polynomiality of the proof checking algorithm is guaranteed.

One should first realize that this fragment indeed restricts the expressiveness of interface contracts. For example, the assertion “eventually, some property $p$ will hold forever” translates into the LTL formula $\mathit{F} \mathit{G} p$, which lies outside the non-branching fragment. In general, any specification requiring the monitor to “pick” a future state and evaluate some non-trivial condition on it will produce branching. Yet, it should be noted that all three properties for the Amazon E-Commerce service are non-branching LTL formulæ. Moreover, as strong as such a condition may be, it should be contrasted with other fragments of LTL amenable to cooperative runtime monitoring.

1) Simple Subset of PSL: The Property Specification Language (PSL) is a formalism that has been developed from IBM’s Sugar language and made into an IEEE Standard [23]. It can be used to complement other specification languages such as VHDL with a “temporal layer” based on LTL. PSL is a rich language that extends LTL with a limited form of quantification, in addition to the use of regular expressions to specify traces of events.

However, the richness of this language makes it prone to the development of complex and confusing expressions. The simple subset of PSL is a fragment restricting the form of possible expressions [23, Section 4.4.4]. Intuitively, the simple subset imposes a linear flow of time in the evaluation of a temporal property. Therefore, if one needs to evaluate a subformula at some time $T$, then the value of anything at the right of this subformula need not be known before time $T$. It has been advocated that the properties of a system that need to be checked at runtime be in the simple subset of PSL.

To achieve this result, restrictions are imposed on the use of Boolean and temporal operators, as formalized in Table I. In this table, “Boolean” is equivalent to operator-free, and the sequence type is equivalent to a construct of the form $\varphi_0 \land \mathit{X} (\varphi_1 \land \mathit{X} (\cdots \land \varphi_n))$ where all the $\varphi_i$ are operator-free. The expression $p \mathit{before} q$ is equivalent to the LTL expression $\neg q \mathit{W} p$.

The resemblance between non-branching LTL and the simple subset of PSL is striking. One can see that some of the conditions are identical to the non-branching fragment of LTL defined above; however, a few are even stronger. For example, the non-branching formula $\mathit{G} \neg(p \to \mathit{F} q)$ translates in PSL as $\mathit{never \ (p \to \mathit{next} \ q)}$, which is not in the simple subset of PSL.

2) Other NP-complete Fragments: One of the advantages of using a logic-based formalism is that many complexity problems have already been thoroughly studied from a formal perspective. Demri and Schnoebelen [24] provide a complete characterization of complexity results for satisfiability of all fragments of LTL. They define the notation $L^k_n(S)$, where $S$ is a subset of $\{\mathit{X}, \mathit{F}, \mathit{G}, \mathit{U}\}$, to denote the set of LTL formulæ obtained by using only Boolean connectives, temporal operators in $S$, $n$ propositional variables and at most $k$ nested temporal operators ($L(S)$ denotes an unlimited nesting and number of variables). The fragments of LTL that are in NP (or a lower complexity class) are:

- $L()$, $L(F)$, $L(X)$
- $L^1_\infty(U), L^1_\infty(F, X), L^1_\infty(U, X)$

<table>
<thead>
<tr>
<th>PSL Operator</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$</td>
<td>Operand must be Boolean</td>
</tr>
<tr>
<td>$\mathit{eventually!}$</td>
<td>Operand must be Boolean or sequence</td>
</tr>
<tr>
<td>$\lor$</td>
<td>One operand must be Boolean or sequence</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Left-hand side must be Boolean</td>
</tr>
<tr>
<td>$\mathit{until}, \mathit{until!}$</td>
<td>Both operands must be Boolean</td>
</tr>
<tr>
<td>$\mathit{before}, \mathit{until!}$</td>
<td>Both operands must be Boolean</td>
</tr>
<tr>
<td>$\mathit{next}$</td>
<td>Operand must be Boolean</td>
</tr>
</tbody>
</table>

Table 1: The Simple Subset of PSL, as Defined in [13]
Any other combination of operators or temporal depth is PSPACE-complete.

One can see that these fragments, compared to non-branching LTL, are relatively unattractive: most of them disallow the nesting of temporal operators, or impose the use of a single propositional variable. Neither is sufficient to express the Amazon ECS properties described in Section II-A.

V. EXPERIMENTAL RESULTS

To demonstrate the interest of cooperative runtime monitoring, a proof-of-concept implementation of this principle has been developed as a pair of Java applications, and tested on the example described in Section II-A. This section summarizes initial findings.

A. Implementation

We implemented two components, intended to operate on the client and server side respectively, as shown in Figure 2.

The first component is a runtime prover (RP) which, given an LTL specification, updates its state at each message sent and received, computes a proof for each message sent, and attaches it to the message. This is the implementation of function $\gamma$ in Figure 2.

The prover is based on BeepBeep [25], a lightweight runtime monitor that integrates seamlessly to any Ajax web application and intercepts its incoming and outgoing SOAP messages. Each message is first routed through the runtime monitor, which makes sure that it is a valid continuation of the current message trace. If this is the case, a proof of this fact is generated, and added to the SOAP Header element in XML format, as is shown in Figure 5. The CRM:proof tag is a custom element that will be ignored by any application other than our CRM tool. It encodes each branch of an LTL proof as a series of $\text{op}$ tags, giving the chain of decomposition rules used to obtain the end state, represented as an LTL formula. If the input formula is non-branching, by Theorem 4 we are sure that only one branch element will be present in the header.

The second component is a runtime checker (RC) which, given an LTL specification, a message and a proof produced by the RP, checks that the proof is valid. It first makes sure that any atomic $\text{op}$ element (a simple XPath expression) can actually be found in the SOAP-Body. This first step ensures that the proof indeed applies to the current message, and is not spoofed. Once this is done, the checker takes its current internal state, and checks that the sequence of decompositions given in the CRM:Proof element can indeed be applied to this state. It simply starts from the current state and updates it according to the sequence of derivations given in the proof. If the whole chain of decompositions can be carried to the end, the proof is deemed valid; the message is stripped of its proof, relayed to the application, and the contents of the state element is adopted as the new internal state. This corresponds to the implementation of functions $\mu$ and $\nu$ in Figure 2.

To simplify the implementation, the runtime prover and checker are actually the same application, which is instructed to run either as the client or the server side. It consists of roughly 50 kb of compiled Java code.

B. Results and Discussion

To test the approach, we used a sample Ajax application for the Amazon E-Commerce Service, and generated traces of request-response messages between the application and the web service. The specification to be monitored consisted of the conjunction of all LTL properties described in Section III-A. The messages were successively processed by the runtime prover, which appended a (linear) proof of their validity with respect to this specification, and the runtime checker, which read and checked the proof as described previously. Since we (obviously) did not have disk access to the Amazon server itself, we could not install the runtime checker on the server side; we instead simulated processing on the server side by running the runtime checker on the same computer as the client. To simplify the analysis, we excluded the serialization into XML that the prover would normally do, which would then be cancelled by a de-serialization of the XML by the runtime checker. The proof was instead transferred from prover to checker in its symbolic representation as a Java object.

For each message, we computed the running time required by the prover (representing the computing load on the client) and compared it to the number of nodes analyzed during the checking step (representing the computing load on the server). The difference between the two represents the

\[ \text{Figure 5. A SOAP message, to which an “XML-ized” proof that it is valid} \]

\[ \text{a valid continuation of the current trace has been appended in the SOAP header.} \]

\[ \text{and the contents of the state element is adopted as the new internal state. This corresponds to the implementation of functions } \mu \text{ and } \nu \text{ in Figure 2.} \]

\[ \text{To simplify the implementation, the runtime prover and checker are actually the same application, which is instructed to run either as the client or the server side. It consists of roughly 50 kb of compiled Java code.} \]

\[ \text{B. Results and Discussion} \]

\[ \text{To test the approach, we used a sample Ajax application for} \]

\[ \text{the Amazon E-Commerce Service, and generated traces of request-response messages between the application and the web service.} \]

\[ \text{The specification to be monitored consisted of the conjunction of all LTL properties described in Section III-A. The messages were successively processed by the runtime prover, which appended a (linear) proof of their validity with respect to this specification, and the runtime checker, which read and checked the proof as described previously. Since we (obviously) did not have disk access to the Amazon server itself, we could not install the runtime checker on the server side; we instead simulated processing on the server side by running the runtime checker on the same computer as the client. To simplify the analysis, we excluded the serialization into XML that the prover would normally do, which would then be cancelled by a de-serialization of the XML by the runtime checker. The proof was instead transferred from prover to checker in its symbolic representation as a Java object.} \]

\[ \text{For each message, we computed the running time required by the prover (representing the computing load on the client) and compared it to the number of nodes analyzed during the checking step (representing the computing load on the server). The difference between the two represents the}\]
amount of computing saved by the server through cooperative runtime monitoring vs. classical, server-side monitoring. The results are summarized in Figure 6.

One can first see that the running times for the runtime prover are slightly longer than those obtained for the same kind of traces by the original BeepBeep monitor in an earlier study [25]; the increased execution time is caused by the additional proof generation step that the original monitor did not do. However, once this proof is computed, checking that proof on the server side is much simpler, which translates into execution times about 15 times shorter.

From these results, one can draw the following conclusions. First, the server is assured of the client’s compliance with the contract. This is a direct consequence of the use of runtime monitoring, be it collaborative or not. Second, for the server, this compliance is enforced for a fraction of the computing cost of verifying it by itself. This fraction averages 7% for the traces we generated.

VI. RELATED WORK

A distinction must first be made between the cooperative runtime monitoring introduced in this paper and runtime monitoring of distributed systems such as CORBA [26]. The latter requires a centralized observer, external to the agents involved in a communication and that receives all the information monitored from all parts of the application. In the same way, the notion of cooperative data management introduced by [27] also requires a trusted third party, which is used as a witness for a transaction between two peers. This trusted party records the sequences of events exchanged between the peers, which can then refer to it to settle disagreements. In contrast, in cooperative runtime monitoring, no external third party is required, nor any trust assumed from any external agent. Moreover, the approach suggested in both papers records the messages, but is “neutral”: it does not actively interfere in the actual message exchange, even in the case of a violation, since no particular specification is given to it.

Cooperative runtime monitoring draws natural parallels with proof-carrying code (PCC) [28]. PCC suggests that compiled programs be accompanied by a “proof” of their correctness that an execution environment could then easily check before allowing it to run. A malicious or tampered program could then be detected. For example, a compiler can produce Java code, accompanied with a proof that the compiled program is memory safe. While this idea has been used to statically prove that a program follows a set of requirements (mostly memory safety) beforehand, our approach rather provides runtime proofs that individual messages produced by a program follow some contract. The program itself is not checked, signed or validated in any way. As far as we know, the present paper is the first application of this idea to individual messages produced at runtime by a program.

CRM is actually closer to the classical idea of a “token” or “hash value” that is used to ensure the integrity of a message, which has become common practice in the field of computer security [29]. However, while traditional hashes ensure the integrity of a particular message individually, the token produced by CRM provides integrity assurance relative to some contract. A message can still be tampered, but not in a way that would constitute a violation of the particular contract being monitored. In addition this contract involves a sequence of messages, defined by some LTL formula, and their particular position in that sequence; traditional hashing ensures the integrity of messages individually.

Finally, the parallelization of LTL has already been studied by [30] as an alternate way of sharing the verification burden between multiple machines. However, the approach described in the paper assumes that the path is completely known when the analysis starts; different machines take care of different parts of that trace and combine their results. Therefore, it cannot be applied directly for runtime monitoring.

VII. CONCLUSION

In this paper, we introduced the notion of cooperative runtime monitoring. We showed how this concept can be used to shift the computing load of contract compliance from the server to the client of an application, while still maintaining the same compliance guarantees for the server, and yet without any trust assumptions about the client. Our initial experimental results show that indeed, having the sender of a message compute a proof of compliance that can easily be checked by the receiver can reduce the work required on the server side by a large fraction.

We have also shown how seemingly unrelated specification languages, such as the simple subset of PSL, actually become natural consequences of CRM constraints over Linear Temporal Logic.

Preliminary results on CRM look very promising, and suggest that the approach could become a fruitful alternative to enforce sequential patterns in message-based interactions such as web services. Future work includes taking into account data elements inside messages in addition to the sequence of such messages. This would involve devising a cooperative runtime monitoring framework for a more expressive logic than LTL, such as LTL-FO or LTL-FO⁺.

REFERENCES

[1] G. Hughes, T. Bultan, and M. Alkhalaf, “Client and server verification for web services using interface grammars,” in


