Preliminary Results Towards Contract Monitorability

Annalizz Vella
CS, ICT, University of Malta
annalizz.vella.10@um.edu.mt

Adrian Francalanza
CS, ICT, University of Malta
adrian.francalanza@um.edu.mt

This paper discusses preliminary investigations on the monitorability of contracts for web service descriptions. There are settings where servers do not guarantee statically whether they satisfy some specified contract, which forces the client (*i.e.,* the entity interacting with the server) to perform dynamic checks. This scenario may be viewed as an instance of Runtime Verification, where a pertinent question is whether contracts can be monitored for adequately at runtime, otherwise stated as the *monitorability of contracts*. We consider a simple language of finitary contracts describing both clients and servers, and develop a formal framework that describes server contract monitoring. We define monitor properties that potentially contribute towards a comprehensive notion of contract monitorability and show that our simple contract language satisfies these properties.

1 Introduction

Web services [7, 6] typically consist of two types of computing entities. *Servers* offer ranges of sequences of *service interactions* to *clients*, which in turn interact with these services and occasionally reach a state denoting client satisfaction. The service interactions offered by a server typically follow some predefined structure that may be formalised as a contract [6, 7, 15, 3]. Dually, the service interactions invoked by a client may also be expressed within the same formalism.

The contract calculus defined in [15, 2, 5] is an abstract formalism equipped with an operational semantics that provides an implementation-agnostic, high-level description of client-server interactions; this permits formal reasoning about web services, such as whether a client is compatible with a server or whether a server is able to satisfy the service interactions requested by the client. Such reasoning may, for instance, be used by clients for *dynamic service discovery*, where a client decides to interact with a server whenever the contract it advertises satisfies the requirements of the client.

Example 1.1. *Consider the contract below describing the behaviour of an internet banking server:*

$$\mathsf{login}.\big((\overline{\mathsf{valid}}.(\mathsf{query}.\mathbf{0} + \mathsf{transfer}.\mathbf{0})) \oplus (\overline{\mathsf{invalid}}.\mathbf{0})\big)$$

It states that the server first expects a login *service interaction followed by either a* valid *or* invalid *service invocation; the operator $\oplus$ denotes that the server decides autonomously whether to invoke* valid *or* invalid *in response. If it branches to the latter, it terminates all interactions, denoted by* $\mathbf{0}$*. However, if it internally decides to invoke the service interaction* valid*, it then offers a choice (denoted by the symbol $+$) of service interactions: it either accepts (account balance)* query *interactions or else (fund)* transfer *interactions. A contract describing the behaviour of a possible bank client is given below:*

$$\overline{\mathsf{login}}.\big((\mathsf{invalid}.\overline{\mathsf{reason}}.\mathbf{1}) + (\mathsf{expired}.\mathbf{1}) + (\mathsf{valid}.\overline{\mathsf{query}}.\mathbf{1})\big)$$

After a login service invocation, this client expects either of three responses: an invalid *interaction prompting another service request that asks for a* reason *why the login was invalid, a login* expired

Submitted to:
PrePost16

invocation or else a valid *login interaction that is followed by invoking a* query *service request. All these alternative sequences leave the client in a satisfied state,* **1**. *By analysing the* resp. *contracts, one can deduce that interactions on the* valid *service following a client* login *interaction necessarily lead to a* query *interaction, which then leaves the client satisfied. One can also discern that* invalid *interactions lead to a deadlock, whereby the client asks for a* reason *service that is not offered by the server. One can also note that the* expired *option offered by the client is never chosen by the server.* ■

Within this framework, there still remains the question of whether a service behaviour actually adheres to the contract it advertises. In general, static techniques (such as session-based type systems [9], or state-based model-checking of compliance, must or fair testing inclusion [15, 2, 5]) are used to verify *before deployment* whether a server implementation respects the contract that describes it. However, there are cases where this solution is not applicable. For instance, the client may decide *not* to trust the static verifier used by the server. Alternatively, in a dynamic setting where service components are downloaded and installed at runtime, pre-deployment checks cannot be made on the server implementation since some components only become available for inspection at runtime. There are also cases whereby a server does not come equipped with a formal description at all.

In these circumstances, a client can check that a server respects an advertised (or expected) contract by analysing the behaviour exhibited by the server *at runtime*. There are a number of cases where such a solution is adopted [4, 12], making use of dynamic monitoring, possibly in conjunction with other verification techniques. This monitoring of systems may be seen as an instance of Runtime Verification (RV) [13], a lightweight formal verification technique used to check the current execution of a program by verifying it against some properties. In a typical setup, the monitor observing the running system raises a flag when a *conclusive* verdict is reached, denoting that the property being checked for is either *satisfied* or *violated*.

An important question in any RV setup is that of the *monitorability* of the specification language considered. Indeed, it is generally the case that not all aspects of a specification can be monitored for and determined at runtime, as shown in [8, 1, 11] for specification languages such as LTL and the modal μ-calculus. In this work, we start to investigate the monitorability of contracts which, in turn, sheds light on the viability and expressiveness of the dynamic contract checking setup discussed above. In contrast to earlier work on monitorability, we do *not* rely on an external formal logic for specifying the properties expected by a server contract, *e.g.*, a satisfaction relation $p \models \phi$ where ϕ would be a formula from a logic defined over server contract p through the semantic relation $\models$. Instead, we use the subcontract server relation $q \sqsubseteq_{\text{SRV}} p$ defined in [15] as a *refinement semantic relation* where q is an abstract description of the expected properties of a server contract p, thus using the contract language itself as a specification language. Within this setting, we investigate whether our monitoring mechanism is expressive enough to verify whether a server p indeed refines an abstract description q.

The rest of the paper is structured as follows. Section 2 overviews our contract language and defines our notion of contract satisfaction. Section 3 introduces our monitoring setup and Section 4 relates verdicts reached by our monitored computations to the contract satisfactions discussed in Section 2. Section 5 concludes by discussing related and future work.

2 Servers, Clients and Satisfaction

Figure 1 describes the syntax and semantics of (finite) servers and clients. Let $a, b, c, d \ldots \in \text{NAMES}$ be a set of names denoting interaction addresses. Let $\bar{\cdot}$ be a complementation operation on these names where we refer to the complement of a as $\overline{a}$; the operation is an involution, where $\overline{\overline{a}} = a$. The set of actions

Syntax

$$p,q \in \textsc{Srv} ::= \mathbf{0} \quad \text{(inaction)} \qquad \mid \alpha.p \quad \text{(prefixing)}$$
$$\mid p+q \quad \text{(external choice)} \qquad \mid p \oplus q \quad \text{(internal choice)}$$

$$r,s \in \textsc{Cli} ::= \mathbf{0} \mid \alpha.r \mid r+s \mid r \oplus s \mid \mathbf{1} \quad \text{(success)}$$

Dynamics

$$\textsc{Act}\ \frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \textsc{SelL}\ \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'} \qquad \textsc{SelR}\ \frac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$$

$$\textsc{ChoL}\ \frac{}{p \oplus q \xrightarrow{\tau} p} \qquad \textsc{ChoR}\ \frac{}{p \oplus q \xrightarrow{\tau} q}$$

Interaction

$$\textsc{AsyS}\ \frac{p \xrightarrow{\tau} q}{r \parallel p \xrightarrow{\tau} r \parallel q} \qquad \textsc{AsyC}\ \frac{r \xrightarrow{\tau} s}{r \parallel p \xrightarrow{\tau} s \parallel p} \qquad \textsc{Syn}\ \frac{r \xrightarrow{\overline{\alpha}} s \quad p \xrightarrow{\alpha} q}{r \parallel p \xrightarrow{\tau} s \parallel q}$$

Figure 1: Server and Client Syntax and Semantics

$\alpha \in \textsc{Act} = \big(\textsc{Names} \cup \{\overline{a} \mid a \in \textsc{Names}\}\big)$ includes all names and their complement. Let τ be a distinct action *not* in $\textsc{Act}$ denoting *internal* unobservable activity, where we let $\mu \in \textsc{Act} \cup \{\tau\}$.

Servers, $p,q \in \textsc{Srv}$, consist of either the terminated server $\mathbf{0}$, a prefixed server $\alpha.p$ that first engages in interaction α and then behaves as p, an external choice $p+q$ that can either behave as p or q depending on the interactions it engages in, or an internal choice $p \oplus q$ that autonomously decides to either behave as p or q. Clients, $r,s \in \textsc{Cli}$, have a similar structure but may also consist of the term $\mathbf{1}$ denoting contract fulfilment. The semantics of both servers and clients are given in terms of a Labelled Transition System (LTS) where the labelled transition relation $p \xrightarrow{\mu} q$ is defined as the least relation satisfying the rules in Figure 1; the definition of the transition relation for clients $r \xrightarrow{\mu} s$ is analogous and thus elided. The definition is standard and follows that of related languages such as CCS [14]. For instance, the term $\alpha.p$ transitions with (action) label α to the continuation p; if p can engage in an interaction on μ and transition to p', then an external choice term involving p, *e.g.*, $p+q$ may also transition to p' after exhibiting action μ; by contrast, an internal choice involving p, *e.g.*, $p \oplus q$ may transition to p without exhibiting an external action (τ is used).

Servers and clients may be composed together to form a system, $r \parallel p$, so as to engage in a sequence of interactions. Interactions are also defined as an LTS over systems, through the rules AsyS, AsyC and Syn in Figure 1. As is standard, silent transitions by either server or client allow them to transition autonomously in a system. However, a client transition on an external action must be matched by a server transition on the (dual) co-action for the transition to occur in the *resp.* system, denoting client-server interaction. *Computations* are sequences of system transitions $r_0 \parallel p_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_n \parallel p_n$, denoted as $r_0 \parallel p_0 \Longrightarrow r_n \parallel p_n$; the sequence may be potentially empty, $n = 0$, where *no* transitions are made, in which case we have $r_0 = r_n$ and $p_0 = p_n$. A computation $r_0 \parallel p_0 \Longrightarrow r_n \parallel p_n$ is *maximal* whenever

$\nexists r', p' \cdot r_n \parallel p_n \xrightarrow{\tau} r' \parallel p'$.

Definition 2.1. *A maximal computation, $r \parallel p \Longrightarrow s \parallel q$, is* successful, *whenever the client's contract is fulfilled, meaning that $s = \mathbf{1}$. A service p* satisfies *a client r, denoted as $\mathsf{sat}(p,r)$, when* every *maximal computation rooted at $r \parallel p$ is successful.* ■

Example 2.2. *The server $p = \overline{a}.\mathbf{0} + (b.a.\mathbf{0} \oplus c.\mathbf{0})$ may either transition as $p \xrightarrow{\overline{a}} \mathbf{0}$ using rules* ACT *and* SELL *from Figure 1, or silently transition as $p \xrightarrow{\tau} b.a.\mathbf{0}$ or $p \xrightarrow{\tau} c.\mathbf{0}$ via rules* CHOL, CHOR *and* SELR *from Figure 1. It satisfies the client $r = \overline{b}.\mathbf{1} + \overline{c}.\mathbf{1}$, denoted as $\mathsf{sat}(p,r)$, because the only maximal computations possible are the following*

$$r \parallel p \xrightarrow{\tau} r \parallel b.a.\mathbf{0} \xrightarrow{\tau} \mathbf{1} \parallel a.\mathbf{0} \qquad\qquad r \parallel p \xrightarrow{\tau} r \parallel c.\mathbf{0} \xrightarrow{\tau} \mathbf{1} \parallel \mathbf{0}$$

both of which are successful. By contrast, server p does not *satisfy client $\overline{b}.\mathbf{1}$, denoted as $\neg\mathsf{sat}(p,\overline{b}.\mathbf{1})$, nor does it satisfy the clients $\overline{b}.\mathbf{1} + \overline{b}.\mathbf{0} + \overline{c}.\mathbf{1}$ and $\overline{b}.c.\mathbf{1} + \overline{c}.\mathbf{1}$. In each case, we can show this through the* unsuccessful *maximal computations below.*

$$\overline{b}.\mathbf{1} \parallel p \xrightarrow{\tau} \overline{b}.\mathbf{1} \parallel c.\mathbf{0} \qquad\qquad \overline{b}.\mathbf{1} + \overline{b}.\mathbf{0} + \overline{c}.\mathbf{1} \parallel p \xrightarrow{\tau} \overline{b}.\mathbf{1} + \overline{b}.\mathbf{0} + \overline{c}.\mathbf{1} \parallel b.\mathbf{0} \xrightarrow{\tau} \mathbf{0} \parallel \mathbf{0}$$
$$\overline{b}.c.\mathbf{1} + \overline{c}.\mathbf{1} \parallel p \xrightarrow{\tau} \overline{b}.c.\mathbf{1} + \overline{c}.\mathbf{1} \parallel b.\mathbf{0} \xrightarrow{\tau} c.\mathbf{1} \parallel \mathbf{0}$$ ■

The satisfaction predicate $\mathsf{sat}(-,-)$ induces a natural preorder amongst servers.

Definition 2.3 (Server Preorder [15])**.** *A server p is a subcontract of server q, denoted as $p \sqsubseteq_{\text{SRV}} q$, whenever, for* all *clients r, $\mathsf{sat}(p,r)$ implies $\mathsf{sat}(q,r)$. Dually, q is referred to as a supercontract of p.* ■

Intuitively, $p \sqsubseteq_{\text{SRV}} q$ of Definition 2.3 means that we can substitute a server p by a server q, safe in the knowledge that any client satisfied by p would not be affected.

Example 2.4. *Definition 2.3 allows us to establish a number of useful server (in)equations such as*

$$\overline{a}.\mathbf{0} \oplus b.\mathbf{0} \sqsubseteq_{\text{SRV}} \overline{a}.\mathbf{0} \qquad b.a.\mathbf{0} + b.c.\mathbf{0} \sqsubseteq_{\text{SRV}} b.(a.\mathbf{0} \oplus c.\mathbf{0}) \qquad b.(a.\mathbf{0} \oplus c.\mathbf{0}) \sqsubseteq_{\text{SRV}} b.a.\mathbf{0} + b.c.\mathbf{0}$$

but also justify subtle cases where substituting one server for another might break client satisfaction. For instance, we have $\mathbf{0} \not\sqsubseteq_{\text{SRV}} a.\mathbf{0}$ because for the client $(\mathbf{1} \oplus \mathbf{1}) + \overline{a}.\mathbf{0}$ we have $\mathsf{sat}(\mathbf{0}, (\mathbf{1} \oplus \mathbf{1}) + \overline{a}.\mathbf{0})$ since $(\mathbf{1} \oplus \mathbf{1}) + \overline{a}.\mathbf{0} \parallel \mathbf{0} \xrightarrow{\tau} \mathbf{1} \parallel \mathbf{0}$ is the only maximal computation (which is also successful), but *also have $\neg\mathsf{sat}(a.\mathbf{0}, (\mathbf{1} \oplus \mathbf{1}) + \overline{a}.\mathbf{0})$ due to the unsuccessful computation $(\mathbf{1} \oplus \mathbf{1}) + \overline{a}.\mathbf{0} \parallel a.\mathbf{0} \xrightarrow{\tau} \mathbf{0} \parallel \mathbf{0}$.* ■

3 Monitors and Monitored Computations

Figure 2 describes the monitoring framework used to analyse servers purporting to adhere to some advertised contract. It defines the syntax of these monitors, which follow the general structure used in earlier works [11, 1] whereby monitors may reach any one of the three verdicts VERD, namely acceptance, rejection, or the inconclusive verdict. In addition to the basic prefixing patterns used in [11, 10], we here also use action complementation, $\underline{\alpha}$, to denote any action *apart from* α. As in [11, 10], a monitor is allowed to branch, $m + n$, depending on the actions observed at runtime. We also find it convenient to express a merge monitor operator that facilitates the composition of monitor specifications, $m \times n$.

The semantics of a monitor is given in terms of the LTS defined by the rules in Figure 2. This is best viewed as the evolution of a monitor in response to a (finite) execution trace $t \in \text{ACT}^*$, consisting of a sequence of actions $\alpha_1, \ldots, \alpha_n$. Verdicts are irrevocable when reached, and do not change upon viewing

Syntax

$$
\begin{array}{rlll}
v,u \in \textsc{Verd} & ::= \mathsf{Y} & \text{(acceptance)} & \mid \mathsf{N} \quad \text{(rejection)} \\
& \mid \mathsf{end} & \text{(inconclusive)} & \\
\theta \in \textsc{Patterns} & ::= \alpha & \text{(action)} & \mid \underline{\alpha} \quad \text{(complement)} \\
m,n \in \textsc{Mon} & ::= v & \text{(verdict)} & \mid \theta.m \quad \text{(interaction)} \\
& \mid m+n & \text{(choice)} & \mid m \times n \quad \text{(conjunction)}
\end{array}
$$

Dynamics

$$\textsc{mVer}\ \frac{}{v \xrightarrow{\alpha} v} \qquad \textsc{mAct}\ \frac{}{\alpha.m \xrightarrow{\alpha} m} \qquad \textsc{mNAct}\ \frac{\beta \neq \alpha}{\underline{\alpha}.m \xrightarrow{\beta} m}$$

$$\textsc{mSelL}\ \frac{m \xrightarrow{\alpha} m'}{m+n \xrightarrow{\alpha} m'} \qquad \textsc{mSelR}\ \frac{n \xrightarrow{\alpha} n'}{m+n \xrightarrow{\alpha} n'} \qquad \textsc{mConj}\ \frac{m \xrightarrow{\alpha} m' \quad n \xrightarrow{\alpha} n'}{m \times n \xrightarrow{\alpha} m' \times n'}$$

Instrumentation

$$\textsc{iMon}\ \frac{p \xrightarrow{\alpha} p' \quad m \xrightarrow{\alpha} m'}{m \triangleleft p \xrightarrow{\alpha} m' \triangleleft p'} \qquad \textsc{iTer}\ \frac{p \xrightarrow{\alpha} p' \quad m \overset{\alpha}{\not\longrightarrow}}{m \triangleleft p \xrightarrow{\alpha} \mathsf{end} \triangleleft p'} \qquad \textsc{iAsy}\ \frac{p \xrightarrow{\tau} p'}{m \triangleleft p \xrightarrow{\tau} m \triangleleft p'}$$

Figure 2: Monitors and Instrumentation

further actions in the trace (rule mVer). Prefixing releases the guarded monitor when the expected pattern is encountered (rules mAct and mNAct). The rules mSelL and mSelR describe left and right monitor branching as expected, whereas rule mConj describes the synchronous evolution of merged monitors.

A *monitored server contract* consists of a server p that is instrumented with a monitor m, denoted as $m \triangleleft p$. The behaviour of monitored contracts is defined as an LTS through the rules stated in Figure 2, and relies on the *resp.* LTSs of the monitor and the server. Rule iMon states that if a server can transition with action α and the monitor can follow this by transitioning with the same action, then in an instrumented server they transition in lockstep. However, if the monitor cannot follow such a transition the instrumentation forces it to terminate with an inconclusive verdict, end, while the process is allowed to proceed unaffected; see rule iTer. Finally, rule iAsy allows a contract to evolve independently from the monitor when performing silent τ moves (which are unobservable to the monitor). We refer to a sequence of transitions from a monitored contract as a *monitored computation* and use the standard notation $m \triangleleft p \overset{t}{\Rightarrow} m' \triangleleft p'$ that abstracts over τ-moves in trace t.

A few comments are in order. First, we highlight the fact that in the operational semantics for monitored systems of Figure 2, the monitor does *not* have access to the internal state of the server generating the trace, and its observations are limited to the execution that the server chooses to exhibit at runtime. This is meant to model the RV scenarios mentioned in Section 1, where the source of the executing system cannot be analysed: from the point of view of the runtime monitoring and verification, the server description is merely used to generate traces. Second, we note that, in a monitored server setup, any visible behaviour is instigated by the server, relegating the instrumented monitor to a *passive* role

that merely follows the server actions. Stated otherwise, the server *drives* the behaviour in a monitored system and dictates the execution path that the monitor can analyse at runtime.

In what follows, we explain how monitors work through a series of examples. The exposition focuses on monitors that produce rejection verdicts, but the discussion can be extended to acceptance verdicts in a straightforward manner.

Example 3.1. *The monitor $\underline{\overline{a}}.N+\overline{a}.\underline{b}.N$ checks for violations from contracts that are expected to adhere to (i.e., be supercontracts of) the contract $\overline{a}.b.\mathbf{0}$. In fact, the monitor reaches a rejection verdict whenever a contract either emits an action that is not $\overline{a}$ at runtime, $\underline{\overline{a}}.N$, or else emits an action that is not $\overline{b}$ following action a, $\overline{a}.\underline{b}.N$. Consider the server $\overline{a}.c.\mathbf{0}$; when instrumented with our monitor we can observe the following monitored computation whereby the monitor reaches a rejection verdict, N.*

$$\left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(\overline{a}.c.\mathbf{0}\right) \quad\xrightarrow{\overline{a}}\quad \underline{b}.N\triangleleft c.\mathbf{0} \quad\xrightarrow{c}\quad N\triangleleft\mathbf{0}$$

By contrast, when the server $\overline{a}.b.\mathbf{0}$ is instrumented with the monitor, no rejection verdict is reached; in particular, the final transition below is derived using rule ITER *because $\underline{b}.N \overset{b}{\not\longrightarrow}$.*

$$\left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(\overline{a}.b.\mathbf{0}\right) \quad\xrightarrow{\overline{a}}\quad \underline{b}.N\triangleleft b.\mathbf{0} \quad\xrightarrow{b}\quad end\triangleleft\mathbf{0}$$

We emphasise the fact that monitor termination through rule ITER *is crucial to avoid unwanted detections. Consider a variant of the earlier monitor, $\overline{a}.b.N$, which now reports violations whenever it observes the trace consisting of the action $\overline{a}$ followed by the action b. When composed with the system $\overline{a}.c.b.\mathbf{0}$ we observe the following monitored computation.*

$$\begin{aligned}\overline{a}.b.N\triangleleft\overline{a}.c.b.\mathbf{0} &\xrightarrow{\overline{a}} b.N\triangleleft c.b.\mathbf{0}\\ &\xrightarrow{c} end\triangleleft b.\mathbf{0}\\ &\xrightarrow{b} end\triangleleft\mathbf{0}\end{aligned} \qquad (**)$$

*At transition (**), the server can perform an action, c, that the monitor is not able to follow (*i.e.*, it is* not *specified how the monitor should behave at that point should it observe action c). Accordingly, the semantics instructs the monitor to terminate (prematurely) with an inconclusive verdict. There are two instrumentation alternatives that could have been adopted, both of which are arguably wrong from a monitoring perspective. The first option would have been to prohibit the server from exhibiting action c, which goes against the tenet that the monitor should adopt a passive role and not interfere with the execution of the program it monitors. The second option is arguably even worse: we could have let the server transition and left the monitor in its present state, i.e., $b.N\triangleleft c.b.\mathbf{0} \xrightarrow{c} b.N\triangleleft b.\mathbf{0}$, but then this would have led to an* unspecified/erroneous *detection at the next transition $b.N\triangleleft b.\mathbf{0} \xrightarrow{b} N\triangleleft\mathbf{0}$.* ■

Example 3.2. *The server $\overline{a}.b.\mathbf{0}\oplus c.b.\mathbf{0}$ is* not *a supercontract of $\overline{a}.b.\mathbf{0}$ according to Definition 2.3. Crucially, however, in an RV setting, monitor detection depends on the runtime behaviour exhibited by the server. This contrasts with other forms of verification which may be allowed to explore* all *the execution paths of a server under scrutiny.*[1]

$$\begin{aligned}&\left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(\overline{a}.b.\mathbf{0}\oplus c.b.\mathbf{0}\right) \xrightarrow{\tau} \left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(\overline{a}.b.\mathbf{0}\right) \xrightarrow{\overline{a}} \underline{b}.N\triangleleft b.\mathbf{0} \xrightarrow{b} end\triangleleft\mathbf{0}\\ &\left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(\overline{a}.b.\mathbf{0}\oplus c.b.\mathbf{0}\right) \xrightarrow{\tau} \left(\underline{\overline{a}}.N+\overline{a}.\underline{b}.N\right)\triangleleft\left(c.b.\mathbf{0}\right) \xrightarrow{c} N\triangleleft b.\mathbf{0} \xrightarrow{b} N\triangleleft\mathbf{0}\end{aligned}$$

[1] In the general case, a pre-deployment verification technique may also analyse *infinite* paths.

In the first monitored computation above, the server exhibits the behaviour described by the trace $\stackrel{\overline{a}b}{\Longrightarrow}$*, which prohibits the monitor from detecting any violations. However, the same server exhibits a different trace* $\stackrel{cb}{\Longrightarrow}$ *in the second monitored computation which permits monitor detection. The rejection verdict is in fact reached after the first visible transition on action c, and then preserved throughout the remainder of the computation.* ■

Example 3.3. *We can monitor for violations of the contract* $\overline{a}.b.\mathbf{0}+c.\mathbf{0}$ *by composing two submonitors that monitor for the constituents. Specifically, since the monitor* $\underline{c}.N+c.\mathsf{end}$ *checks for violations of contract* $c.\mathbf{0}$ *and, the minimally extended monitor* $\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end})$ *checks for violations of* $\overline{a}.b.\mathbf{0}$ *as discussed in Example 3.1, we can construct the composite monitor* $(\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end}))\times(\underline{c}.N+c.\mathsf{end})$ *to monitor for violations of* $\overline{a}.b.\mathbf{0}+c.\mathbf{0}$*.*

$$\begin{array}{lll}\big((\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end}))\times(\underline{c}.N+c.\mathsf{end})\big)\triangleleft\overline{a}.b.\mathbf{0}+c.\mathbf{0} & \stackrel{\overline{a}}{\longrightarrow} & \underline{b}.N+b.\mathsf{end}\times N\triangleleft b.\mathbf{0}\\ & \stackrel{b}{\longrightarrow} & \mathsf{end}\times N\triangleleft\mathbf{0}\\ \big((\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end}))\times(\underline{c}.N+c.\mathsf{end})\big)\triangleleft\overline{a}.b.\mathbf{0}+c.\mathbf{0} & \stackrel{c}{\longrightarrow} & N\times\mathsf{end}\triangleleft\mathbf{0}\end{array}$$

When the composite monitor is instrumented with the contract it is expected to monitor for, we note that it does not reach a rejection along every *(parallel) submonitor.*

$$\big((\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end}))\times(\underline{c}.N+c.\mathsf{end})\big)\triangleleft b.\mathbf{0}\quad\stackrel{b}{\longrightarrow}\quad N\times N\triangleleft\mathbf{0}$$

By contrast, the violating contract above generates a rejection along every submonitor. ■

Example 3.3 clearly suggests a definition of monitor rejection.

Definition 3.4 (Rejection)**.** *A monitor m is in a rejection state, denoted as* $\boldsymbol{rej}(m)$*, whenever it is of the form* $N\times\ldots\times N$*. We overload this predicate to denote a server p being rejected by a monitor m, defined formally as*

$$\boldsymbol{rej}(p,m)\stackrel{def}{=}\exists t,p'\cdot m\triangleleft p\stackrel{t}{\Rightarrow}m'\triangleleft p'\text{ and }\boldsymbol{rej}(m')$$

Example 3.5. *The monitor* $\underline{c}.N+c.\mathsf{end}$ *rejects server* $b.\mathbf{0}$*,* $\boldsymbol{rej}(b.\mathbf{0},(\underline{c}.N+c.\mathsf{end}))$ *as well as server* $c.\mathbf{0}+b.\mathbf{0}$*,* $\boldsymbol{rej}((c.\mathbf{0}+b.\mathbf{0}),(\underline{c}.N+c.\mathsf{end}))$ *because both may exhibit an execution trace that leads the monitor to a rejection state. By contrast,* $\underline{c}.N+c.\mathsf{end}$ *does* not *reject server* $c.\mathbf{0}$*. Recalling monitor* $m=\big((\underline{a}.N+\overline{a}.(\underline{b}.N+b.\mathsf{end}))\times(\underline{c}.N+c.\mathsf{end})\big)$ *from Example 3.3, we can also state that it rejects server* $b.\mathbf{0}$*,* $\boldsymbol{rej}(b.\mathbf{0},m)$*.* ■

4 Preliminary results towards Monitorability

Monitorability may be broadly described as the relationship between the properties of a logic specifying program behaviour and the detection capabilities of a monitoring setup instrumented over such programs. It is therefore parametric with respect to the logic and monitoring setup considered. In what follows, we sketch out preliminary investigations that focus on the monitor rejections defined in Section 3, and attempt to relate them to violations of the server preorder defined in Section 2.

We have already defined enough machinery to be able to state formally two important properties. Definition 4.1 states that a monitor *m soundly monitors* for a server contract p if and only if, whenever it rejects a server q, it is indeed the case that q is not a supercontract of p. In a sense, the dual of this is Definition 4.2, which states that a monitor *m completely monitors* for a server contract p if and only if every q that is not a supercontract of p is rejected by m.

Definition 4.1 (Rejection Sound). $\boldsymbol{smon}(p,m) \stackrel{def}{=} \forall q \cdot \boldsymbol{rej}(q,m)$ *implies* $p \not\sqsubseteq_{\text{SRV}} q$. ■

Definition 4.2 (Rejection Complete). $\boldsymbol{cmon}(p,m) \stackrel{def}{=} \forall q \cdot\ p \not\sqsubseteq_{\text{SRV}} q$ *implies* $\boldsymbol{rej}(q,m)$. ■

We can also extend these monitorability definitions to a specification language of contracts (*i.e.,* a set of contracts).

Definition 4.3 (Language Rejection Monitorability). *A set of contracts* $\mathscr{C}$ *is:*

- *sound rejection-monitorable iff* $\forall p \in \mathscr{C} \cdot \exists m \in \text{MON} \cdot \boldsymbol{smon}(p,m)$
- *complete rejection-monitorable iff* $\forall p \in \mathscr{C} \cdot \exists m \in \text{MON} \cdot \boldsymbol{cmon}(p,m)$
- *rejection-monitorable iff* $\forall p \in \mathscr{C} \cdot \exists m \in \text{MON} \cdot \boldsymbol{smon}(p,m)$ *and* $\boldsymbol{cmon}(p,m)$ ■

We can readily argue in a formal manner that the contract language SRV of Figure 1 *cannot* be rejection-monitorable. Consider as an example $\overline{a}.\mathbf{0}+b.\mathbf{0} \in \text{SRV}$. If this language is rejection-monitorable, then there must exist a monitor m such that $\mathbf{smon}(\overline{a}.\mathbf{0}+b.\mathbf{0},m)$ and $\mathbf{cmon}(\overline{a}.\mathbf{0}+b.\mathbf{0},m)$. We argue towards a contradiction. From Section 2 we know that $\overline{a}.\mathbf{0}+b.\mathbf{0} \not\sqsubseteq_{\text{SRV}} \overline{a}.\mathbf{0}$, and thus, by $\mathbf{cmon}(\overline{a}.\mathbf{0}+b.\mathbf{0},m)$, it must be the case that $\mathbf{rej}(\overline{a}.\mathbf{0},m)$. Now this rejection predicate holds if either m reaches a rejection state immediately or else reaches rejection after observing action a. In either case, this monitor would also reject the contract $\overline{a}.\mathbf{0}+b.\mathbf{0}$ as well, which would make the monitor necessarily unsound, *i.e.,* $\neg\mathbf{smon}(\overline{a}.\mathbf{0}+b.\mathbf{0},m)$, since, by the reflexivity property of the preorder, we have $\overline{a}.\mathbf{0}+b.\mathbf{0} \sqsubseteq_{\text{SRV}} \overline{a}.\mathbf{0}+b.\mathbf{0}$.

We deem sound rejection to be the minimum correctness requirement to be expected from the contract monitors we consider. Note, however, that the contract language SRV of Figure 1 is trivially sound rejection-monitorable via the monitor end; this monitor never reaches a rejection state and thus trivially satisfying $\mathbf{rej}(p,\mathsf{end})$ for any $p \in \text{SRV}$. However, we argue that this monitor, end, is not very useful.

We attempt to go one step further and define an automated monitor synthesis function that returns a monitor for *every server* in the contract language SRV. We argue, at least informally, that these synthesised monitors are, in some sense, useful because they perform a degree of violation detections. Importantly, however, we show that these synthesised monitors are rejection sound, according to Definition 4.1.

Definition 4.4 (Monitor Synthesis). *The function* $[\![-]\!] : \text{SRV} \to \text{MON}$ *synthesises a monitor from a server contract description, and is defined inductively on the structure of this contract as follows:*

$$[\![\mathbf{0}]\!] \stackrel{def}{=} end \qquad [\![\alpha.p]\!] \stackrel{def}{=} \underline{\alpha}.N + \alpha.[\![p]\!]$$
$$[\![p+q]\!] \stackrel{def}{=} [\![p]\!] \times [\![q]\!] \qquad [\![p \oplus q]\!] \stackrel{def}{=} [\![p]\!] \times [\![q]\!]$$ ■

A few comments on Definition 4.4 are in order. First, note that a number of the monitors considered earlier in Section 3 are in fact instances of this translation. For instance, we have

$$[\![\overline{a}.b.\mathbf{0}]\!] = \underline{\overline{a}}.\mathsf{N} + \overline{a}.(\underline{b}.\mathsf{N} + b.\mathsf{end}) \qquad \text{and} \qquad [\![\overline{a}.b.\mathbf{0}+c.\mathbf{0}]\!] = (\underline{\overline{a}}.\mathsf{N} + \overline{a}.(\underline{b}.\mathsf{N} + b.\mathsf{end})) \times (\underline{c}.\mathsf{N} + c.\mathsf{end})$$

from Example 3.3. Secondly, note that the monitor synthesis does not attempt to perform any detection violation for the contract $\mathbf{0}$. Since $\mathbf{0}$ is in some sense a bottom element in the preorder, no supercontract of $\mathbf{0}$ is allowed to perform any visible action. Thus, in cases where all the actions permissible in SRV are known up front as a *finite* set $\{\alpha_1,\ldots,\alpha_n\}$, we can improve the precision of our synthesis through the alternative definition $[\![\mathbf{0}]\!] \stackrel{\text{def}}{=} \underline{\alpha_1}.\mathsf{N} + \ldots + \underline{\alpha_n}.\mathsf{N}$ for the case where $p = \mathbf{0}$. Third, note that the synthesis for both internal and external choice constructs coincide which, in a sense, is due to the inherent discriminating limits of RV. Consider, by way of example, the monitor syntheses below:

$$[\![\overline{a}.\mathbf{0}+b.\mathbf{0}]\!] \;=\; ((\underline{\overline{a}}.\mathsf{N} + \overline{a}.\mathsf{end})) \times ((\underline{b}.\mathsf{N} + b.\mathsf{end})) \;=\; [\![\overline{a}.\mathbf{0} \oplus b.\mathbf{0}]\!]$$

The server $c.\mathbf{0}$ is rejected by the monitor $((\underline{\overline{a}}.\mathsf{N} + \overline{a}.\mathsf{end})) \times ((\underline{b}.\mathsf{N} + b.\mathsf{end}))$ and accordingly it is *neither* a supercontract of $\overline{a}.\mathbf{0} + b.\mathbf{0}$ *nor* of $\overline{a}.\mathbf{0} \oplus b.\mathbf{0}$. However, the server $\overline{a}.\mathbf{0}$ is *not* rejected by the monitor $((\underline{\overline{a}}.\mathsf{N} + \overline{a}.\mathbf{0})) \times ((\underline{b}.\mathsf{N} + b.\mathbf{0}))$; whereas it is correct to do so in the case of monitoring for the internal choice contract $\overline{a}.\mathbf{0} \oplus b.\mathbf{0}$ because $\overline{a}.\mathbf{0} \oplus b.\mathbf{0} \sqsubseteq_{\text{SRV}} \overline{a}.\mathbf{0}$, it leads to lack of precision in the case of the external choice $\overline{a}.\mathbf{0} + b.\mathbf{0}$ since $\overline{a}.\mathbf{0} + b.\mathbf{0} \not\sqsubseteq_{\text{SRV}} \overline{a}.\mathbf{0}$. In spite of these limitations, we are able to show that our proposed monitor synthesis is sound.

Theorem 4.5 (Synthesis Soundness)**.** *For every server specification $p \in$ SRV, every server implementation $q \in$ SRV, and the monitor synthesis function $[\![-]\!]$ of Definition 4.4:*

$$\textit{Whenever } \textbf{\textit{rej}}(q, [\![p]\!]) \textit{ then it is necessarily the case that } p \not\sqsubseteq_{\text{SRV}} q$$

Proof. By structural induction on the server specification p. □

5 Conclusion

We have presented preliminary investigations relating to the monitorability of contracts, high-level descriptions for web services. We developed a monitoring framework that complements the operational semantics of server contracts. We then focused on the rejection expressivity of the monitors within this framework and related it to cases where it is unsafe to replace one server (contract) with another. Within our simple framework, we were already able to identify limits with respect to monitor detection powers, and were able to diagnose problems with a proposed automated monitor synthesis procedure. We were also able to formally prove that, in spite of its limit, the monitor synthesis considered is, in some sense, correct (Theorem 4.5).

Related and Future Work The language of contracts for web services has been discussed in several other works prior to ours, such as [2, 5, 15, 7]; although conceptually simple, it has been shown to be expressive enough to capture the dynamicity of interactions specified by more elaborate contract descriptions. The server preorder considered in this paper captures the essence of the must preorder, studied in [3] and the compliance preorder, studied in [15, 7]; in our simplistic case of finite servers and clients, the two preorders coincide (modulo minor technical details regarding client satisfaction and computation success). Our notion of monitorability is inspired by that presented in [11], which relates process satisfaction of a branching-time logic, $p \models \phi$, with detections of monitors synthesised from formulas in this logic, $[\![\phi]\!] \lhd p$. The instrumentation relation considered in this paper is in fact an adaptation to the one used in [11].

For future work, we aim to achieve a more comprehensive study of monitorability than the preliminary one presented in Section 4. In particular, we plan to consider monitor acceptances as a verdict in addition to rejections, establish stronger results with respect to rejections and consider extended contract descriptions similar to [3, 7] that include recursion and the potential for infinite computation. This will lead to different notions of server refinements such as those resulting from compliance and fair testing preorders [5, 15]. It will be interesting to study whether any of the aforementioned server preorder variants are more monitorable than the others.

Acknowledgements: This research was partly supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" of the Icelandic Research Fund.

References

[1] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL*. *TOSEM, ACM* 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800.

[2] Giovanni Bernardi & Matthew Hennessy (2012): *Modelling Session Types Using Contracts*. SAC, ACM, pp. 1941–1946, doi:10.1145/2245276.2232097.

[3] Giovanni Bernardi & Matthew Hennessy (2015): *Mutually Testing Processes*. *LMCS* 11(2), doi:10.2168/LMCS-11(2:1)2015.

[4] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): *Monitoring Networks through Multiparty Session Types*. In: *FMOODS/FORTE*, *LNCS* 7892, Springer, pp. 50–65, doi:10.1007/978-3-642-38592-6_5.

[5] M. Bravetti & G. Zavattaro (2009): *A theory of contracts for strong service compliance*. *MSCS* 19(3), doi:10.1017/S0960129509007658.

[6] S. Carpineti, G. Castagna, C. Laneve & L. Padovani (2006): *A Formal Account of Contracts for Web Services*. In: *WS-FM*, *LNCS* 4184, Springer, pp. 148–162, doi:10.1007/11841197_10.

[7] Giuseppe Castagna, Nils Gesbert & Luca Padovani (2009): *A Theory of Contracts for Web Services*. *TOPLAS, ACM* 31(5), pp. 19:1–19:61, doi:10.1145/1538917.1538920.

[8] Clare Cini & Adrian Francalanza (2015): *An LTL Proof System for Runtime Verification*. In: *TACAS*, *LNCS* 9035, Springer, pp. 581–595, doi:10.1007/978-3-662-46681-0_54.

[9] Mariangiola Dezani-Ciancaglini & Ugo De'Liguoro (2009): *Sessions and Session Types: An Overview*. In: *WS-FM*, *LNCS* 6194, Springer, pp. 1–28, doi:10.1007/978-3-642-14458-5_1.

[10] Adrian Francalanza (2016): *A Theory of Monitors (Extended Abstract)*. In: *FoSSaCS*, *LNCS* 9634, Springer, pp. 145–161, doi:10.1007/978-3-662-49630-5_9.

[11] Adrian Francalanza, Luca Aceto & Anna Ingólfsdóttir (2015): *On Verifying Hennessy-Milner Logic with Recursion at Runtime*. In: *RV*, *LNCS* 9333, Springer, pp. 71–86, doi:10.1007/978-3-319-23820-3_5.

[12] Limin Jia, Hannah Gommerstadt & Frank Pfenning (2016): *Monitors and Blame Assignment for Higher-order Session Types*. POPL, ACM, pp. 582–594, doi:10.1145/2837614.2837662.

[13] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *JLAP* 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[14] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall.

[15] Luca Padovani (2009): *Contract-Based Discovery and Adaptation of Web Services*. In: *SFM*, *LNCS* 5569, Springer, pp. 213–260, doi:10.1007/978-3-642-01918-0_6.