Causality in Message-Based Contract Violations: A Temporal Logic “Whodunit”

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

Abstract—Interface contracts are sets of constraints specifying valid exchanges of messages between two or more peers. A contract violation occurs when one of the peers fails to fulfill one of these constraints and emits a message that is not a valid continuation of a message “trace”. In some cases, the message that directly exposes the violation turns out to be the last of a succession of forced moves, while the “root cause” of the violation resides earlier in the trace and may emanate from a different peer. We formally define the notion of causality for interface contracts expressed in a first-order extension of Linear Temporal Logic. In particular, we show how the detection of root causes reduces to satisfiability solving of a precise set of formulæ. An experimental setup shows how causality can be analyzed automatically on a pre-recorded message trace.

Keywords—compliance checking; temporal logic; distributed computing; web services

I. INTRODUCTION

The interaction of multiple, loosely-coupled components in a system can generally be understood by studying their communication through data units called messages. Each component might have its individual requirements on the way it can be invoked, defining the interface it exposes to potential peers. The combination of individual requirements creates a form of contract, called an interface contract, defining valid message exchanges. For example, for communicating processes interacting through message channels in an operating system, a valid exchange may represent a sequence of operations that is free from errors and deadlocks. For business processes implemented in the form of web services, a contract may encode in a formal notation legal requirements on each partner of a transaction.

However, while extensive work concentrates on the discovery of contract violations, much less work has been done to explain them, and in particular determine their exact cause. The problem is especially relevant in a legal setting, where the fact that a violation occurred alone is not of much use. Compensation mechanisms must be initiated by the agent ultimately responsible for the fault, and hence a precise notion of causality must be defined, accompanied by efficient mechanisms to pinpoint the source of eventual violations.

At first glance, finding the cause of a contract violation is a straightforward process. One simply processes the messages one by one, until one of them is of a type (or contains data) that is not expected, given the current state of the conversation. Indeed, these kinds of direct violations can be detected with reasonable effort, using existing techniques that rely on trace analysis or runtime monitoring.

Alas, not all violations can be detected in this way. In some cases, the message that directly exposes the violation turns out to be the last of a succession of forced moves, while the actual cause of the violation resides earlier in the trace and may emanate from a different peer. Consider a peer giving two conditions to its partner: 1) whenever I send A, you should reply by B and 2) you should never reply B. If the peer sends A, it puts its partner into the impossible position of having to reply both B and not B. Any message sent by B will be a direct violation of the contract —but is it really B’s fault? Causality analysis, whose task is to find the root cause of a contract violation, can hence be likened to a form of formal “whodunit”.

The previous example is arguably contrived; yet, the situation becomes very plausible when the interface contract becomes the combination of independent requirements coming from all the peers involved in a message exchange: one can simply consider the scenario where constraint 1 above is given by A, while constraint 2 comes from B’s own requirements. Inconsistencies that were absent from each individual contract may become possible once combined.

In this paper, we formally define the notion of root cause for a violation, when interface contracts are expressed in a first-order extension of Linear Temporal Logic called LTL-FO+. In particular, we show in Section III how detection of non-compliance reduces to satisfiability solving of a particular set of formulæ. In Section IV, an experimental setup describes a proof-of-concept trace analyzer combining an LTL-FO+ runtime monitor with a state-of-the-art satisfiability solver for first-order temporal logic. Initial results show how causality can be analyzed automatically on a pre-recorded message trace with reasonable overhead compared to the traditional analysis of direct violations, and more importantly, detect errors earlier.

1who-dun-it: a detective or mystery story
II. Causality in Temporal Contracts

In this paper, we consider interface contracts where multiple processes, called peers, interact through the exchange of structured pieces of data called messages. Each of these messages can have a name, and one or more data elements carrying values. Without loss of generality, we assume that the messages can be represented under a tree form as small XML documents.

A. Interface Contracts

To illustrate the paper’s concepts, we first show a few examples of contracts. They will be only succinctly presented, as the topic has been the focus of earlier papers, to which the reader will be referred for more details.

1) A Playful Example: We consider as a first, light-hearted example the simple game called tic-tac-toe, which alternates two players placing one of their symbols (X or O) on a 3×3 grid. The game can be seen as a form of interaction involving two services whose messages consist of each player’s move:

```xml
<Move>
  <Player>X</Player>
  <Col>2</Col>
  <Row>Λ</Row>
</Move>
```

The message “protocol” simply consists of all the constraints specifying a valid game of tic-tac-toe:

T1. Players alternate at each move 
T2. No move can involve the same row-column pair 
T3. The same player putting three symbols in the same row, the same column or in a diagonal wins 
T4. After a winning move, the next message must announce the winner, and no further message is allowed.

The interest of this protocol is that it encapsulates a number of properties found in real-world interactions. First, a valid protocol involves sequences of correlated messages. Second, these correlations involve parameters inside messages, in such a way that data values found in one message condition what values (and what messages) can be seen in the remainder of the exchange. Finally, the peers involved in that protocol must each record and update a representation of the current state of the interaction. These constraints have been labelled “data-aware” in previous work.

2) Shopping Transactions: A more serious example involves a buyer and a seller engaging in commercial transactions through web services. A list of items can be queried from the seller, using an ItemSearch message specifying some Keyword. The returned list of items provides an item ID for each. A shopping cart can then be created, through the CartCreate message. When a cart is created, the client must specify the intended payment method, although this can be changed afterwards using the CartModify message. The returned Cart message summarizes the payment method, cart ID and list of item IDs it contains. This state can be queried at any time by issuing a CartGet message specifying the ID for the cart.

Each cart created during a session must be checked out (using CartCheckout). The checkout message must provide the chosen payment method for the transaction and the cart ID. A session ends by issuing a Logout message.

Each of these peers has specific requirements on the way it can interact with the other side. For example, on the seller’s part, one might assume the following constraints:

S1. All carts involving more than three items have the additional data field “large”. Large carts must be paid with a credit card.
S2. Every cart created must be checked out.
S3. Payment mode must be exactly one of “PayPal” or “Credit”.

One can also imagine an additional requirement on the buyer side:

B1. A cart created with a mode of payment must conclude with the same mode of payment.

A complete contract would include additional ordering and data constraints; they have been left out to keep the scope on the type of violations focussed by the present paper.

B. Linear Temporal Logic With Data: LTL-FO⁺

For these contracts to be processed and analyzed in an automated fashion, a formal representation of their requirements must be provided. In the present case, requirements are assertions on the sequence and content of messages. A trace of messages π is simply a sequence m₀, m₁, ..., such that every mᵢ is a message. We shall write π|ᵢ to denote the suffix of π starting at message mᵢ. Assertions over message traces are expressed in an extension of Linear Temporal Logic, called LTL-FO⁺, which has been shown to be appropriate for the modelling of such assertions. The logic is briefly presented and the reader is referred to earlier work for an in-depth coverage [2].

LTL-FO⁺’s 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 ∨ (“and”), ∨ (“or”), ¬ (“not”) and → (“implies”), following their classical meaning. In addition, LTL temporal operators can be used. The temporal operator G means “globally”. For example, 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

²The reader is referred, for example, to a detailed study of constraints in the Amazon E-Commerce Service [1].
m |= α ⇔ m₀ = (a, ∗) and α = α
m |= p = v ⇔ m₀ = (a, ∗) and sₚ(p) = v
m |= ¬φ ⇔ m |= φ
m |= φ ∧ ψ ⇔ m |= φ and m |= ψ
m |= φ ∨ ψ ⇔ m |= φ or m |= ψ
m |= φ → ψ ⇔ m |= φ or m |= ψ
m |= Gφ ⇔ m₀ |= φ and m¹ |= Gφ
m |= Fφ ⇔ m₀ |= φ or m¹ |= Fφ
m |= Xφ ⇔ m₁ |= φ
m |= ∃x : φ ⇔ m |= φ[x/v] for some v ∈ m₀(π)

Table I

SEMANTICS OF LTL-FO⁺

We illustrate this by a simple example. Suppose each peer can choose from three messages named p, q and r. Peer A’s specification is: G (p → (X¬p)) (“a p is never followed by another p”), while Peer B’s specification is G (p → Gp) (“once p, always p”). A simple trace where B sends r, A sends p and B sends q does not comply with the global specification, since B sends a second p in a row. However, while B is the direct cause for the violation (by sending the second p), it was forced to send it because of A’s action (sending p). Had A decided to send anything else instead, B would have replied with p, q or r, and in any case the global contract would have not been violated. Therefore, although the last message of that trace revealed a violation, it was already bound to fail at A’s message. Intuitively, the cause of non-compliance should be traced back to this event, and the “blame” for the failure of the transaction be put on A instead of B.

Bauer et al. [3] study LTL and introduce what is called anticipatory semantics. A trace prefix is regarded as non-compliant as soon as it becomes impossible to extend it by any valid sequence of messages, even if no direct violation has yet been observed. This definition translates directly to LTL-FO⁺:

Definition 1 (Non-compliant prefix). Let π = s₀s₁...sᵢ be a trace prefix, and ϕ be an LTL-FO⁺ formula. We say π is

Finally, LTL-FO⁺ adds quantifiers that refer to parameter values inside messages. If π denotes a path in a message m, we note m(π) the set of of values at the end of that path inside m. For example, in the tic-tac-toe message shown earlier, if π = Move/Col, then m(π) = {2}. The expression ∃x : φ(x) states that in the current message m, there exists a value v ∈ m(π) such that φ(v) is true. Dually, the expression ∀x : φ(x) requires that φ(v) holds for all v ∈ m(π). When the context is clear, we abbreviate ∃x : φ(x) as ∃x : φ, stating that the (only) value at the end of path π is k. The semantics of LTL-FO⁺ are summarized in Table I, where m = m₀, m₁, ..., is a sequence of messages; the reader is referred to [2] for a deeper coverage of LTL-FO⁺ in a related context.

Using this logic, it is possible to convert the interface contract constraints shown previously into LTL-FO⁺ formulae. In the tic-tac-toe protocol, the requirement that players alternate can be formalized as follows:

G (∀/Move/Playerπp : (X∀/Move/Playerπp′ : p′ ≠ p))

This formula states that globally, whenever the player field of the move message has some value, then in the next message the same field must have a value p′ ≠ p. Similarly, a winning condition for player O goes as:

G (Move/Player/O → ∀/Move/Rowr₁):
X G (Move/Player/O → ∀/Move/Rowr₂ : r₁ = r₂ → X G (Move/Player/O → ∀/Move/Rowr₃ :

r₂ = r₃ → X/winners/Player/O)))

The formula simply states that seeing player O playing three times in the same row entails that the next message declares O as the winner. A similar formula can be written for columns and diagonals.

Properties of the shopping cart example can also be formalized using LTL-FO⁺. Here is for example a rendition of constraint B1 for credit card payment.

G (∀/Cart/CartID : /Cart/Payment/Credit → G (∀/Cart/CartIDi : i = i′ → /Cart/Payment/Credit))

We note m |= φ the fact that a trace m satisfies the condition expressed by formula φ. Otherwise, m is said to violate φ.

C. Non-Compliant Prefixes and Root Causes

At first glance, finding the cause for the violation of a formula is a straightforward process. One simply processes the messages one by one, until one of them is of a type or contains data that is not expected given the current state of the conversation. For property T1, a violation occurs when player O makes a second move in a row. For property B1, a violation occurs when a cart is checked out using a different payment mode than the one it was created with. In both cases, the faulty message is called a direct violation of the contract (or direct error). However, in some cases, the message that directly exposes the violation turns out to be the last of a succession of “forced moves”. This does not necessarily indicate that the violation of the contract is genuinely caused by the last event processed.

We illustrate this by a simple example. Suppose each peer can choose from three messages named p, q and r. Peer A’s specification is: G (p → (X¬p)) (“a p is never followed by another p”), while Peer B’s specification is G (p → Gp) (“once p, always p”). A simple trace where B sends r, A sends p and B sends q does not comply with the global specification, since B sends a second p in a row. However, while B is the direct cause for the violation (by sending the second p), it was forced to send it because of A’s action (sending p). Had A decided to send anything else instead, B would have replied with p, q or r, and in any case the global contract would have not been violated. Therefore, although the last message of that trace revealed a violation, it was already bound to fail at A’s message. Intuitively, the cause of non-compliance should be traced back to this event, and the “blame” for the failure of the transaction be put on A instead of B.

Bauer et al. [3] study LTL and introduce what is called anticipatory semantics. A trace prefix is regarded as non-compliant as soon as it becomes impossible to extend it by any valid sequence of messages, even if no direct violation has yet been observed. This definition translates directly to LTL-FO⁺:

Definition 1 (Non-compliant prefix). Let π = s₀s₁...sᵢ be a trace prefix, and ϕ be an LTL-FO⁺ formula. We say π is

3Operator V is the dual of U, i.e. ϕ Vψ ⇔ ¬(¬ϕ U ¬ψ).
a non-compliant prefix exactly when, for every continuation \( \sigma' = s_{i+1}s_{i+2} \ldots \), the trace \( \sigma\sigma' \) violates \( \varphi \).

Finding the root cause of a violation therefore amounts to finding the shortest non-compliant prefix of a trace. Since by definition, any shorter prefix is compliant, one can attribute the root cause to the last message of the prefix. The peer responsible for the violation is the sender of that message. We see how this definition coincides with our intuitive analysis of the situation, which put the blame on that same peer and that same message.

III. Causality Analysis in LTL-FO\(^+\)

The task of detecting root causes of contract violations is far from trivial, since the previous definition requires to determine at any point in time, of all the possible future traces, whether there exists at least one that can fulfil the formula.

To convince the reader of the difficulty of this problem, we revisit the tic-tac-toe protocol described in the previous section with the additional requirement that “O must always be the winner”. This can be easily formalized by stating that any trace ends with a \(<\text{Winner}>\) message whose Player field has value “O”, which, because of the other rules, can only happen if O effectively wins the game. A trace prefix should be identified as non-compliant whenever no matter what X and O play from now on, O can at best get a draw. Hence evaluating whether a message is a root cause against a set of LTL-FO\(^+\) formulæ is similar to proving the absence of a winning strategy in a particular “game”.

A. Possible Approaches

There exist a number of potential methods to search for root violations in a trace. We briefly present some solutions that appear feasible at first glance.

1) Labelled Büchi Automaton: One might first want to look at Bauer et al.’s solution for the simpler case of propositional Linear Temporal Logic, and try to extend it to LTL-FO\(^+\). The authors tackle the problem by generating a first Büchi automaton \( M_\varphi \) accepting the specification \( \varphi \). They then label a state \( s \) with \( \top \) if there exists a trace starting in \( s \) that visits accepting states infinitely often, and \( \bot \) otherwise. This labelling is computed by checking non-emptiness of the language accepted by the automaton, taking successively each state as the initial state. This process is linear in the size of the automaton, but the automaton itself is exponential in the size of the original formula.

This method provides a straightforward way to identify root causes. It suffices to read a trace, event by event, and to keep pointers in the automaton to each possible current state (since the automaton is non-deterministic) and take the appropriate transitions. Any pointer on a state labelled \( \bot \) can be discarded, since it cannot lead into any accepting sequence. An event is a root cause if it is the first that causes the deletion of all pointers.

2) Conversion to LTL: A second possibility is to translate an existing formula into LTL. Each quantifier \( \forall x : \varphi(x) \) must be replaced by the equivalent expression \( \varphi(b_1) \land \varphi(b_2) \land \cdots \land \varphi(b_n) \), for all possible values of \( b \) in a message (existential quantifiers are replaced with disjunctions of terms). We can then use a model checker to verify that a given trace satisfies the formula. The model checker is then fed with a finite-state machine that has only one possible execution trace, namely the one to verify.

At first, the model checker is given a finite state machine that contains only the first message of the trace to evaluate, which is connected to a universal graph, which does not give any condition to future states of the trace, and hence represents all possible continuations with all possible messages. If the LTL property returns false on this model, then no trace can extend this first message in a way that satisfies the original specification, and hence this message is non-compliant. Otherwise, the same process is repeated with the first two messages, and so on.

3) Conversion to First-Order Logic: A third possibility is to translate LTL-FO\(^+\) into a set of first-order logic sentences. Then, the use of a model finder such as Prover9/Mace [4] can be leveraged to determine whether the resulting system is consistent or contradictory. Since the solver’s answer is only true or false, an incremental approach of feeding progressively longer prefixes of the trace to the solver must be taken, as with the model checker solution.

B. Caveats to these solutions

There are multiple problems to these potential solutions. In the Büchi approach, the resulting automaton is exponential in the size of the formula. In order to pre-compute and label all states, the whole automaton must be generated and kept in memory. Moreover, since in LTL-FO\(^+\) the evaluation of the formula depends on data values that are observed in the current trace, the complete automaton cannot be pre-computed unless all the possible values are known in advance: this is called domain bounding. In addition to being impractical for realistic scenarios, this solution results in yet another exponential blow-up of the (already exponential) automaton, this time in the number of quantifiers in the original formula.

The conversion to LTL and use of a model checker also requires domains to be bounded when the LTL-FO\(^+\) formula is converted into LTL, and hence shares the same weakness as the automaton approach. Moreover, the repeated execution of the model checker on progressively longer traces wastes computing power, since the work done for the first \( n \) messages must be rebuilt from scratch when evaluating the trace of length \( n + 1 \).

Finally, the use of a first-order model finder presents the appeal of not having to bound the domains of the formula: the translation of LTL-FO\(^+\) into first-order logic yields a totally equivalent formula. However, this approach shares the
same problem as the model checker: the model finder can express the fact that no model exists that satisfies the given specification, but does not explain why. One must resort to providing incrementally longer traces in order to pinpoint the first non-compliant event.

C. An “On-The-Fly” Algorithm

There exists, however, an alternate algorithm that can already handle LTL-FO+ formulæ “natively”—that is, without translating them into an auxiliary notation. It does not suffer from the bounding of domains present in some of the previous approaches and has the upside of building its analysis of a new message on the intermediate results computed from the previous messages. The algorithm is only briefly recalled in this section; a detailed description can be found in previous work [2].

First, we define a watcher, which is a process that updates its internal state upon each message it is being fed.

**Definition 2** (Watcher). Let \( M \) be a set of messages. A watcher for a formula \( \varphi \) is a tuple \( \mathcal{W}_\varphi = \langle Q, q_0, \delta, O, f \rangle \) where:

- \( Q \) is a set of states;
- \( q_0 \in Q \) is the initial state;
- \( \delta : Q \times M \rightarrow Q \) is the transition or update function;
- \( O \) is a set of outcomes, i.e. the possible conclusions that a watcher can draw on a given trace;
- \( f : Q \rightarrow O \) is an outcome function.

In this algorithm, the watcher’s state is a set of nodes. Each node \( \hat{N} \) is of the form \( \Gamma \models \Delta \), where \( \Gamma \) is a set of LTL-FO+ formulæ that must be true in the current state, and \( \Delta \) is a set of LTL-FO+ formulæ that must be true in the next state.

The update function \( \delta \) simply takes each node \( \hat{N} \in q \), moves the contents of the right-hand side of the node to the left-hand side (leaving the right-hand side empty), and then calls an auxiliary function \( \text{UPDATE} \) on that resulting node. \( \text{UPDATE} \) takes a node and decomposes the formula from its left-hand side according to the rules shown in Figure 1. On some occasions, the decomposition of a formula produces more than one formula; the decomposition is then recursively repeated on each resulting node until no further rule applies. The set of these terminal, “spawned” nodes is then returned to \( \delta \) and included in the new state for the watcher. Termination is guaranteed by the fact that each rule decomposes at least one formula into formulæ with strictly less nested quantifiers.

It remains to determine how the watcher can conclude that a trace fulfils or violates a property. To this end, a set of three outcomes is used: \( \top \) indicates that the property is fulfilled, \( \bot \) indicates that the property is violated, and “?” indicates an inconclusive result: the property cannot be guaranteed to be neither true, nor false. The outcome function assigns to each possible watcher state one of these outcomes.

**Definition 3** (Outcome function). Let \( q \) be a watcher state, and \( O = \{ \top, \bot, ? \} \) be the set of outcomes. The outcome function \( f \) is defined as follows:

\[
f(q) = \begin{cases} 
\top & \text{if } \emptyset \models \emptyset \in q \\
\bot & \text{if } q = \emptyset \\
? & \text{otherwise}
\end{cases}
\]

The violation condition is straightforward: if a call to \( \delta \) produces no nodes, then there is no possible way for the trace to continue while still fulfilling the property, and a violation can be announced. On the contrary, if a node is empty, then no condition has to apply on the remainder of the trace, and hence the formula holds no matter what future messages are produced. If neither case occurs, then it is not possible to conclude yet, and the “?” symbol is returned.
D. Catching Violations

The previous algorithm, originally developed in [2], works “on-the-fly” and generates only the state space required for the particular trace that is being analyzed. Moreover, its analysis of messages one by one provides, without additional work, a partial outcome of any trace prefix that it reads. However, it suffers from one major defect: its outcome function is insufficient for detecting root causes.

This can be shown by the example in Section II-C. The global specification to watch is the conjunction of A’s and B’s requirements, yielding the start state of the monitor:

$$\emptyset \vdash G(p \rightarrow (X\neg p)), G(p \rightarrow Gp)$$

When B sends r, the watcher’s state does not change. Then A sends p, leading to this watcher state:

$$\emptyset \vdash G(p \rightarrow (X\neg p)), G(p \rightarrow Gp), Gp, \neg p$$

We know from our intuitive analysis that A’s message is the end of a non-compliant prefix. However, the outcome function $f$, applied to this state, returns “?”. —indeed, the node cannot be reduced any further using the simplification rules in Figure 1. To identify this message as a root cause, one must somehow determine whether some trace can extend the current prefix according to the initial specification.

We extend this algorithm by showing here that a property based on future paths can be translated into a property on the current state itself. This is possible thanks to our runtime monitoring algorithm, which keeps as node labels sets of LTL-FO+ formulæ representing requirements on future messages traces. For a set of $k$ monitor nodes $\{N_1,\ldots,N_k\}$ of the form $N_i = \Gamma_i \models \Delta_i$ ($0 \leq i \leq k$), we define an auxiliary function $\Psi$ as follows:

$$\Psi(\{N_1,\ldots,N_k\}) = \bigvee_{i=0}^{k} (\bigwedge \Gamma_i \land X (\bigwedge \Delta_i))$$

where $\bigwedge \Gamma_i$ (resp. $\bigwedge \Delta_i$) represents the conjunction of all formulæ contained in $\Gamma_i$ (resp. $\Delta_i$). By definition we take $\bigwedge \emptyset = \top$ and $\bigvee \emptyset = \bot$.

We then show how non-compliant prefixes can be characterized through properties of their state formulæ.

**Theorem 1.** Let $\delta$ be the runtime monitoring function described earlier, and $\pi = s_0s_1\ldots s_n$ be a trace prefix. Let $S = \{N_1,\ldots,N_k\}$ be the monitor state returned by $\delta(\pi)$. The trace prefix $\pi$ is non-compliant if and only if $\Psi(S)$ is unsatisfiable.

**Proof:** Let’s first show that, for a node $N = \Gamma \models \Delta$, if $\Psi(N)$ is a contradiction, then the application of any decomposition rule produces a node $N'$ such that $\Psi(N')$ is also a contradiction. We omit the details; this is done by taking each decomposition rule one by one and comparing the definition of $\Psi$ in the start and resulting nodes. This can then be generalized to a monitor state $S = \{N_1,\ldots,N_k\}$, since $\Psi(S) \equiv \bigvee_{i=0}^{k} \Psi(N_i)$; if each of the $\Psi(N_i)$ is a contradiction, then so is $\Psi(S)$. Hence, as soon as the watcher’s outcome is a contradiction, it is impossible that it ever becomes $\top$ in a future state, hence $S$ is non compliant.

Conversely, suppose $\Psi(S)$ is satisfiable. By definition of satisfiability, there exists an (infinite) trace $\pi'$ such that $\pi' \models \Psi(S)$. This trace is a continuation of $\pi$, which hence cannot be a non-compliant prefix.

Equipped with this result, it is now possible to define a new outcome function $f'$ that returns $\bot$ if all nodes in the watcher’s state are unsatisfiable. The other conditions remain identical to $f$. In the previous example, one can see that one of the conjuncts requires that the only valid message be $p$ from now on ($Gp$), while the next one requires that the next message not be $p$ ($\neg p$). Both cannot be true for the same trace, and hence $\Psi(N)$, the conjunction of all formulæ on the right-hand side of the node, is a contradiction. This is the first watcher node having such a property, which means that A sending $p$ is a root cause for the violation.

The problem of finding non-compliant events has hence been reduced to the problem of evaluating LTL-FO+ satisfiability on watcher states. As a matter of fact, the original outcome function $f$ defined in this paper is a simpler form of outcome function that recognizes $\Psi(\emptyset)$, but not other, more complex forms of contradictions such as the ones previously discussed. The outcome function based on satisfiability is hence a generalization of the original principle.

However, while the source of contradiction in the previous example is simple to identify ($Gp$ vs. $\neg p$), a contradiction is seldom that obvious. The problem now remains of providing a procedure that can compute LTL-FO+ satisfiability on arbitrary formulæ. In fact, for LTL alone, it has been shown that deciding satisfiability is PSPACE-complete [5]. Any fragment of first-order linear temporal logic, such as LTL-FO+, is at least in the (higher) EXPSPACE complexity class [6]. This should not be surprising, as the equivalent construction using Büchi automata also results in (double) exponential blow-up of the original formulæ.

IV. A Proof-of-Concept Implementation

To determine whether the potentially high complexity class of LTL-FO+ satisfiability is an obstacle to its use for causality detection, we devised a set of experiments on a proof-of-concept implementation of a trace analyzer.

A. Architecture

The LTL-FO+ validation algorithm described in Section III-C was implemented in earlier work as a Java runtime monitor called BeepBeep [2]. As we have seen, its basic outcome function only computes direct violations of a specification and cannot evaluate satisfiability of general formulæ.

To this end, the monitor has been modified to accommodate the use of an external tool. The closest suitable
applications available are two solvers for monodic first-order temporal logic, called TeMP [7] and TSPASS [8]. In monodic temporal logic, each subformula beginning with a temporal operator must have at most one free variable [6]. There is a priori no such restriction in LTL-FO⁺; fortunately, all the example formulæ shown in this paper belong to this fragment and can hence be handled by these solvers.

The working principle for causality analysis is shown in Figure 2. When a message is to be processed (1 in the figure), the UPDATE procedure is called, taking as input the message and the current state of the algorithm, made of a set of \( n \) nodes, each of the form \( \Gamma \models \Delta \). The result of the application of the procedure is a new set of \( k \) nodes (2).

Each of these nodes is then taken one by one; the formula it contains are joined together by conjunctions, and this compound formula is sent to the satisfiability solver “S”. If \( S \) declares the formula unsatisfiable, the node is deleted from the monitor’s state (3). Otherwise, it is kept (4). The remaining nodes after this pruning step constitute the new state of the monitor, waiting for the next message. By the previous results, if no node remains after pruning by the external solver, then the message is a root cause for non-compliance.

All the experiments have been made with the latest version (0.94) of the TSPASS solver. The link between the external solver and the monitor is done through the execution of an external executable file. The LTL-FO⁺ formula is transformed into a text string following the input format of TSPASS, piped through the solver’s standard input. TSPASS’s result is grabbed from the standard output and is checked for the presence of either the String “Satisfiable” or “Unsatisfiable”, yielding the corresponding verdict. TSPASS was modified so that any other text it normally prints to the standard output is disabled. Another slight modification was performed, which will be described below. Apart from these two changes, the solver was used as is.

B. Experiments

We generated 100 message traces for the shopping cart protocol described in Section II. The traces were randomly produced by choosing a new message to produce from the set of all actions, e.g. creating a new cart, changing payment mode for an existing cart, adding items to a cart, etc. Each of these traces was then processed by the tool chain shown above.

1) Overhead per Message: We first measured the overhead per message of using a full-fledged solver for LTL-FO⁺, instead of the simple, constant-time detection routine of the original algorithm. To this end, we ran the analyzer twice on each trace, using alternately the TSPASS solver or the internal emptiness detector. For each individual message, we measured the overhead, i.e. the ratio between processing time in each method.

Initial results based on elapsed time show a significant increase in processing time —by a factor of 14. While the simple monitor processes each message in an average of 5 ms, for the causality monitor these times jump at 80 ms per message. However, it was suspected early on that part of this lag was caused by the primitive way in which the Java monitor and command-line solver were chained. The solver was modified to display the elapsed system time in microseconds between the start and the end of its main loop (including parsing from standard input). For the same set of traces, the experiments were re-run, counting the elapsed time spent inside the solver. Figure 3 plots the resulting overhead.

This time, processing time drops to an average of 13 ms per message. It turns out that roughly 97% of all the overhead incurred by the use of a solver is wasted by the fact that the monitor and the solver are two separate applications that communicate through the command line. This is an issue that could easily be fixed in a dedicated setup where both the monitor and the solver use the same data structures and are executed under the same program.

We shall still emphasize the fact that in absolute values, the processing of each message hence takes much less than one tenth of a second. This is lower than the typical network latency from major service providers. Therefore, causality analysis of the shopping cart requirements would not cause a noticeable delay if it were performed at runtime on web services by monitoring each incoming and outgoing message.

Finally, for 12 of the 100 traces, the time per message for the TSPASS-based monitor is lower than for the simple monitor. This can be explained by the relatively short running times of the analysis process compared to the effective precision of the system clock.

---

4http://www.verizonbusiness.com/about/network/latency/
2) Early Error Detection: Although the use of a full-fledged solver incurs a time overhead, in counterpart it presents the advantage that that errors are detected at the earliest possible moment. Therefore, the TSPASS-based monitor should be able to stop analyzing a trace earlier than its simpler counterpart in some cases.

For the same traces as above, we also measured the number of messages consumed by each implementation. Each monitor was programmed to stop reading the trace as soon as the $\top$ or $\bot$ outcome was computed. Figure 4 plots the distribution of the difference in length between both methods.

In the 100 generated traces, the simple monitor consumed a total of 1,350 messages. The causality analyzer consumed 18% less, or 1,104 messages, before reaching the same conclusions. A large portion of the generated traces contained a direct violation; in such a case, both monitors return the same verdict at the same time, and hence the difference in length is zero. However, about a quarter of the traces contained non-compliance that could be detected before provoking any direct error. For some traces, the difference is very large, with the causality analyzer reaching a verdict as much as 20 messages ahead of the simple monitor.

Indeed, in some cases, the simple emptiness condition of the original algorithm can never conclude to an error and exhaust the entire trace without producing a conclusive outcome. This is the case, for example, with a formula of the form $Gp \land F\neg p$. No trace can satisfy this formula, yet the prefix $pp \ldots p$ does not cause any direct violation. While the TSPASS solver would detect the unsatisfiability and stop the analysis of the trace right away, the simple monitor would keep returning the “?” outcome and processes the entire trace.

V. RELATED WORK

The present work draws many parallels to existing solutions related to trace validation and runtime monitoring. They can be divided into three broad classes.

A. Violation Detection

A first line of work concentrates on the detection of specification violations in a source of events. This can be done statically by analyzing traces of pre-recorded events, a process called log analysis. The approaches differ by their specification language, which includes finite-state machines [9] and regular expressions [10]. Temporal logic and variants thereof can also be used, such as Linear Temporal Logic in the ProM workbench [11], or propositional temporal logic with past operators in LogLogics [12].

In all these input languages, messages are considered as atomic units where data content is discarded; hence they are not appropriate for the kind of contract constraints described in this paper, which require quantification. To address the issue, tools like RuleR [13] and Logscope [14] allow a form of first-order quantification over event fields in the specification language. The quantification is similar, but not identical, to LTL-FO$^+$. Other works take a database approach: they log events into a database, and transform compliance requirements into SQL queries that look for violation patterns [15]. A similar idea was used using XML databases, transforming a trace of events into an XML file, and temporal logic specifications into XQuery expressions that can be evaluated on that file [16].

However, all these approaches suffer from one common limitation: they can only detect direct violations of a specification, similar to the simple monitor we described.
A second category of works take the diagnosis of a violation as given, and concentrate on finding the cause of that violation. The term “whodunit” for this kind of task has first been employed by Wang et al. [17]. Their work concentrates on a single trace of execution of a concrete program that ends up at a line of code violating an assertion; an algorithm can compute a minimal set of conditions that are necessary for the program to stay on that path and violate the assertion. However, the approach requires both a counter-example trace and the source code of the program. A similar attempt has been done on temporal logic specifications [18], where the causes are specific values of Boolean variables that contribute to the (eventual) falsification of the said formula. Both works concentrate on direct violations; their analysis aims at simplifying the representation of the trace to eliminate variables that do not contribute to the violation of that direct error. We use a different definition of causality.

Another form of causality analysis stems from the concept of auditing business process compliance [19]. For example, deontic logic deals with contract obligations and the repARATION of eventual violations [20]; yet, contract violations must be explicit in order to be compensated for, and hence the detection of these violations again corresponds to our notion of direct errors.

A recent work studies violations between multiple interacting components, each of which follows its own finite-state machine specification [21]. A peer is defined a necessary cause for a violation if replacing its projection onto the trace by any other compliant trace does not cause a violation. This necessary cause is slightly different from the notion of non-compliant event; while there can be multiple necessary causes, in our setting there can be only one root violation. Moreover, the paper provides a method to determine which component is responsible for the violation, but cannot determine which particular event caused it.

Ancillary to the notion of violation is that of vacuity, or the detection of formulae or parts thereof that do not affect the truth value of the global specification [22], [23]. The present approach is orthogonal to these works; a non-compliant event is such that the remainder of the trace does not affect the truth value of the specification.

A final category of works relate to control theory. Given an open system, controller synthesis aims at building a partner whose interaction with this system is guaranteed to follow a specification; when such a partner exists, the process is said to be controllable. In the shopping cart example, a controller for the seller would be a client implementation that is guaranteed to fulfill the global contract, and vice versa. The existence of a controller is equivalent to the absence of a root violation: when an interaction reaches a non-compliant event, no trace of future messages can ever satisfy the requirements; hence, at least one of the peers is non-controllable.

A finite representation of all possible partners (that is, a “universal” controller) is called an operating guideline [24]. In LTL, this simply amounts to the a priori labelling of the Büchi automaton, according to Bauer et al.’s method, followed by the removal of all states labelled ⊥. We have already discussed the impracticality of any finite-state representation of interface contracts expressed in LTL-FO+.

In addition, controller synthesis for propositional LTL is 2EXPTIME-complete [25]. This should be contrasted with the lower complexity class of EXPSPACE for the satisfiability problem in a first-order extension of LTL. This difference in complexity can be explained by the fact that our approach simply computes whether a set of formulae is satisfiable, which is equivalent to determining whether there exists a controller —without necessarily producing one. In contrast, synthesizing a (universal) controller amounts to pre-computing a verdict for all possible message traces, a much harder task.

VI. CONCLUSION

We can now take stock of the available options for the detection of interface contract violations. We first defined the root cause of a contract violation as the first message that eliminates all possibilities of a valid termination of the exchange. When the contracts are expressed in classical Linear Temporal Logic, the root cause can be determined in a straightforward way by labelling the states of a Büchi automaton. However, when contracts are expressed in LTL-FO+, no pre-computation can be made without first imposing a bound on the possible data values that can occur in messages.

Yet, we have shown how an “on-the-fly” algorithm for the runtime monitoring of LTL-FO+ formulae can be transformed to detect root violations in addition to direct ones. To this end, we have demonstrated that a property expressed on future paths can be translated into a property on the current state of the monitor. More precisely, finding the root cause of a contract violation has been reduced to computing satisfiability of a set of LTL-FO+ formulae.

The proof-of-concept implementation of this method relies on an external solver for this computation. This solver only works for a portion of LTL-FO+ that limits the way quantifiers can be used. However, the reasonable running times of this implementation suggest that the same approach could be taken online and validate messages at runtime. Implementing non-compliance detection into a runtime monitor would allow it to block events before the exchange is bound to failure, thus allowing the faulty sender to try a different message and avoid the triggering of compensation mechanisms. The scalability of the approach on formulae producing multiple “or” branches should be assessed by further experiments.
The approach lends itself to a couple of refinements. First, as the experimental discussion showed it, an obvious improvement consists of integrating the runtime monitoring algorithm with the satisfiability solver under a single application to avoid excessive cross-process overhead. Second, since LTL-FO+ models constraints on message parameters in addition to their sequences, it can be possible to extend the current framework not only to pinpoint the non-compliant event, but also the value inside that event, if any, that is responsible for the violation. Finally, one could try to simplify the explanation of a violation by finding the smallest set of unsatisfiable formulae inside the monitor’s node, drawing on well-known techniques of abstraction and refinement.

REFERENCES


