Behavioural contracts with request-response operations [★]

Lucia Acciai   Michele Boreale

Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy

Abstract

In the context of service-oriented computing, behavioural contracts are abstract descriptions of the message-passing behaviour of services. They can be used to check properties of service compositions such as, for instance, client-service compliance. To the best of our knowledge, previous formal models for contracts consider unidirectional *send* and *receive* operations. In this paper, we present two models for contracts with bidirectional *request-response* operations, in the presence of unboundedly many instances of both clients and servers. The first model takes inspiration from the abstract service interface language WSCL, the second one is inspired by Abstract WS-BPEL. We prove that two different notions of client-service compliance (one based on client satisfaction and another one requiring mutual completion) are decidable in the former while they are undecidable in the latter, thus showing an interesting expressiveness gap between the modeling of *request-response* operations in WSCL and in Abstract WS-BPEL.

1 Introduction

One interesting aspect of Service Oriented Computing (SOC) and Web Services technology is the need to describe in rigorous terms not only the format of the messages exchanged among interacting parties (as done, e.g., with the standard language WSDL [30]), but also the order in which such messages should be received and transmitted (as done, e.g., with the languages WSCL [29], WSCI [28],

[★] Extended version of [3]. The third author is partly supported by the EU integrated projects HATS and is member of the joint INRIA/University of Bologna Research Team FOCUS.

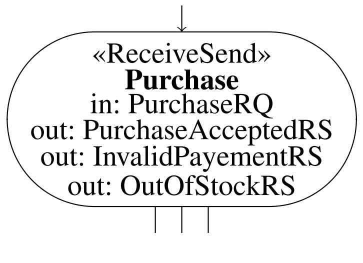
and Abstract WS-BPEL [24]). This specific aspect is clearly described in the Introduction of the Web Service Conversation Language (WSCL) specification [29], one of the proposals of the World Wide Web Consortium (W3C) for the description of the so-called Web Services *abstract interfaces*:

> Defining which XML documents are expected by a Web service or are sent back as a response is not enough. It is also necessary to define the order in which these documents need to be exchanged; in other words, a business level conversation needs to be specified. By specifying the conversations supported by a Web service —by defining the documents to be exchanged and the order in which they may be exchanged— the external visible behavior of a Web service, its abstract interface, is defined.

The abstract interface of services, sometimes called *behavioural contracts* (simply *contracts* in the following) can be used in several ways. For instance, one could check the *compliance* between a client and a service, that is, a guarantee for the client that the interaction with the service will in any case be completed successfully. One could also check, during the service discovery phase, the *conformance* of a concrete service to a given abstract interface by verifying whether the service implements at least the expected functionalities and does not require more.

Formal models are called for to devise rigorous forms of reasoning and verification techniques for services and abstract interfaces. To this aim, theories of *behavioural contracts* based on CCS-like process calculi [21] have been thoroughly investigated [4,7,8,9,11,12,13]. However, these models lack of expressiveness in at least one respect: they cannot be employed to describe bidirectional *request-response* interactions, in contexts where several instances of the client and of the service may be running at the same time. This situation, on the other hand, is commonly found in practice-oriented contract languages, like the abstract service interface language WSCL [29], WSCI [28], and Abstract WS-BPEL [24].

The WSCL language is a graphical notation similar to tradition flowcharts. There are four classes of basic actions: the one-way actions *Send* and *Receive*, and the two-way actions *SendReceive* and *ReceiveSend*. An example of *ReceiveSend* action is on the right.



WSCI is a richer XML-based language used to describe the behaviour of the actors involved in a multiparty service composition. Differently from WSCL, in WSCI it is possible to indicate an activity to be executed between the receive and the send actions of a two-way input operation, as the process `tns:BookSeats` executed within the request-response operation `tns:TAtoTraveler/bookTickets` in the example below:

```
<action name = "ReceiveConfirmation"
    role = "tns:TravelAgent"
    operation = "tns:TAtoTraveler/bookTickets">
         <call process = "tns:BookSeats" />
</action>
```

Abstract WS-BPEL is an XML-based language, but differently from WSCl, the request-response operations are modeled with two distinct `receive` and `reply` actions, that are correlated because they are executed with the same partner (see the example on the right). Between the `receive` and the `reply` actions any other action could be specified.

```
<receive partnerLink="purchase"
         portType="lns:OrderPT"
         operation="sendOrder"
         variable="PO">
...
<reply   partnerLink="purchase"
         portType="lns:OrderPT"
         operation="sendOrder"
         variable="Invoice">
```

In this paper, we present a formal investigation of contract languages of the type described above, that is allowing bidirectional request-response interactions, taking place between instances of services and clients. We present two formal contract languages that, for simplicity, include only the request-response pattern [1] : the first language is inspired by WSCL while the second one by Abstract WS-BPEL. We consider these two approaches as they represent the two ends of the spectrum of the different forms of request-response operations described above: in WSCL there is only one *ReceiveSend* event, while in Abstract WS-BPEL an arbitrary amount of other actions can be performed between two correlated `receive` and `reply` activities.

In both the two models that we present, the request-response interaction pattern is decomposed into sequences of more fundamental *send-receive-reply* steps: the client first *sends* its invocation, then the service *receives* such an invocation, and finally the service sends its *reply* message back to the client. The binding between the requesting and the responding sides (instances) of the original operation is maintained by employing naming mechanisms similar to those found in the π-calculus [22]. In both models, we do not put any restriction on the number of client or service instances that can be generated at runtime, so that the resulting systems are in general infinite-state. The difference between the two models is that in the former it is not possible to describe intermediate activities of the service taking place between the receive and the reply steps, while this is possible in the latter.

We define client-service compliance on the basis of the *must testing* relation of [14]: a client and a service are compliant if any sequence of interactions between them

[1] As discussed in Section 4, the languages that we propose are sufficiently expressive to model also the one-way communication pattern.

leads the client to a successful state. Our main results show that client-service compliance is decidable in the WSCL-inspired model, while it is undecidable in the Abstract WS-BPEL model: this points to an interesting expressiveness gap between the two approches for the modeling of the request-response interaction pattern. In the former case, the decidability proof is based on a translation of contracts into Petri nets. This translation is not precise, in the sense that intermediate steps of the request-response interaction are not represented in the Petri net. However, the translation is complete, in the sense that it preserves and reflects the existence of unsuccessful computations, which is enough to reduce the original compliance problem to a decidable problem in Petri nets. This yields a practical compliance-checking procedure, obtained by adaptation of the classical Karp-Miller coverability tree construction [19].

We check the robustness of our approach for verifying client-service compliance taking under consideration also different notions of compliance. In multiparty service compositions, for instance, there is no clear distinction between clients and services as one partner could play both the roles. In those cases, a symmetric notion of compliance in which all the involved partners should reach successful completion is more appropriate. We first define a more restrictive notion of compliance, that we call *mutual compliance*, that guarantees completion of both the client and the service in any possible computation. We show that our decision procedure can be slightly modified to cope also with this notion of compliance.

The rest of the paper is organized as follows. In Section 2 we present the two formal models and the definition of client-service compliance. Sections 3 contains the Petri nets semantics and the proof of decidability of client-service compliance for the WSCL model. Section 4 reports on undecidability for the Abstract WS-BPEL model. In Section 5 we consider *mutual compliance* and we show how to modify the decision procedure defined in Section 3 to cope with this symmetric version of compliance. Finally, in Section 6 we draw some conclusions and discuss related and further work.

2 Behavioural contracts with request-response

We presuppose a denumerable set of contract variables *Var* ranged over by X, Y, $\cdots$, a denumerable set of names *Names* ranged over by a, b, r, s, $\cdots$. We use I, J, $\cdots$ to denote a sets of indexes.

Definition 2.1 (WSCL Contracts) *The syntax of* WSCL contracts *is defined by the following grammar*

$$G ::= \textsf{\textit{invoke}}(a, \textstyle\sum_{i\in I} b_i.C_i) \mid \textsf{\textit{recreply}}(a, \textstyle\sum_{i\in I} b_i.C_i) \mid \surd$$
$$C ::= \textstyle\sum_{i\in I} G_i \mid C|C \mid X \mid recX.C$$

where recX._ is a binder for the contract variable X. We assume guarded recursion*, that is, given a contract recX.C all the free occurrences of X in C are inside a guarded contract G.*

A client contract *is a contract C containing at least one occurrence of the guarded* success *contract* $\surd$*, while a* service contract *is a contract not containing* $\surd$*.*

G is used to denote guarded contracts, ready to perform either an invoke or a receive on a request-response operation a: the selection of the continuation C_i depends on the actual reply message b_i. A set of guarded contracts G_i can be combined into a choice $\sum_{i\in I} G_i$; if the index set I is empty, we denote this term by $\mathbf{0}$. Contracts can be composed in parallel. Note that infinite-state contract systems can be defined using recursion (see example later in the section). In the following, we use $Names(C)$ to denote the set of names occurring in C, and C and S to denote respectively client and service contracts. Before presenting the semantics of WSCL contracts, we introduce BPEL contracts as well.

Definition 2.2 (BPEL Contracts) *BPEL contracts are defined like WSCL contracts in Definition 2.1, with the only difference that guarded contracts are as follows*

$$G ::= \textsf{\textit{invoke}}(a, \sum_{i\in I} b_i.C_i) \quad | \quad \textsf{\textit{receive}}(a).C \quad | \quad \textsf{\textit{reply}}(a,b).C \quad | \quad \surd .$$

We now define the operational semantics of both models. We start by observing that the WSCL contract $\textsf{recreply}(a, \sum_{i\in I} b_i.C_i)$ is the same as the BPEL contract $\textsf{receive}(a).\sum_{i\in I}(\textsf{reply}(a,b_i).C_i)$ that receives an invocation on the operation a and then replies with one of the messages b_i. We shall rely on a run-time syntax of contracts, which is obtained from the original one by extending the clause for guarded contract, thus $G ::= \cdots \mid \overline{a}\langle r\rangle \mid r\langle b\rangle.C$. Both terms $\overline{a}\langle r\rangle$ and $r\langle b\rangle.C$ are used to represent an emitted and pending invocation of a request-response operation a: the name r represents a (fresh) channel r that will be used by the invoked operation to send the reply message back to the invoker. From now onwards we will call (WSCL) contract any term that can be obtained from this run-time syntax. In the following, we let $Labels \triangleq \{\tau, \surd\} \cup \{a\langle b\rangle, \overline{a}\langle b\rangle, (a) \mid a, b \in Names\}$. Moreover, by $C\{^r/_a\}$ we denote the term obtained from C by replacing with r every occurrence of a not inside a $\textsf{receive}(a).D$, while $C\{^{recX.C}/_X\}$ denotes the usual substitution of free contract variables with the corresponding definition.

Definition 2.3 (Operational semantics) *The operational semantics of a contract is given by the minimal labeled transition system, with labels taken from the set Labels, satisfying the axiom and rules in Table 1.*

Table 1
Operational semantics of contracts.

$$\frac{G_l \xrightarrow{\alpha} G'_l \quad l \in I}{\sum_{i\in I} G_i \xrightarrow{\alpha} G'_l} \qquad \frac{r \notin Names(\sum_{i\in I} b_i.C_i)}{\mathsf{invoke}(a, \sum_{i\in I} b_i.C_i) \xrightarrow{(r)} \sum_{i\in I}(r\langle b_i\rangle.C_i) \mid \overline{a}\langle r\rangle}$$

$$\frac{r \notin Names(C)}{\mathsf{receive}(a).C \xrightarrow{a\langle r\rangle} C\{^r/_a\}} \qquad \mathsf{reply}(r,b).C \xrightarrow{\tau} C \mid \overline{r}\langle b\rangle \qquad \frac{C_1 \xrightarrow{\overline{a}\langle b\rangle} C'_1 \quad C_2 \xrightarrow{a\langle b\rangle} C'_2}{C_1|C_2 \xrightarrow{\tau} C'_1|C'_2}$$

$$\overline{r}\langle b\rangle \xrightarrow{\overline{r}\langle b\rangle} \mathbf{0} \qquad r\langle b\rangle.C \xrightarrow{r\langle b\rangle} C \qquad \surd \xrightarrow{\surd} \mathbf{0}$$

$$\frac{C_1 \xrightarrow{\alpha} C'_1 \quad \alpha \neq (r)}{C_1|C_2 \xrightarrow{\alpha} C'_1|C_2} \qquad \frac{C_1 \xrightarrow{(r)} C'_1 \quad r \notin Names(C_2)}{C_1|C_2 \xrightarrow{\alpha} C'_1|C_2} \qquad \frac{C\{^{recX.C}/_X\} \xrightarrow{\alpha} C'}{recX.C \xrightarrow{\alpha} C'}$$

($a, b, r \in Names$, symmetric version of the rules for parallel composition omitted)

In the following, we use $C \xrightarrow{\alpha}$ to say that there is some C' such that $C \xrightarrow{\alpha} C'$. Moreover, we use $C \longrightarrow C'$ to denote reductions, i.e. transitions that C can perform also when it is in isolation. Namely, $C \longrightarrow C'$ if $C \xrightarrow{\tau} C'$ or $C \xrightarrow{(r)} C'$ for some r.

We now formalize the notion of client-service compliance resorting to *must-testing* [14]. Intuitively, a client C is *compliant* with a service contract S if all the computations of the system $C|S$ lead to the client's success. Other notions of compliance have been put forward in the literature [7,8,9]; we have chosen this one because of its technical and conceptual simplicity (see e.g. [11]).

Definition 2.4 (Client-Service compliance) *Given a contract D, a* computation *is a sequence of reduction steps $D_1 \longrightarrow D_2 \longrightarrow \cdots \longrightarrow D_n \longrightarrow \cdots$. It is a* maximal computation *if it is infinite or it ends in a state D_n such that D_n has no outgoing reductions.*

A client contract C is compliant *with a service contract S if for every maximal computation $C|S \longrightarrow D_1 \longrightarrow \cdots \longrightarrow D_l \longrightarrow \cdots$ there exists k such that $D_k \xrightarrow{\surd}$.*

Example 2.5 (An impatient client and a latecomer service)
This example shows that even very simple WSCL scenarios could result in infinite-state systems. Consider a client C that asks the box office service S for some tickets and then waits for them by listening on offerTicket. Our client is impatient: at any time, it can decide to stop waiting and issue a new request. This behaviour can be described in WSCL as follows

$$C \triangleq recX.(\mathsf{invoke}(requireTicket, ok.X) + \mathsf{recreply}(offerTicket, ok.\surd))$$

Consider the box office service S, defined below, that is always ready to receive a requireTicket invocation and immediately responds by notifying (performing a call-back) on offerTicket.

$$S \triangleq recX.\mathsf{recreply}(requireTicket, ok.(\mathsf{invoke}(offerTicket,ok)|X))$$

It is easy to see that, in case $\mathsf{invoke}(offerTicket, \cdots)$ *on the service side and* $\mathsf{recreply}(offerTicket, \cdots)$ *on the client side never synchronize,* $C|S$ *generates an infinite-state system where each state is characterized by an arbitrary number of* $\mathsf{invoke}(offerTicket, \cdots)$*. This infinite computation, moreover, does not traverse states in which the client can perform its* $\surd$ *action, thus C is not compliant with S according to Definition 2.4.* [2]

3 Decidability of client-service compliance for WSCL contracts

We translate WSCL contract systems into place/transitions Petri nets [25], an infinite-state model in which several reachability problems are decidable (see, e.g., [16] for a review of decidable problems for finite Petri nets). The translation into Petri nets does not faithfully reproduce the operational semantics of contracts. In particular, in finite Petri nets it is not possible to represent the unbounded number of names dynamically created in contract systems to bind the reply messages to the corresponding invocations. The Petri net semantics that we present models bidirectional request-response interactions as a unique event, thus merging together the four distinct events in the operational semantics of contracts: the emission and the reception of the invocation, and the emission and the reception of the reply. We will prove that this alternative modeling preserves client-service compliance because in WSCL the invoker and the invoked contracts do not interact with other contracts during the request-response.

Another difference is that the Petri net semantics can be easily modified so that when the client contract enters in a successful state, i.e. a state with an outgoing transition $\surd$, the corresponding Petri net enters a particular successful state and blocks its execution. This way, a client contract is compliant with a service contract if and only if in the corresponding Petri net all computations are finite and finish in a *successful* state. As we will show, this last property is verifiable for finite Petri nets using the so-called *coverability* tree [19].

[2] Other definitions of compliance, see e.g. [7], resort to should-testing [26] instead of must-testing: according to these alternative definitions C and S turn out to be compliant due to the fairness assumption characterizing the should-testing approach.

We first recall the definition of Petri nets. For any set S, we let $\mathcal{M}_{fin}(S)$ be the set of the finite multisets (*markings*) over S.

Definition 3.1 (Petri net) *A Petri net is a pair $N = (S,T)$, where S is the set of* places *and $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions. *A transition (c,p) is written $c \Rightarrow p$. A transition $c \Rightarrow p$ is* enabled *at a marking m if $c \subseteq m$. The* execution of the transition *produces the marking $m' = (m \setminus c) \oplus p$ (where $\setminus$ and $\oplus$ are the multiset difference and union operators). This is written as $m[\rangle m'$. A* dead marking *is a marking in which no transition is enabled. A* marked *Petri net is a triple $N(m_0) = (S,T,m_0)$, where (S,T) is a Petri net and m_0 is the* initial marking. *A computation in $N(m_0)$ leading to the marking m is a sequence $m_0[\rangle m_1[\rangle m_2 \cdots m_n[\rangle m$.*

Note that in $c \Rightarrow p$, the marking c represents the tokens to be "consumed", while the marking p represents the tokens to be "produced". The Petri net semantics that we present for WSCL contracts decomposes contract terms into multisets of terms, that represents sequential contracts at different stages of invocation. We introduce the decomposition function in Definition 3.3. Instrumental to this definition is the set $\mathrm{Pl}(C)$, for C a WSCL contract, defined below.

Definition 3.2 ($\mathrm{Pl}(C)$) *For any contract C, let $C^{(k)}$ denote the term obtained by performing k unfolding of recursive definitions in C. Let k be the minimal integer s.t. in $C^{(k)}$ every $recX.D$ is guarded by one of the following prefixes: $\mathsf{invoke}(\cdot,\cdot), \mathsf{receive}(\cdot), \mathsf{reply}(\cdot,\cdot), a\langle b\rangle$. Then* $\mathrm{Pl}(C)$ *is defined as follows:*

$$\mathrm{Pl}(C) \triangleq \{\textstyle\sum_{i\in I} G_i,\ a\uparrow\sum_{i\in I} b_i.C_i,\ c\downarrow\sum_{i\in I} b_i.C_i \ :\ \sum_{i\in I} G_i,\ \sum_{i\in I} b_i.C_i \ occur\ in\ C^{(k)},\ a,c \in Names(C^{(k)})\}.$$

The function $dec(\cdot)$ transforms every WSCL contract C, as given in Definition 2.1, into a multiset $m \in \mathrm{Pl}(C)$.

Definition 3.3 (Decomposition) *The* decomposition $dec(C)$ *of a* WSCL *contract C, as given in Definition 2.1, is $dec_C(C)$. The auxiliary function $dec_C(D)$ is defined in Table 2 by lexicographic induction on the pair (n_1,n_2), where n_1 is the number of unguarded (i.e. not under an $\mathsf{invoke}(\cdot,\cdot), \mathsf{receive}(\cdot), \mathsf{reply}(\cdot,\cdot), a\langle b\rangle$) sub-terms of the form $recX.D'$ in D and n_2 is the syntactic size of D.*

There are three kinds of transitions in the Petri net we are going to define:

- transitions representing the emission of an invocation;
- transitions representing (atomically) the reception of the invocation and the emission and reception of the reply;
- and transitions representing (atomically) the reception of the invocation and the emission of a reply that will never be received by the invoker because it is outside

Table 2
The auxiliary function $dec_C(D)$.

$$dec_C(\sum_{i\in I} r\langle b_i\rangle.D_i) = a\uparrow\sum_{i\in I} b_i.D_i \quad \text{if } \overline{a}\langle r\rangle \text{ occurs in } C$$

$$dec_C(\sum_{i\in I} r\langle b_i\rangle.D_i) = c\downarrow\sum_{i\in I} b_i.D_i \quad \text{if } \overline{r}\langle c\rangle \text{ occurs in } C \text{ and } c \neq b_i \text{ for every } i \in I$$

$$dec_C(recX.D) = dec_C(D\{^{recX.D}/_X\}) \qquad dec_C(\overline{a}\langle b\rangle) = \emptyset$$

$$dec_C(D_1|D_2) = dec_C(D_1) \oplus dec_C(D_2) \qquad dec_C(D) = D, \text{ otherwise}$$

Table 3
Transitions schemata for the Petri net semantics of WSCL contracts.

$$\{\sum_{i\in I} G_i\} \Rightarrow \{a\uparrow\sum_{j\in J} b_j.C_j\} \qquad \text{if } G_k = \mathsf{invoke}(a, \sum_{j\in J} b_j.C_j) \text{ for some } k \in I$$

$$\{a\uparrow\sum_{j\in J} b_j.C_j, \sum_{i\in I} G_i\} \Rightarrow\ dec(C_y) \oplus dec(D_z)$$

$$\text{if}\begin{cases} G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l) \text{ for some } k \in I \text{ and} \\ b_y = c_z \text{ for some } y \in J,\, z \in L \end{cases}$$

$$\{a\uparrow\sum_{j\in J} b_j.C_j, \sum_{i\in I} G_i\} \Rightarrow\ \{c_z\downarrow\sum_{j\in J} b_j.C_j\} \oplus dec(D_z)$$

$$\text{if}\begin{cases} G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l) \text{ for some } k \in I \text{ and} \\ \text{there exists } z \in L \text{ s.t. } c_z \neq b_j \text{ for every } j \in J \end{cases}$$

the set of admitted replies.

These three cases are taken into account in the definition below.

Definition 3.4 (Petri net semantics) *Let C be a* WSCL *contract system as in Definition 2.1. We define $Net(C)$ as the Petri net (S,T) where:*

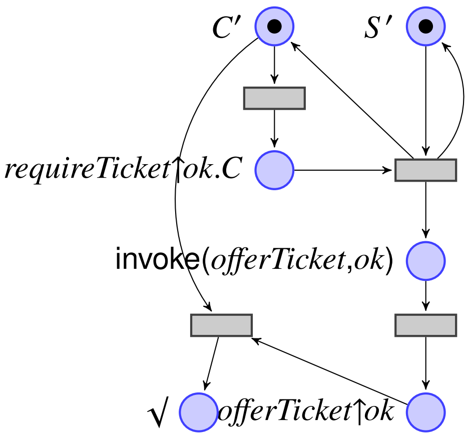
- $S = \mathrm{Pl}(C)$;
- $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ *includes all the transitions that are instances of the transitions schemata in Table 3.*

We define the marked net $Net^{\mathrm{m}}(C)$ *as the marked net* (S,T,m_0)*, where the initial marking is* $m_0 = dec(C)$.

Example 3.5 *Consider the client C and the service S introduced in Example 2.5 and let*

$$C' \triangleq \mathsf{invoke}(requireTicket, ok.C)$$
$$+\mathsf{recreply}(offerTicket, ok.\surd)$$
$$S' \triangleq \mathsf{recreply}(requireTicket,$$
$$ok.(\mathsf{invoke}(offerTicket,ok)|S)).$$

The marked net $Net^{\mathrm{m}}(S|C)$ is depicted on the right. A bunch of unreachable places (like $ok\downarrow ok.\surd$, $ok\uparrow ok.\surd, \ldots$) have been omitted for the sake of clarity.



C'
S'
$requireTicket\uparrow ok.C$
$\mathsf{invoke}(offerTicket,ok)$
$\surd$
$offerTicket\uparrow ok$

We divide the proof of the correspondence between the operational and the Petri net semantics of WSCL contracts in two parts: we first prove a *soundness* result showing that all Petri net computations reflect computations of contracts, and then a *completeness* result showing that contract computations leading to a state in which there are no uncompleted request-response interactions are reproduced in the Petri net.

In the proof of the soundness result we use the following structural congruence rule to remove empty contracts and in order to rearrange the order of contracts in parallel compositions. Let $\equiv$ be the minimal congruence for contract systems such as

$$C|\mathbf{0} \equiv C \qquad C|D \equiv D|C \qquad C|(D|E) \equiv (C|D)|E \qquad recX.C \equiv C\{^{recX.C}/_X\}$$

As usual, we have that the structural congruence respects the operational semantics.

Proposition 3.6 *Let C and D be two contract systems such that $C \equiv D$. If $C \xrightarrow{\alpha} C'$, then there exists D' such that $D \xrightarrow{\alpha} D'$ and $C' \equiv D'$.*

The following result establishes a precise relationship between the form of m and the form of C when $dec(C) = m$.

Lemma 3.7 *Let C be a WSCL contract system and suppose $dec(C) = m$. The following holds:*

(1) if $m = \{\sum_{i\in I} G_i\} \oplus m'$ then $C \equiv \sum_{i\in I} G_i \mid D$, for some D such that $dec(D) = m'$;
(2) if $m = \{a\uparrow\sum_{j\in J} b_j.C_j\} \oplus m'$ then $C \equiv \sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{a}\langle r\rangle \mid D$, for some D and r such that $r \notin Names(D)$ and $dec(D) = m'$;
(3) if $m = \{c\downarrow\sum_{j\in J} b_j.C_j\} \oplus m'$ then $c \neq b_j$ for each $j \in J$ and $C \equiv \sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{r}\langle c\rangle \mid D$, for some D and r such that $r \notin Names(D)$ and $dec(D) = m'$.

Proposition 3.8 *Let C be a* WSCL *contract. Consider the Petri net $Net(C) = (S, T)$ and a marking m of $Net(C)$. We have that m is dead if and only if D has no outgoing reductions, for every D such that $dec(D) = m$.*

Proof: ($\Rightarrow$). Suppose m is dead, we have to prove that any D, with $dec(D) = m$, has no outgoing reductions. The proof is by induction on the structure of m. The case $m = \emptyset$ is trivial.

Suppose $m = \{\sum_{i\in I} G_i\} \oplus m'$, hence $D \equiv \sum_{i\in I} G_i | C$, with $dec(C) = m'$ (Lemma 3.7). By definition, m' is dead, therefore, by induction, C has no outgoing reductions. Moreover:

- $G_k \neq \mathsf{invoke}(a, \sum_{j\in J} b_j.C_j)$, for each $k \in I$ (otherwise the first kind of transition would apply to m), hence $\sum_{i\in I} G_i$ has no outgoing reductions.
- $G_k = \mathsf{recreply}(a_k, \sum_{l\in L_k} c_l.D_l)$ but m' does not contain $a_k \uparrow \sum_{j\in J} b_j.C_j$, for each $k \in I$ (otherwise either the second or the third kind of transition would apply to m). Therefore there is no $\overline{a_k}\langle r\rangle \mid \sum_{j\in J} r\langle b_j\rangle.C_j$ in C (by $dec(C) = m'$) and D has not outgoing reductions.

Suppose $m = \{a \uparrow \sum_{j\in J} b_j.C_j\} \oplus m'$, hence $D \equiv \sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{a}\langle r\rangle \mid C$, with $r \notin Names(C)$ and $dec(C) = m'$ (Lemma 3.7). By definition, m' is dead, therefore, by induction, C has no outgoing reductions. Moreover $m' \neq \{\sum_{i\in I} G_i\} \oplus m''$ with $G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$, for some $k \in I$. Otherwise either the second or the third kind of transition would apply to m. Hence, by definition of $dec(\cdot)$, C cannot contain an unguarded subterm of the form $\mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$ and D has no outgoing reductions.

Suppose $m = \{c \downarrow \sum_{j\in J} b_j.C_j\} \oplus m'$. Then $c \neq b_j$, for each j and $D \equiv \sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{r}\langle c\rangle \mid C$, with $dec(C) = m'$ (Lemma 3.7). By definition, m' is dead, therefore, by induction, C has no outgoing reductions and by inspection of the semantics of contracts it is easy to see that D has no outgoing reductions too.

($\Leftarrow$). By induction on the structure of m and by Lemma 3.7 it can be easily seen that if D has an outgoing reduction then we have a contradiction and m is not dead. $\square$

In order to prove that the Petri net semantics preserves client-service compliance, we need to introduce the notion of *success* marking. A *success* marking m contain at least one token in a place corresponding to a successful client state, formally, $m(\sum_{i\in I} G_i) > 0$ for some contract $\sum_{i\in I} G_i$ such that $G_k = \surd$, for some $k \in I$.

We are now ready to prove the *soundness* result.

Theorem 3.9 (Soundness) *Let C be a* WSCL *contract. Consider the Petri net $Net(C) = (S, T)$ and let m be a marking of $Net(C)$. If $m[\rangle m'$ then for each D such that $dec(D) = m$ there exists a computation $D \triangleq D_0 \longrightarrow D_1 \longrightarrow \cdots \longrightarrow D_l$,* *with $dec(D_l) = m'$. Moreover, if m is not a success marking then there exists no $j \in \{0, \cdots, l-1\}$ such that $D_j \xrightarrow{\surd}$.*

Proof: The proof proceeds by case analysis on the three possible kinds of transition.

(1) If $m[\rangle m'$ by applying the first kind of transition then $m = \{\sum_{j\in J} G_j\} \oplus m''$, with $G_k = \mathsf{invoke}(a, \sum_{i\in I} b_i.C_i)$ for a $k \in J$. Moreover, $m' = \{a\uparrow\sum_{i\in I} b_i.C_i\} \oplus m''$.

By Lemma 3.7, $D \equiv \sum_{j\in J} G_j \mid C' \longrightarrow \overline{a}\langle r\rangle \mid \sum_{i\in I} r\langle b_i\rangle.C_i \mid C' \stackrel{\triangle}{=} D'$ and $dec(D') = m'$.

(2) If $m[\rangle m'$ by applying the second kind of transition then $m = \{a \uparrow \sum_{j\in J} b_j.C_j, \sum_{i\in I} G_i\} \oplus m''$, with $G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$, for some $k \in J$, and $b_y = c_z$ for some $y \in J$ and $z \in L$. By Lemma 3.7, if $dec(D) = m$ then $D \equiv \overline{a}\langle r\rangle \mid \sum_{j\in J} r\langle b_j\rangle.C_j \mid \sum_{i\in I} G_i \mid C'$, for any C' such that $dec(C') = m''$. Therefore, by $G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$:

$$\begin{aligned} D &\longrightarrow \textstyle\sum_{j\in J} r\langle b_j\rangle.C_j \mid \sum_{l\in L} \mathsf{reply}(r,c_l).D_l \mid C' \\ &\longrightarrow \textstyle\sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{r}\langle c_z\rangle \mid D_z \mid C' \\ &\longrightarrow C_y \mid D_z \mid C' \quad \stackrel{\triangle}{=} D' \end{aligned}$$

with $dec(D') = dec(C_y) \oplus dec(D_z) \oplus m'' = m'$. Notice that each intermediate state in the reduction sequence from D to D' cannot perform a successful transition.

(3) If $m[\rangle m'$ by applying the third kind of transition then $m = \{a \uparrow \sum_{j\in J} b_j.C_j, \sum_{i\in I} G_i\} \oplus m''$, with $G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$, for some $k \in J$, and there is $z \in L$ such that $b_y \neq c_z$ for each $y \in J$. By Lemma 3.7, if $dec(D) = m$ then $D \equiv \overline{a}\langle r\rangle \mid \sum_{j\in J} r\langle b_j\rangle.C_j \mid \sum_{i\in I} G_i \mid C'$, for any C' such that $dec(C') = m''$. Therefore, by $G_k = \mathsf{recreply}(a, \sum_{l\in L} c_l.D_l)$, we get

$$\begin{aligned} D &\longrightarrow \textstyle\sum_{j\in J} r\langle b_j\rangle.C_j \mid \sum_{l\in L} \mathsf{reply}(r,c_l).D_l \mid C' \\ &\longrightarrow \textstyle\sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{r}\langle c_z\rangle \mid D_z \mid C' \stackrel{\triangle}{=} D' \end{aligned}$$

with $dec(D') = \{c_z\downarrow\sum_{j\in J} b_j.C_j\} \oplus dec(D_z) \oplus m'' = m'$. Notice that each intermediate state in the reduction sequence from D to D' cannot perform a successful transition.

□

Definition 3.10 (Stable contracts) *A* WSCL *contract C (in the run-time syntax) is said* stable *if it contains neither unguarded* $\mathsf{reply}(r,b)$ *actions nor pairs of matching terms of the form $\overline{r}\langle b\rangle$ and $r\langle b\rangle$.*

Notice that any *initial* WSCL contract (according to the syntax of Definition 2.1) is stable.

Lemma 3.11 *Suppose C is stable and that $C \longrightarrow C'$. Then, there exists C'' stable such that $C' \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_l \longrightarrow C''$ $(l \geq 0)$ and for each $i = 1, \cdots, l$ it holds* *that $C_i \overset{\surd}{\not\longrightarrow}$.*

Proof: If C' is not stable then it may contain both unguarded reply actions and pairs of the form $\overline{r}\langle b\rangle$ and $r\langle b\rangle$. According to the operational semantics of contracts, all unguarded reply actions and $\overline{r}\langle b\rangle$ and $r\langle b\rangle$ can be consumed performing a sequence of reductions. Therefore a stable contract C'' can be reached from C' without traversing any state capable of $\overset{\surd}{\longrightarrow}$. □

We now move to the completeness part.

Theorem 3.12 (Completeness) *Let C be a* WSCL *contract and let D be a contract reachable from C through the computation $C = C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n = D$. If D is stable then there exists a computation $m_0[\rangle m_1[\rangle m_2 \cdots m_{l-1}[\rangle m_l$ of the marked Petri net $Net^{\mathrm{m}}(C)$ such that $dec(D) = m_l$. Moreover, if there exists no $k \in \{0, \cdots, n\}$ such that $C_k \overset{\surd}{\longrightarrow}$ then for every $j \in \{0, \cdots, l\}$ we have that m_j is not a success marking.*

Proof: The proof is by induction on the length n of the derivation $C = C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n = D$. The base case ($n = 0$) is trivial. In the inductive case there are two possible cases: C_{n-1} is stable or it is not stable. In the first case the proof is straightforward. In the second case, there are two possible scenarios to be considered: either C_n contains an unguarded action reply(r,b) term, or it contains a pair of matching terms $\overline{r}\langle b\rangle$ and $r\langle b\rangle$. We consider the first of these two cases, the second one can be treated similarly.

Let C_{n-1} be a non stable contract containing an unguarded action reply(r,b). This action cannot appear unguarded in the initial contract C: let C_j, with $j > 0$, be the first contract traversed during the computation of C in which the action reply(r,b) appears unguarded. Hence, we have that $C_{j-1} \longrightarrow C_j$ consists of the execution of a receive action. We now consider a different computation from C to D obtained by rearranging the order of the steps in the considered computation $C = C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n = D$. Namely, let $C = C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_{l-1} \longrightarrow C'_l \longrightarrow \cdots \longrightarrow C'_{n-2} \longrightarrow C_{n-1} \longrightarrow C_n = D$ be the computation obtained by delaying as much as possible the execution of the receive action generating the unguarded action reply(r,b). In the new computation, this action appears for the first time in the contract C_{n-1}. Moreover, C'_{n-2} must be a stable contract otherwise C_n is not stable. Hence, we can straightforwardly prove the thesis by applying the inductive hypothesis to the shorter computation $C = C_0 \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_{l-1} \longrightarrow C'_l \longrightarrow \cdots \longrightarrow C'_{n-2}$ leading to the stable contract C'_{n-2}. □

As a simple corollary of the last two theorems, we have that client-service compliance is preserved by the Petri net semantics.

Corollary 3.13 (Compliance preservation) *Let C and S be respectively a* WSCL *client and service contract, as in Definition 2.1. We have that C is compliant with*

S if and only if in the marked Petri net $Net^{\mathrm{m}}(C|S)$ all the maximal computations traverse at least one success marking.

Proof: ($\Rightarrow$). Trivial by Theorem 3.9.

($\Leftarrow$). Suppose that in $Net^{\mathrm{m}}(C|S)$ all the maximal computations traverse at least one success marking and suppose by contradiction that C is not compliant with S. This means that there is a maximal computation from $C|S$ that does not traverse a state D such that $D \xrightarrow{\surd}$. This computation can either end in a state D' with no outgoing reductions or can be infinite.

In the first case we get a contradiction by Theorem 3.12. Indeed there would be a maximal computation from $Net^{\mathrm{m}}(C|S)$ traversing only non-success markings.

Consider the second case. From the infinite sequence of reductions, we can build an infinite set of maximal computations of arbitrary length, starting from C and ending in a stable state (Lemma 3.11) without traversing a succes state. By Theorem 3.12, for each of these maximal computations there exists a corresponding maximal computation in the net $Net^{\mathrm{m}}(S|C)$ that does not traverse a success marking. We can arrange these computations so as to form a tree where m' is a child of m iff $m[\rangle m'$: this is an infinite, but finitely-branching, tree. By König's lemma, in $Net^{\mathrm{m}}(S|C)$ there exists then an infinite computation that does not traverse a success marking and we get a contradiction. $\square$

3.2 Verifying client-service compliance using the Petri net semantics

In the light of Corollary 3.13, checking whether C is compliant with S reduces to verifying if all the maximal computations in $Net^{\mathrm{m}}(C|S)$ traverse at least one success marking. In order to verify this property, we proceed as follows:

- we first modify the net semantics in such a way that the net computations block if they reach a success marking;
- we define a (terminating) algorithm for checking whether in the modified Petri net all the maximal computations are finite and end in a success marking.

The modified Petri net semantics simply adds one place that initially contains one token. All transitions consume such a token, and reproduce it only if they do not introduce tokens in success places, i.e., places $\sum_{i \in I} G_i$ such that $G_k = \surd$ for some $k \in I$

Definition 3.14 (Modified Petri net semantics) *Let C be a* WSCL *contract and $Net(C) = (S, T)$ the corresponding Petri net as defined in Definition 3.4. We define $ModNet(C)$ as the Petri net (S', T') where:*

- $S' = S \cup \{run\}$, *where run is an additional place;*
- *for each transition* $c \Rightarrow p \in T$, *then* T' *contains a transition that consumes the multiset* $c \uplus \{run\}$ *and produces either* p, *if* p *contains a place* $\sum_{i \in I} G_i$ *such that* $G_k = \surd$ *for some* $k \in I$, *or* $p \uplus \{run\}$, *otherwise.*

The marked modified net $ModNet^{\mathrm{m}}(C)$ *is defined as the net* $ModNet(C)$ *with initial marking* m_0 *where*

$$m_0 = \begin{cases} dec(C) \uplus \{run\} & \textit{if } dec(C) \textit{ is not a success marking} \\ dec(C) & \textit{otherwise.} \end{cases}$$

We now state an important relationship between $Net^{\mathrm{m}}(C)$ and $ModNet^{\mathrm{m}}(C)$. It can be proved by relying on the definition of modified net.

Proposition 3.15 *Let* C *be a* WSCL *contract,* $Net^{\mathrm{m}}(C)$ *(resp.* $ModNet^{\mathrm{m}}(C)$*) the corresponding Petri net (resp. modified Petri net). We have that all the maximal computations of* $Net^{\mathrm{m}}(C)$ *traverse at least one success marking if and only if in* $ModNet^{\mathrm{m}}(C)$ *all the maximal computations are finite and end in a success marking.*

We now present the algorithm for checking whether in a Petri net all the maximal computations are finite and end in a success marking. In the algorithm and in the proof, we utilize the following preorder over multisets on Places(C): $m \leq m'$ iff for each p, $m(p) \leq m'(p)$. It can be shown that this preorder is a *well-quasi-order*, that is, in any infinite sequence of multisets there is a pair of multisets m and m' such that $m \leq m'$ (see e.g. [17]).

Theorem 3.16 *Let* C *be a* WSCL *contract as in Definition 2.1 and let* $ModNet^{\mathrm{m}}(C) = (S, T, m_0)$ *be the corresponding modified Petri net. The algorithm described in Table 4 always terminates. Moreover, it returns TRUE iff all the maximal computations in* $ModNet^{\mathrm{m}}(C)$ *are finite and end in a success marking.*

Proof: Suppose by contradiction that the algorithm does not terminate. This means that there exists an infinite computation from m_0 of the form $m_0[\rangle m_1[\rangle \cdots [\rangle m_n[\rangle \cdots$ such that, for each each m_i: *(i)* m_i is not a success marking and *(ii)* for no m_j, with $0 \leq j < i$, it holds that $m_j \leq m_i$.

The last assertion implies that there exists an infinite sequence of elements in S that are not related by the preorder $\leq$, and this would violate the fact $\leq$ is a well-quasi-order.

Assume now that the algorithm returns FALSE. This may happen at (b) or at (c)-ii. In the first case, we have found a maximal computation ending at m and not traversing a success state. In the second case, it is easy to see that we can build computations of arbitrary length that do not traverse success, again implying the existence of an infinite unsuccessful computation (via König's lemma). The case

Table 4
An algorithm for checking the coverability of success markings.

```
(1) If the initial marking $m_0$ is not a success marking then label it as the root and tag it "new".
(2) While "new" markings exist do the following:
    (a) Select a "new" marking $m$.
    (b) If no transitions are enabled at $m$, return FALSE.
    (c) While there exist enabled transitions at $m$, do the following for each of them:
        (i) Obtain a marking $m'$ that results from firing the transition.
        (ii) If on the path from the root to $m$ there exists a marking $m''$ such that $m'(p) \geq m''(p)$ for each place $p$ then return FALSE.
        (iii) If $m'$ is not a success marking introduce $m'$ as a node, draw an arc from $m$ to $m'$, and tag $m'$ "new".
    (d) Remove the tag "new" from the marking $m$.
(3) Return TRUE.
```

when the algorithm returns TRUE is obvious. □

4 Undecidability of client-service compliance for BPEL contracts

We now move to the proof that client-service compliance is undecidable for BPEL contracts. The proof is by reduction from the termination problem in Random Access Machines (RAMs) [23], a well known Turing powerful formalism based on registers containing nonnegative natural numbers. The registers are used by a program, that is a set of indexed instructions I_i which are of two possible kinds:

- $i : Inc(r_j)$ that increments the register r_j and then moves to the execution of the instruction with index $i+1$ and
- $i : DecJump(r_j, s)$ that attempts to decrement the register r_j; if the register does not hold 0 then the register is actually decremented and the next instruction is the one with index $i+1$, otherwise the next instruction is the one with index s.

Without loss of generality we assume that given a program $I_1, \cdots, I_n$, it starts by executing I_1 with all the registers empty (i.e. all registers contain 0) and terminates when trying to perform the first undefined instruction I_{n+1}.

In order to simplify the notation, in this section we introduce a notation corresponding to standard input and output prefixes [3] of CCS [21]. Namely, we model simple synchronization as a request-response interaction in which there is only one possi-

[3] The input and output prefixes correspond also to the representation of the one-way interaction pattern in contract languages such as those in [11,7,12].

ble reply message. Assuming that this unique reply message is *ok* (with $ok \in Names$ not necessarily fresh), we introduce the following notation:

$$\overline{a}.P \; = \; \mathsf{invoke}(a,ok.P) \qquad\qquad a.P \; = \; \mathsf{receive}(a).\mathsf{reply}(a,ok).P$$

In order to reduce RAM termination to client-service compliance, we define a client contract that simulates the execution of a RAM program, and a service contract that represent the registers, such that the client contract reaches the success $\surd$ if and only if the RAM program terminates.

Given a RAM program $I_1, \cdots, I_n$, we consider the client contract C as follows

$$C \triangleq \textstyle\prod_{i\in\{1,\cdots,n\}} [\![I_i]\!] \mid inst_{n+1}.\surd$$

$$[\![I_i]\!] \triangleq \begin{cases} recX.(inst_i.\overline{inc_j}.ack.(\overline{inst_{i+1}} \mid X)) & \text{if } I_i = (i : Inc(r_j)) \\ recX.(inst_i.\overline{dec_j}.(ack.(\overline{inst_{i+1}} \mid X) + zero.(\overline{inst_s} \mid X))) & \text{if } I_i = (i : DecJump(r_j, s)) \end{cases}$$

An increment instruction $Inc(r_j)$ is modeled by a recursive contract that invokes the operation inc_j, waits for an acknowledgement on *ack*, and then invokes the service corresponding to the subsequent instruction. On the contrary, a decrement instruction $DecJump(r_j, s)$ invokes the operation dec_j and then waits on two possible operations: *ack* or *zero*. In the first case the service corresponding to the subsequent instruction with index $i+1$ is invoked, while in the second case the service corresponding to the target of the jump is invoked instead.

We now move to the modeling of the registers. Each register r_j is represented by a contract representing the initially empty register in parallel with a service modeling every unit subsequently added to the register

$$\begin{aligned} [\![r_j]\!] \; = \; & recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.X)) \mid \\ & recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j,ok) + \\ & \qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y))). \end{aligned}$$

The idea of the encoding is to model numbers with chains of nested request-response interactions. When a register is incremented, a new instance of a contract is spawn invoking the operation $unit_j$, and a request-response interaction is opened between the previous instance and the new one. In this way, the previous instance blocks waiting for the reply. When an active instance receives a request for decrement, it terminates by closing the request-response interaction with its previous instance, which is then re-activated. The contract that is initially active represents the empty register because it replies to decrement requests by performing an invocation on the *zero* operation.

We extend structural congruence $\equiv$, introduced in Section 3, to $\equiv_{ren}$ to admit the injective renaming of the operation name

$$C \equiv_{ren} D \;\text{ if there exists an injective renaming } \sigma \text{ such that } C\sigma \equiv D$$

Clearly, injective renaming is an equivalence and preserves the operational semantics.

Proposition 4.1 *Let C and D be two contract systems such that $C \equiv_{ren} D$. If $C \xrightarrow{\alpha} C'$, then there exists D' and a label α' obtained by renaming the operation names in α such that $D \xrightarrow{\alpha'} D'$ and $C' \equiv_{ren} D'$.*

Now, we introduce $\{\!\!\{ r_j, c \}\!\!\}$ that we use to denote the modeling of the register r_j when it holds the value c. Namely, $\{\!\!\{ r_j, 0 \}\!\!\} \;=\; \llbracket r_j \rrbracket$, while if $c > 0$ then

$$\{\!\!\{ r_j, c \}\!\!\} = \begin{cases} b_0\langle ok\rangle.\overline{ack}.recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.X)) \mid \\ b_1\langle ok\rangle.\overline{ack}.recY.(dec_j.\overline{b_0}\langle ok\rangle + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y)) \mid \\ \cdots \mid \\ recY.(dec_j.\overline{b_{c-1}}\langle ok\rangle + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y)) \mid \\ recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j,ok) + \\ \qquad\qquad\qquad\qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y))) \end{cases}$$

In the following theorem, stating the correctness of our encoding, we use the following notation: $(i, c_1, \cdots, c_m)$ to denote the state of a RAM in which the next instruction to be executed is I_i and the registers $r_1, \cdots, r_m$ respectively contain the values $c_1, \cdots, c_m$, and $(i, c_1, \cdots, c_m) \rightarrow_R (i', c'_1, \cdots, c'_m)$ to denote the change of the state of the RAM R due to the execution of the instruction I_i.

Theorem 4.2 *Consider a RAM R with instructions $I_1, \cdots, I_n$ and registers $r_1, \cdots, r_m$. Consider also a state $(i, c_1, \cdots, c_m)$ of the RAM R and a corresponding contract C such that $C \equiv_{ren} \overline{inst_i} \mid \llbracket I_1 \rrbracket \mid \cdots \mid \llbracket I_n \rrbracket \mid inst_{n+1}.\surd \mid \{\!\!\{ r_1, c_1 \}\!\!\} \mid \cdots \mid \{\!\!\{ r_m, c_m \}\!\!\}$. We have that*

- *either the RAM computation has terminated, thus $i = n+1$*
- *or $(i, c_1, \cdots, c_m) \rightarrow_R (i', c'_1, \cdots, c'_m)$ and there exists $l > 0$ such that $C \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_l$ and*
  - $\cdot$ $C_l \equiv_{ren} \overline{inst_{i'}} \mid \llbracket I_1 \rrbracket \mid \cdots \mid \llbracket I_n \rrbracket \mid inst_{n+1}.\surd \mid \{\!\!\{ r_1, c'_1 \}\!\!\} \mid \cdots \mid \{\!\!\{ r_m, c'_m \}\!\!\}$
  - $\cdot$ *for each k ($1 \le k < l$): $C_k \overset{\surd}{\not\longrightarrow}$*

Proof: Suppose $i \neq n+1$. The proof proceeds by distinguishing two cases depending on the instruction i:

$i : Inc(r_j)$**:** for the sake of simplicity, suppose $r_j = 0$. Then $(i, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m) \rightarrow_R (i+1, c_1, \cdots, c_{j-1}, 1, c_{j+1}, \cdots, c_m)$.

The contract C corresponding to $(i, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m)$ is:

$$\begin{aligned}
C \equiv_{ren} D \triangleq\ & \overline{inst_i} \mid \cdots \mid recX.(inst_i.\overline{inc_j}.ack.(\overline{inst_{i+1}} \mid X)) \mid \cdots \mid inst_{n+1}.\surd \mid \{\!\{r_1, c_1\}\!\} \mid \cdots \\
& \mid recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.X)) \mid \\
& \mid recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j, ok) + \\
& \qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y))) \\
& \mid \cdots \mid \{\!\{r_m, c_m\}\!\}
\end{aligned}$$

and

$$\begin{aligned}
D \rightarrow^*\ & \cdots \mid ack.(\overline{inst_{i+1}} \mid [\![I_i]\!]) \mid \cdots \mid inst_{n+1}.\surd \mid \{\!\{r_1, c_1\}\!\} \mid \cdots \\
& \mid \overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.X))) \mid \\
& \mid recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j, ok) + \\
& \qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y))) \\
& \mid \cdots \mid \{\!\{r_m, c_m\}\!\} \\
\rightarrow^*\ & \cdots \mid ack.(\overline{inst_{i+1}} \mid [\![I_i]\!]) \mid \cdots \mid inst_{n+1}.\surd \mid \{\!\{r_1, c_1\}\!\} \mid \cdots \\
& \mid r\langle ok\rangle.\overline{ack}.recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.X)) \mid \\
& \mid \overline{ack}.recY.(dec_j.\overline{r}\langle ok\rangle + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y)) \\
& \mid recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j, ok) + \\
& \qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y))) \\
& \mid \cdots \mid \{\!\{r_m, c_m\}\!\} \\
\rightarrow\ & \overline{inst_{i+1}} \mid \cdots \mid [\![I_i]\!] \mid \cdots \mid inst_{n+1}.\surd \mid \{\!\{r_1, c_1\}\!\} \mid \cdots \\
& \mid r\langle ok\rangle.\overline{ack}.recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.X)) \mid \\
& \mid recY.(dec_j.\overline{r}\langle ok\rangle + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y)) \\
& \mid recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j, ok) + \\
& \qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j, ok.\overline{ack}.Y))) \\
& \mid \cdots \mid \{\!\{r_m, c_m\}\!\} \triangleq D'
\end{aligned}$$

where D' corresponds to the state $(i+1, c_1, \cdots, c_{j-1}, 1, c_{j+1}, \cdots, c_m)$ and clearly each D'' in the derivation from D to D' cannot perform a successful transition. Therefore, by Proposition 4.1, $C \rightarrow^* C'$ with $C' \equiv_{ren} D'$.

$i : DecJump(r_j, s)$**:** suppose again that $r_j = 0$. Then, $(i, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m) \rightarrow_R (s, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m)$.

The contract C corresponding to $(i, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m)$ is:

$$
\begin{aligned}
C \equiv_{ren} D \triangleq\ & \overline{inst_i} \,|\, \cdots \,|\, recX.(inst_i.\overline{dec_j}.(ack.(\overline{inst_{i+1}}|X) + zero.(\overline{inst_s}|X))) \,|\, \cdots \,| inst_{n+1}.\surd| \\
& \{\!\!\{r_1, c_1\}\!\!\} \,|\, \cdots \,|\, recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.X)) \,| \\
& |\ recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j,ok)+ \\
& \qquad\qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y))) \\
& | \cdots | \{\!\!\{r_m, c_m\}\!\!\}
\end{aligned}
$$

and

$$
\begin{aligned}
D \to^*\ & \cdots | ack.(\overline{inst_{i+1}} | [\![I_i]\!]) + zero.(\overline{inst_s} | [\![I_i]\!])) \,|\, \cdots | inst_{n+1}.\surd | \{\!\!\{r_1, c_1\}\!\!\} \,|\, \cdots \\
& |\ \overline{zero}.\ recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.X)) \,| \\
& |\ recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j,ok)+ \\
& \qquad\qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y))) \\
& | \cdots | \{\!\!\{r_m, c_m\}\!\!\} \\
\to^*\ & \overline{inst_s} \,|\, \cdots \,|\, [\![I_i]\!] \,|\, \cdots | inst_{n+1}.\surd | \{\!\!\{r_1, c_1\}\!\!\} \,|\, \cdots \\
& |\ recX.(dec_j.\overline{zero}.X + inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.X)) \,| \\
& |\ recX.unit_j.(X \mid \mathsf{receive}(u_j).\overline{ack}.recY.(dec_j.\mathsf{reply}(u_j,ok)+ \\
& \qquad\qquad\qquad\qquad inc_j.\overline{unit_j}.\mathsf{invoke}(u_j,ok.\overline{ack}.Y))) \\
& | \cdots | \{\!\!\{r_m, c_m\}\!\!\} \triangleq D'
\end{aligned}
$$

where D' corresponds to the state $(s, c_1, \cdots, c_{j-1}, 0, c_{j+1}, \cdots, c_m)$ and clearly each D'' in the derivation from D to D' cannot perform a successful transition. Again, by Proposition 4.1, $C \to^* C'$ with $C' \equiv_{ren} D'$.

The proof proceeds similarly in case $r_j \neq 0$.

□

As a corollary we get that client-service compliance is undecidable.

Corollary 4.3 *Consider a RAM R with instructions $I_1, \cdots, I_n$ and registers $r_1, \cdots, r_m$. Consider the client contract $C = \overline{inst_1} | [\![I_1]\!] | \cdots | [\![I_n]\!] | inst_{n+1}.\surd$ and the service contract $S = \{\!\!\{r_1, 0\}\!\!\} | \cdots | \{\!\!\{r_m, 0\}\!\!\}$. We have that C is compliant with S if and only if R terminates.*

Proof: The proof proceeds by proving that both directions hold in the most general case: $C|S \equiv_{ren} \overline{inst_1} | [\![I_1]\!] | \cdots | [\![I_n]\!] | inst_{n+1}.\surd | \{\!\!\{r_1, 0\}\!\!\} | \cdots | \{\!\!\{r_m, 0\}\!\!\}$.

($\Leftarrow$): Suppose R terminates. Theorem 4.2 can be applied to guarantee that C and S are compliant. The proof proceeds by induction on the number n of steps needed by R to terminate.

Suppose $n = 0$, hence R has terminated. In this case $C \equiv_{ren} \overline{inst_{n+1}} | [\![I_1]\!] | \cdots | [\![I_n]\!] | inst_{n+1}.\surd$, hence (Proposition 4.1) there exists exactly one computation from $C|S$ as below.

$$C|S \rightarrow\equiv_{ren} [\![I_1]\!]|\cdots|[\![I_n]\!]|\surd|S \xrightarrow{\surd} .$$

Clearly, this computation guarantees the compliance of C and S.

Suppose now $n > 0$ and $(i, c_1, \cdots, c_m) \rightarrow_R (i', c'_1, \cdots, c'_m) \triangleq R'$. Theorem 4.2 and Proposition 4.1 guarantee that there exists $l > 0$ such that $C|S \longrightarrow D_1 \longrightarrow \cdots \longrightarrow D_l$ and $D_l \equiv_{ren} \overline{inst_{i'}}|[\![I_1]\!]|\cdots|[\![I_n]\!]|inst_{n+1}.\surd|\{\!|r_1, c'_1|\!\}|\cdots|\{\!|r_m, c'_m|\!\}$ and that $D_k \overset{\surd}{\not\longrightarrow}$, for any $1 \leq k \leq l$. By looking at the proof of the theorem, it is also clear that any D_k cannot originate other transitions, a part from that already considered in the computation above. Therefore, D can only evolve into D_l and then, by applying the induction hypothesis to R', it follows that C and S are compliant.

(⇒): Suppose now that C and S are compliant. To prove that R terminates it is sufficient to suppose, by contradiction, that it is not the case. By Theorem 4.2, this implies that there exists an infinite (hence maximal) computation $C|S \rightarrow D_1 \rightarrow \cdots \rightarrow D_l \rightarrow \cdots$ where $D_k \overset{\surd}{\not\longrightarrow}$, for any k. This contradicts the hypothesis that C and S are compliant.

□

5 Mutual compliance

In Section 2 we have introduced a notion of compliance based on client's satisfaction: whenever the client reaches a success state the whole system is successful. Hence, the success of the system is established by ignoring what happens on the service side. This could leave the service in an inconsistent state. Let's consider for example a service demanding for an additional confirmation from the client before executing the required task. In case the client decides to abandon the session before sending this final approval the (current instance of the) service is blocked. This situation could require the usage of timeouts mechanisms, especially in presence of recursive services having at most only one active instance at a time (this could be the case e.g. when the service accesses critical data). A concrete example could be an e-banking service demanding for the executive password of the client before performing any operation, e.g. bank transfer, shares purchase,. . . , as below.

$$\begin{aligned}
C &\triangleq \mathsf{invoke}(e-bank, ok.\mathsf{recreply}(login, log\_data.C')) \\
C' &\triangleq \mathsf{invoke}(transfer, ok.\mathsf{recreply}(send\_data, tran\_data.\surd)) \\
B &\triangleq \mathsf{recreply}(e-bank, ok.\mathsf{invoke}(login, log\_data.B')) \\
B' &\triangleq \ \mathsf{recreply}(transfer, ok.\mathsf{invoke}(send\_data, tran\_data. \\
&\qquad\qquad \mathsf{invoke}(confirm, pw) + \mathsf{recreply}(abort, ok))) \\
&\quad +\mathsf{recreply}(other, ok.B'') \\
B'' &\triangleq \ldots
\end{aligned}$$

It is easy to see that when the client decides to "abandon" the request without notifying the service ($tran\_data.\surd$), a pending session rests on the service side waiting for the confirmation ($\mathsf{invoke}(confirm,pw)$) or the abort ($\mathsf{recreply}(abort,ok)$).

In order to avoid such problems, we introduce another notion of compliance, called *mutual compliance*, where the synchronization of client and service's success actions is mandatory in order to establish the success of the whole system.

In this section we modify the syntax and semantics of WSCL contracts to distinguish between clients and services' successes and introduce the notion of mutual compliance. We then prove that this new notion of compliance is still decidable, by slightly modify the reasoning of Section 3.

5.1 WSCL contracts and mutual compliance

Guarded contracts are defined as follows:

$$G ::= \mathsf{invoke}(a, \textstyle\sum_{i\in I} b_i.C_i) \mid \mathsf{recreply}(a, \textstyle\sum_{i\in I} b_i.C_i) \mid \surd_{\mathrm{C}} \mid \surd_{\mathrm{S}}$$

where $\surd_{\mathrm{C}}$ and $\surd_{\mathrm{S}}$ denote the success of the client and the service, respectively.

The *run-time* syntax of contracts extends the syntax introduced in Definition 2.1 in order to take into account the occurred synchronization of $\surd_{\mathrm{C}}$ and $\surd_{\mathrm{S}}$:

$$C ::= \cdots \mid \surd.$$

A *client contract* is a contract C containing at least one occurrence of the guarded contract $\surd_{\mathrm{C}}$ and no occurrences of $\surd_{\mathrm{S}}$ and $\surd$; while a *service contract* is a contract S containing at least one occurrence of the guarded contract $\surd_{\mathrm{S}}$ and no occurrences of $\surd_{\mathrm{C}}$ and $\surd$.

The operational semantics of contracts is extended, as expected, by adding to the rules in Definition 2.3 the following ones:

$$\surd_{\mathrm{C}} \xrightarrow{\surd_{\mathrm{C}}} \mathbf{0} \qquad \surd_{\mathrm{S}} \xrightarrow{\surd_{\mathrm{S}}} \mathbf{0} \qquad \frac{C \xrightarrow{\surd_{\mathrm{C}}} C' \quad S \xrightarrow{\surd_{\mathrm{S}}} S'}{C|S \xrightarrow{\tau} C'|S'|\surd}$$

Mutual compliance coincides with Client-Service compliance introduced in Definition 2.4 and, as before, it makes sense only in case of dyadic communications and cannot be applied in a multi-party setting.

Definition 5.1 (Mutual compliance) *A client contract C and a service contract S are* mutually compliant *if for every maximal computation $C|S \longrightarrow D_1 \longrightarrow \cdots \longrightarrow D_l \longrightarrow \cdots$ there exists k such that $D_k \xrightarrow{\surd}$.*

Notice that, with the new semantics, $D_k \xrightarrow{\surd}$ in the definition above implies a previous synchronization in the computation of $\surd_C$ and $\surd_S$.

Example 5.2 (An e-bank service) *Consider the e-bank process B introduced at the beginning of this section and anther version of the client, D, that confirms the execution of the bank transfer before exiting the session. The two processes are reported below.*

$$\begin{aligned}
D &\triangleq \mathsf{invoke}(e-bank,ok.\mathsf{recreply}(login,log\_data.D'))\\
D' &\triangleq \mathsf{invoke}(transfer,ok.\mathsf{recreply}(send\_data,tran\_data.\mathsf{recreply}(confirm,pw.\surd_C)))\\
B &\triangleq \mathsf{recreply}(e-bank,ok.\mathsf{invoke}(login,log\_data.B'))\\
B' &\triangleq \ \mathsf{recreply}(transfer, ok.\mathsf{invoke}(send\_data, tran\_data.\\
&\qquad\qquad \mathsf{invoke}(confirm,pw.\surd_S) + \mathsf{recreply}(abort,ok.\surd_S)))\\
&\quad +\mathsf{recreply}(other,ok.B'')
\end{aligned}$$

There is a sole computation from D|B, reported below, which guarantees mutual compliance of the two.

$$\begin{aligned}
D|B &\longrightarrow^{*} \mathsf{recreply}(login,log\_data.D') \,|\, \mathsf{invoke}(login,log\_data.B')\\
&\longrightarrow^{*} \mathsf{invoke}(transfer,ok.\mathsf{recreply}(send\_data,tran\_data.\mathsf{recreply}(confirm,pw.\surd_C)))\\
&\qquad |\ \mathsf{recreply}(transfer, ok.\mathsf{invoke}(send\_data, tran\_data.\\
&\qquad\qquad \mathsf{invoke}(confirm,pw.\surd_S) + \mathsf{recreply}(abort,ok.\surd_S)))\\
&\qquad + \mathsf{recreply}(other,ok.B'')\\
&\longrightarrow^{*} \mathsf{recreply}(send\_data,tran\_data.\mathsf{recreply}(confirm,pw.\surd_C))\\
&\qquad | \,\mathsf{invoke}(send\_data,tran\_data.\mathsf{invoke}(confirm,pw.\surd_S) + \mathsf{recreply}(abort,ok.\surd_S))\\
&\longrightarrow^{*} \mathsf{recreply}(confirm,pw.\surd_C) \,|\, \mathsf{invoke}(confirm,pw.\surd_S) + \mathsf{recreply}(abort,ok.\surd_S)\\
&\longrightarrow^{*} \surd_C \,|\, \surd_S\\
&\longrightarrow \surd \xrightarrow{\surd} .
\end{aligned}$$

5.2 Decidability of mutual compliance

As before, decidability is obtained by translating WSCL contracts into Petri nets. The definition of the Petri net associated to a contract is essentially the same as in Section 3, except for the presence of new places for $\surd_C$ and $\surd_S$ and of a new transition allowing the synchronization of the two and leading to the success state labeled by $\surd$, as below.

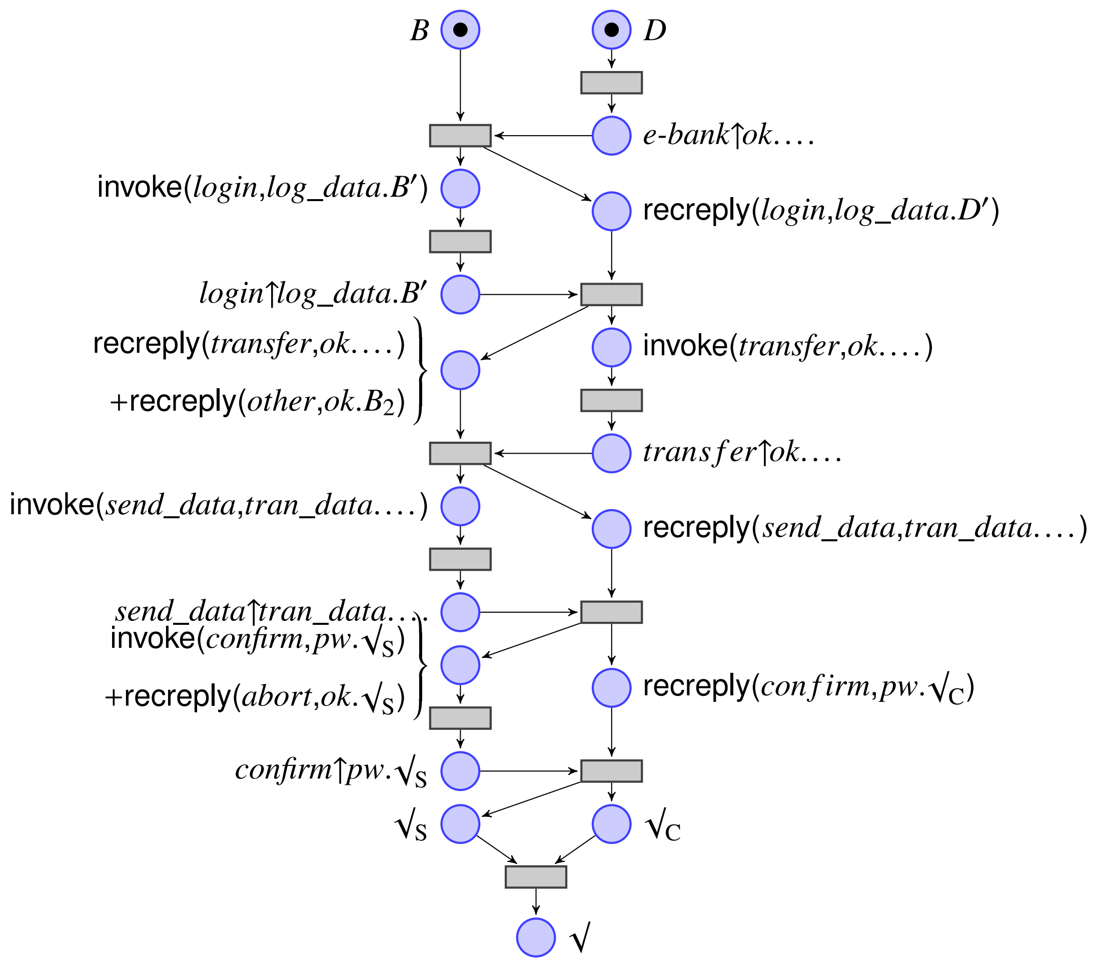

B
D
$e\text{-}bank{\uparrow}ok....$
invoke($login,log\_data.B'$)
recreply($login,log\_data.D'$)
$login{\uparrow}log\_data.B'$
recreply($transfer,ok....$)
+recreply($other,ok.B_2$)
invoke($transfer,ok....$)
$transfer{\uparrow}ok....$
invoke($send\_data,tran\_data....$)
recreply($send\_data,tran\_data....$)
$send\_data{\uparrow}tran\_data....$
invoke($confirm,pw.\surd_S$)
+recreply($abort,ok.\surd_S$)
recreply($confirm,pw.\surd_C$)
$confirm{\uparrow}pw.\surd_S$
$\surd_S$
$\surd_C$
$\surd$

Figure 1. $Net^{\mathrm{m}}(B|D)$.

$$\{\sum_{i\in I} G_i, \sum_{j\in J} G_j\} \Rightarrow \{\surd\} \quad \text{if } G_k = \surd_C \text{ and } G_l = \surd_S \text{ for some } k \in I \text{ and } l \in J$$

Example 5.3 *Consider the e-bank service B and the client D from Example 5.2. The marked net $Net^{\mathrm{m}}(B|D)$ is depicted in Figure 1.*

In the remaining part of the section we prove that soundness and completeness of the translation still hold and that mutual compliance is preserved by the translation.

The relationship between markings and contracts introduced in Lemma 3.7 needs to be modified by adding a fourth item as below:

Lemma 5.4 (Extension of Lemma 3.7) *Let C be a* WSCL *contract system and suppose $dec(C) = m$. The following holds:*

(1) if $m = \{\sum_{i\in I} G_i\} \oplus m'$ then $C \equiv \sum_{i\in I} G_i \mid D$, for some D such that $dec(D) = m'$;
(2) if $m = \{a{\uparrow}\sum_{j\in J} b_j.C_j\} \oplus m'$ then $C \equiv \sum_{j\in J} r\langle b_j\rangle.C_j \mid \overline{a}\langle r\rangle \mid D$, for some D and r such that $r \notin Names(D)$ and $dec(D) = m'$;

(3) if $m = \{c \downarrow \sum_{j \in J} b_j.C_j\} \oplus m'$ then $c \neq b_j$ for each $j \in J$ and $C \equiv \sum_{j \in J} r\langle b_j \rangle.C_j \mid \overline{r}\langle c \rangle \mid D$, for some D and r such that $r \notin Names(D)$ and $dec(D) = m'$;

(4) if $m = \{\surd\} \oplus m'$ then $C \equiv \surd \mid D$, for some D such that $dec(D) = m'$.

The remaining propositions and theorems are still valid; little changes in the proofs of Proposition 3.8 and Theorem 3.9 are needed. In case of Proposition 3.8, it is sufficient to extend the proof of ($\Rightarrow$) by adding another item guaranteeing the absence of synchronization of $\surd_C$ and $\surd_S$. In case of Theorem 3.9, it is necessary to extend the proof by considering the new kind of net transition. In both cases, the changes are minimal and easy to adjust, and the whole proofs are omitted.

The following version of Corollary 3.13 carries over.

Corollary 5.5 (Mutual Compliance preservation) *Let C and S be respectively a WSCL client and service contract, as defined in Subsection 5.1. We have that C and S are mutually compliant if and only if in the marked Petri net $Net^{m}(C|S)$ all the maximal computations traverse at least one success marking.*

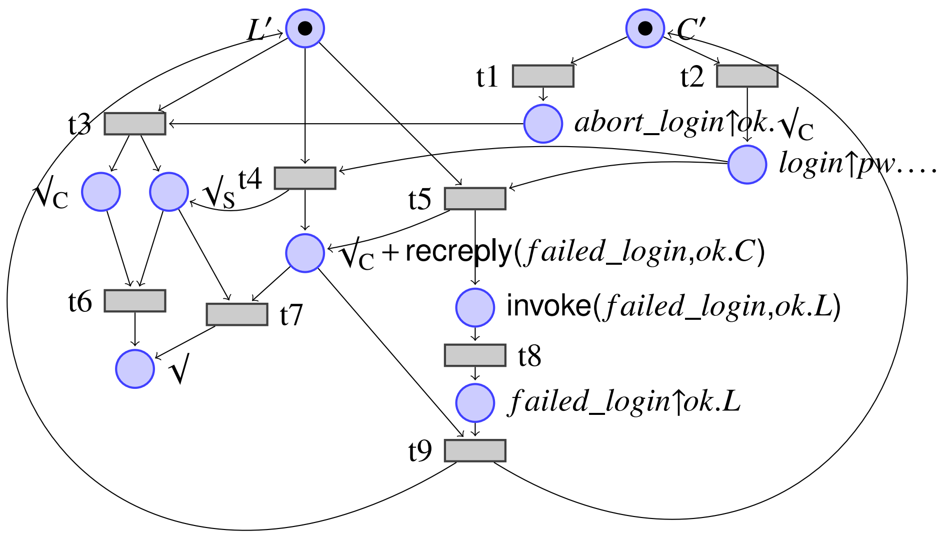
This result is essentially the same of that reported in Corollary 3.13, therefore the reasoning introduced in Subsection 3.2 applies to the new notion of compliance: the algorithm introduced in Table 4 together with Theorem 3.16 guarantee the decidability of mutual compliance.

Example 5.6 (A login service and its client) *Consider a login service, L, that waits for a login request or for an abort message. In case of login, it receives the credentials of the requester and either enters a successful state or notifies the failure and restart its execution from the beginning. In case of abort, it ends in a successful state. Consider a client, C, that is the complementary of L.*

$$\begin{aligned} L &\triangleq recX.\Big(\mathsf{recreply}(login, x_{pw}.\surd_S + x_{pw}.\mathsf{invoke}(failed\_login, ok.X)) \\ &\qquad + \mathsf{recreply}(abort\_login, ok.\surd_S)\Big) \\ C &\triangleq recY.\Big(\mathsf{invoke}(login, pw.(\surd_C + \mathsf{recreply}(failed\_login, ok.Y))) \\ &\qquad + \mathsf{invoke}(abort\_login, ok.\surd_C)\Big) \end{aligned}$$

Suppose L' and C' correspond to the one step unfolding of L and C, the marked net $Net^{m}(L'|C')$ is depicted in Figure 2.

It is easy to see that there exists an infinite computation, involving transitions t2, t5, t8 and t9, which does not traverse the success state $\surd$, therefore the two contracts are not mutually compliant. Notice that by considering the notion of compliance introduced in Definition 2.4, the two contracts are compliant: the infinite computation above would traverse infinitely often a successful state (each occurrence of $\surd_C$ should be replaced by $\surd$ in Figure 2, therefore the state labeled by



L'
C'
t1
t2
t3
$abort\_login\uparrow ok.\surd_C$
$login\uparrow pw....$
t4
t5
$\surd_C$
$\surd_S$
$\surd_C + \mathsf{recreply}(failed\_login,ok.C)$
t6
t7
$\mathsf{invoke}(failed\_login,ok.L)$
t8
$\surd$
$failed\_login\uparrow ok.L$
t9

Figure 2. $Net^{\mathrm{m}}(L'|C')$.

$\surd_C + \mathsf{recreply}(failed\_login,ok.C)$ *should be considered successful).*

6 Related Work and Conclusion

We have presented two models of contracts with bidirectional request-response interaction, studied a notion of compliance based on must testing and established an expressiveness gap between the two models showing that compliance is decidable in the first one while it is undecidable in the second one.

This paper is in the line of recent research dedicated to the formal analysis of service behavioural contracts exploiting process calculi. To the best of our knowledge, though, only one-way operations have been considered so far. An initial theory of contracts for client-service interaction has been proposed by Carpineti et al. [11] and then independently extended along different directions by Bravetti and Zavattaro (see e.g. [7,9]) by Laneve and Padovani [20], and by Castagna et al. [12]. The main objective of those papers was to define a *subcontract* relation suitable to check the replaceability of one service with another one without affecting the correctness of a modeled system. The approach in [11] considers a notion of system correctness similar to the one used in this paper and inspired by must-testing. A corresponding *subcontract* relation, enhancing the must-testing preorder, is defined in [20]. By making use of explicit *interfaces* indicating the operations used by one service to interact with the external environment, both *in-width* and *in-depth* refinements are admitted: a subcontract can have additional behaviour, available either as new choices in branches or as longer continuations, but only if this additional behaviour is activated by actions on operations that are not in the interface. A notion of correctness in which all the involved partners should eventually reach successful completion (similar to the mutual compliance considered in this paper) has been presented in [7], where a corresponding subcontract relation is also introduced. The global completion approach is particularly appropriate for systems where there is no clear distinction between clients and services as in the so called *service choreographies*. The relationship between contract theories and choreography languages (such as WS-CDL [27]) has been investigated in [9]. In all the above theories, the defined subcontract relation is influenced by the operations that a service can use to interact with the external environment, as invocations on these operations could activate the additional behaviour available in refinements. A different approach is taken in [12], where dynamic filters are automatically synthesized in order to guarantee that such an additional behaviour cannot be wrongly activated.

As for future work, we plan to investigate the (un)decidability of other definitions of compliance present in the literature. In fact, the must-testing approach —the one that we consider in this paper— has been adopted in early works about service compliance (see e.g. [11]). More recent papers consider more sophisticated notions. For instance, the should-testing approach [26] adopted, e.g., in [10] admits also infinite computations if in every reached state there is always at least one path leading to a success state.

Moreover, it would be interesting to apply the techniques presented in this paper to more sophisticated orchestration languages, like the recently proposed calculi based on the notion of *session* [6,5]. For instance, in [2], a type system is presented ensuring a client progress property – basically, absence of deadlock – in a calculus where interaction between (instances of) the client and the service is tightly controlled via session channels. It would be interesting to check to what extent the decidability techniques presented here apply to this notion of progress. Also connections with *behavioural types* [18,1] deserve attention. In the setting of process calculi, these types are meant to provide behavioural abstractions that are in general more tractable than the original process. In the present paper, the translation function of WSCL contracts into Petri nets can be seen too as a form of behavioural abstraction. In the case of tightly controlled interactions (sessions) [2], BPP processes, a proper subset of Petri nets featuring no synchronization [15], have been seen to be sufficient as abstractions. For general pi-processes, full CCS with restriction is in general needed. One would like to undertake a systematic study of how communication capabilities in the original language (unconstrained interaction vs. sessions vs. request-response vs....) trades off with tractability of the behavioural abstractions (CCS vs. BPP vs. Petri nets vs. ...).

References

[1] Acciai, L., Boreale, M.: Spatial and behavioural Types in the pi-calculus. In Proc. of CONCUR'08, LNCS 5201:372–386 (2008). Full version in Information and Computation 208:1118–1153 (2010)

[2] Acciai, L., Boreale, M.: A Type System for Client Progress in a Service-Oriented Calculus. In Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. LNCS 5065:642–658 (2008)

[3] Acciai, L., Boreale, M., Zavattaro, G.: Behavioural Contracts with Request-Response Operations. In Proc. of COORD'10, LNCS 6116:16–30 (2010)

[4] Boreale, M., Bravetti, M.: Advanced mechanisms for service composition, query and discovery. LNCS (2010). To appear.

[5] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. In Proc. of FMOODS'08, LNCS 5051:19–38 (2008)

[6] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: A Service Centered Calculus. In Proc. of WS-FM'06, LNCS 4184:38–57 (2006)

[7] Bravetti, M., Zavattaro, G.: Contract based Multi-party Service Composition, In Proc. of FSEN'07, LNCS 4767207–222 (2007)

[8] Bravetti, M., Zavattaro, G.: A Theory for Strong Service Compliance, In Proc. of Coordination'07, LNCS 4467:96–112 (2007)

[9] Bravetti, M., Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance, In Proc. of SC'07, LNCS 4829:34–50 (2007)

[10] Bravetti, M. and Zavattaro, G.: Contract-Based Discovery and Composition of Web Services. In Proc. of SFM'09. LNCS 5569: 261–295 (2009)

[11] Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A Formal Account of Contracts for Web Services, In Proc. of WS-FM'06, LNCS 4184:148–162 (2006)

[12] Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services, In Proc. of POPL'08, ACM Press 261–272 (2008)

[13] Castagna, G. and Padovani, L.: Contracts for Mobile Processes, In Proc. of Concur'09, LNCS 5710:211–228 (2009)

[14] De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci, 34:83–133 (1984)

[15] Esparza, J.: Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. Fundam. Inform. 31(1):13–25 (1997)

[16] Esparza, J., Nielsen, M.: Decidability Issues for Petri Nets - a survey. Bulletin of the EATCS 52:244–262 (1994)

[17] Finkel, A., Schnoebelen, Ph.: Well-Structured Transition Systems Everywhere! Theoretical Computer Science, 256(1-2): 63–92 (2001)

[18] Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. Theoretical Computer Science 311(1-3), 121–163 (2004)

[19] Karp , R.M., Miller, R.E.: Parallel Program Schemata. Journal of Computer and System Sciences 3:147–195 (1969)

[20] Laneve, C., Padovani, L.: The must preorder revisited - An algebraic theory for web services contracts. In Proc. of Concur'07, LNCS 4703:212–225 (2007)

[21] Milner, R.: Communication and concurrency. Prentice-Hall (1989)

[22] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation, volume 100, pages 1–40 (1992)

[23] Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs (1967)

[24] OASIS: Web Services Business Process Execution Language (WSBPEL). Standard proposal available at: `www.oasis-open.org/committees/wsbpel` (2007)

[25] Petri, C.A.: *Kommunikation mit Automaten*. Ph. D. Thesis. University of Bonn (1962)

[26] Rensink A., Vogler W.: Fair testing. Inf. Comput. 205(2), 125–198 (2007)

[27] W3C: Web Services Choreography Description Language (WSCDL). Standard proposal available at: `http://www.w3.org/TR/ws-cdl-10` (2005)

[28] W3C: Web Services Choreography Interface (WSCI). Standard proposal available at: `http://www.w3.org/TR/wsci` (2002)

[29] W3C: Web Services Conversation Language (WSCL). Standard proposal available at: `http://www.w3.org/TR/wscl10` (2002)

[30] W3C: Web Services Description Language (WSDL). Standard proposal available at: `http://www.w3.org/TR/wsdl` (2001)