A Theory of Adaptable Contract-Based Service Composition

G. Bernardi   M. Bugliesi   D. Macedonio   S. Rossi
Dipartimento di Informatica, Università Ca' Foscari, Venice, Italy
e-mail:{gbernard,bugliesi,mace,srossi}@dsi.unive.it

Abstract

Service Oriented Architectures draw heavily on techniques for reusing and assembling off-the-shelf software components. While powerful, this programming practice is not without a cost: the software architect must ensure that the off-the-shelf components interact safely and in ways that conform with the specification. We develop a new theory for adaptable service compositions. The theory provides an effective framework for analyzing the conformance of contract-based service compositions, and for enforcing their compliance, in a uniform, and formally elegant setting.

1. Introduction

Modern software design is increasingly being based on a methodology that has become known as Service Oriented Architecture (SOA). Central to SOA is the idea of *reusing* existing off-the-shelf software units (services) and assembling them to develop new applications. Two approaches have emerged as mainstream: *orchestration* and *choreography*. An orchestration combines existing services around the orchestrator, a component that acts as the coordinator of the services and mediates all of their interactions. A choreography, instead, organizes the services along a message flow which all partners must comply with autonomously, without intervention of a central coordinator.

Several languages have emerged in the recent literature as specification formalisms for orchestrations (e.g., XLANG [21], WSFL [13] and WS-BPEL [17]) and choreographies (WS-CDL [24] and BPEL4Chor [8]). Such languages provide primitives for assembling services based on abstract descriptions of the services' behavior, given as interfaces or service *contracts*.

Needless to say, a number of challenges rest behind the scene of a design paradigm based on reusing off-the-shelf components: most notably, those challenges arising from the mismatches between the behavior expected of the services to be included in a composition, and the behavior of the available services. At a very basic level, such mismatches may hinder the *compliance* [20] of the composite application, by leading it to a deadlock state, or trapping it into an infinite loop (a livelock). At a higher level, they may break the behavioral *conformance* with respect to the specification [1, 4, 5, 23].

There is a large body of work in the literature on how the mismatches may be attacked by means of techniques for component/service adaptation (cf. Section 6 for a brief survey). In this paper, we develop a new theory of adaptable compositions that draws on the theory of contracts developed in [7], and supports the analysis of compliance and conformance of contract-based service compositions in a uniform, and formally elegant setting. Our approach is based on using the filters introduced in [7] both as prescriptions of behavior (coercions to prevent service misbehavior) and as descriptors of the choreographic roles of the service components. We devise an algorithm to ensure the compliance of a composition by the automatic synthesis of an adapting filter. Further, we describe a technique, based on a simulation ordering, to verify the conformance of the adapted composition by contrasting the adapted contract against the associated *role filter*.

Plan. Sections 2 and 3 introduce our theory of service contract compositions and filters. Section 4 describes the algorithm for synthesizing the adapting filter, while Section 5 shows how the theory supports the analysis of conformance. Section 6 discusses related work.

2. A Process Algebra for Service Composition

We represent service contracts as terms of a CCS-like [16] process calculus with recursion and operators for guarded and internal choice, but no parallel composition. Parallel composition arises in service contract compositions (SCC's for short), that we define after [4, 5, 20] as the parallel (and concurrent) composition of located contracts. We presuppose a denumerable set $\mathcal{N}$ of action names, ranged over by $a, b, c, \ldots$, and a denumerable set *Loc* of location names, ranged over by l, m, n. The actions represent the basic units of observable behavior of the underlying services, while the location names specify the peers providing the services together with the ports on which the services are made available. The syntax is formally defined below.

Contract actions

$$\begin{array}{lll} p & ::= & \bar{a}@l \mid a \qquad \text{send} \mid \text{receive a message} \end{array}$$

Contracts

$$\begin{array}{llll} \sigma & ::= & \mathbf{1} \mid X & \text{termination} \mid \text{variable} \\ & \mid & \sum_{i=1}^{n} p_i.\sigma_i & \text{external choice} \\ & \mid & \sigma \oplus \sigma & \text{internal choice} \\ & \mid & \mathtt{rec}(X)\,\sigma & \text{recursion} \end{array}$$

Compositions

$$\begin{array}{llll} C & ::= & [\sigma]_l & \text{located contract} \\ & \mid & C \parallel C & \text{composition} \end{array}$$

The contract $a.\sigma$ describes a service that waits for a message on a and then continues as σ; dually, $\bar{a}@l.\sigma$ first sends a message on a to the service located at l and then behaves as σ. $\sum_{i=1}^{n} p_i.\sigma_i$ is a guarded, external choice: depending on actions exposed by its peers, a service with this contract may choose any of the p_i and then continue as σ_i. We assume the following conditions on external choice: $n \geq 0$ and $p_i \neq p_j$ whenever $i \neq j$, and note $\mathbf{0}$ a sum with $n = 0$. Internal choices may be unguarded: the term $\sigma \oplus \sigma'$ represents a local choice between σ and σ' made irrespective of the structure of the interacting environment. The term $\mathtt{rec}(X)\,\sigma$ defines a possibly recursive contract: the recursion variable X may only occur guarded by a prefix in σ. Finally, $\mathbf{1}$ signals successful termination: services that get stuck before reaching termination or end-up into an empty sum $\mathbf{0}$ are deadlocked.

A composition must be well-formed [4, 5] to constitute a legal SCC, namely: (i) every component $[\sigma]_l$ must occur at a different location l of the composition, and (ii) the target of each output action at a given location must be different from the the location itself (i.e. the action $\bar{a}@l$ cannot occur inside a component $[\sigma]_l$). If $C = [\sigma_1]_{l_1} \parallel \cdots \parallel [\sigma_n]_{l_n}$ is a legal SCC, we say that C is an $\{l_1, \ldots, l_n\}$-SCC (dually, that $\{l_1, \ldots, l_n\}$ are the underlying locations for C).

Throughout, we assume that contracts are closed and that SCC's are well formed, unless otherwise stated. Also, we often omit trailing $\mathbf{1}$'s.

Dynamics of contracts and SCCs. We define the dynamics of the calculus in terms of labelled transition semantics (and a success predicate), with rules reported in Table 1. The first block of rules defines the transitions for contracts, which are mostly self-explanatory. We remark that the syntactic restriction on the external choice operator implies that for any given contract σ and a action p, there is always at most one σ' such that $\sigma \xrightarrow{p} \sigma'$. This is coherent with the intended use of contracts as behavioral interfaces. To illustrate, consider a service offering first action p and then behaving as either $q.\mathbf{1}$ or $r.\mathbf{1}$. Depending on whether the second choice is internal or external, the observable behavior of this service may be specified as $p.(q.\mathbf{1} \oplus r.\mathbf{1})$ or $p.(q.\mathbf{1} + r.\mathbf{1})$, respectively. If the choice between q and r is internal, the behavior could alternatively be specified as $p.q.\mathbf{1} + p.r.\mathbf{1}$ but this is not a legal term of our contract language.

Each contract transition yields a corresponding transition for the location hosting the contract. The transitions for SCC's are relative to the underlying set of locations. The transitions are mostly standard. Perhaps interestingly, the input transition is presented in an *early* style, by anticipating the source m of the signal on which the input action at n is going to synchronize. Indeed, the rule could be stated equivalently in a *late* style, using simpler input actions of the form a_n. On the other hand, the chosen style of presentation gives finer control on the synchronizations an input action may have in a given SCC. This, in turn, provides us with further control on how to shape an SCC by filtering.

Throughout the paper we omit leave the subscript L and write $\xrightarrow{\alpha}$ in place of $\xrightarrow{\alpha}_L$ whenever the underlying set of locations L is understood from the context. Also, we write $\Longrightarrow_L$ to note the reflexive and transitive closure of $\xrightarrow{\tau}_L$.

We conclude the presentation of the calculus by introducing the notion of compliant composition of contracts. This notion is the same as in [3, 4]. Intuitively, a composition of services is compliant if it is deadlock and livelock free, i.e. it does not get stuck nor does it get trapped into infinite loops with no exit states. Formally:

Definition 2.1 [Compliance] A composition C is compliant, noted $C\!\downarrow$, if for every C' such that $C \Longrightarrow C'$ there exists C'' such that $C' \Longrightarrow C''$ and $C'' \checkmark$.

3. Filters and Component Adaptation

A filter is the specification of the legal flow of actions for an individual contract. We define filters after [7], and extend that definition to allow recursive filters. The syntax is defined as follows:

$$f \in \mathcal{F} \ := \ \mathbf{0} \mid \alpha.f \mid f \times f \mid f \otimes f \mid X \mid \mathtt{rec}(X)\,f$$

Filters have a simple semantics, defined in Table 2. The transition relation is readily understood if we view a filter as a deterministic finite-state automaton accepting possibly infinite strings over the alphabet of SCC actions. If we take this view, $f_1 \times f_2$ and $f_1 \otimes f_2$ are directly realized as the union and the intersection of the automata f_1 and f_2.

Table 1 Dynamics of Service Contracts and SCC's

Contract transitions: $\sigma \xrightarrow{\lambda} \sigma'$ with λ ranging over the following actions $\lambda ::= p \mid \tau$.

$$\mathbf{1}\checkmark \qquad \textstyle\sum p_i.\sigma_i \xrightarrow{p_i} \sigma_i \qquad \sigma_1 \oplus \sigma_2 \xrightarrow{\tau} \sigma_i \quad (i = 1, 2)$$

$$\frac{\sigma \left\{ {}^{\mathtt{rec}(X)\,\sigma}/_{X} \right\} \checkmark}{\mathtt{rec}(X)\,\sigma\,\checkmark} \qquad \frac{\sigma \left\{ {}^{\mathtt{rec}(X)\,\sigma}/_{X} \right\} \xrightarrow{\lambda} \sigma'}{\mathtt{rec}(X)\,\sigma \xrightarrow{\lambda} \sigma'}$$

SCC transitions: $C \xrightarrow{\alpha}_L C'$ with α ranging over the actions $a_{n\to m} \mid \bar{a}_{n\to m}$ and $L \subset Loc$ is a finite set of locations.

$$\frac{\sigma\,\checkmark}{[\sigma]_n\,\checkmark} \qquad \frac{\sigma \xrightarrow{\tau} \sigma'}{[\sigma]_n \xrightarrow{\tau}_L [\sigma']_n} \qquad \frac{\sigma \xrightarrow{a} \sigma'}{[\sigma]_n \xrightarrow{a_{m\to n}}_L [\sigma']_n}\,(m \in L) \qquad \frac{\sigma \xrightarrow{\bar{a}@m} \sigma'}{[\sigma]_n \xrightarrow{\bar{a}_{n\to m}}_L [\sigma']_n}\,(m \neq n)$$

$$\frac{C_1\,\checkmark \quad C_2\,\checkmark}{C_1 \parallel C_2\,\checkmark} \qquad \frac{C_1 \xrightarrow{a_{n\to m}} C_1' \quad C_2 \xrightarrow{\bar{a}_{n\to m}} C_2'}{C_1 \parallel C_2 \xrightarrow{\tau}_L C_1' \parallel C_2'} \qquad \frac{C_1 \xrightarrow{\alpha} C_1'}{C_1 \parallel C_2 \xrightarrow{\alpha}_L C_1' \parallel C_2} \qquad \frac{C_1 \xrightarrow{\tau} C_1'}{C_1 \parallel C_2 \xrightarrow{\tau}_L C_1' \parallel C_2}$$

Definition 3.1 [FILTER PRE-ORDER] The filter pre-order $f \leq g$ is the largest relation such that if $f \overset{\alpha}{\longmapsto} f_\alpha$ then $g \overset{\alpha}{\longmapsto} g_\alpha$ and $f_\alpha \leq g_\alpha$.

We note $(\mathcal{F}, \sqsubseteq)$ the partial order induced by $\leq$: as usual we abuse the notation and identify a filter f with its equivalence class $[f]_\sim$, where $\sim$ is the symmetric closure of $\leq$. The union and intersection filter constructors represent the glb and lub operators for $(\mathcal{F}, \sqsubseteq)$. Furthermore, if we assume a finite alphabet A of actions, the set of filters $\mathcal{F}_A$ insisting on A forms a complete lattice with $\mathbf{0}$ and the identity filter: $I_A \overset{\mathit{def}}{=} \mathtt{rec}(X) \prod_{\alpha \in A} \alpha.X$ as bottom and top elements, respectively.

Filters may be employed to block any contract transitions that hinder the mutual compliance of the peers interacting within an SCC: specifically, the application $f \rhd [\sigma]_\ell$ blocks any action from $[\sigma]_\ell$ that is not explicitly enabled by f (cf. the second block of rules in Table 2). For uniformity, we may then express the behavior of an unfiltered location in terms of a location filtered by the identity (on a suitable alphabet).

Filters may be composed to help shape an SCC. Given a set of locations L, a composite L-filter F is finite map from the locations in L to filters: $\{\ell \to f_\ell \mid \ell \in L\}$. An L-filter may be applied to an L-SCC:

$$F \rhd_L C ::= F[l_1] \rhd [\sigma_1]_{l_1} \parallel \cdots \parallel F[l_n] \rhd [\sigma_n]_{l_1}$$

When we write $F \rhd_L C$ we tacitly assume that the underlying set of locations for both F and C is L. The operators of union and intersection, as well as the ordering on filters extend directly to composite filters, as expected. Namely, for F and G L-filters, and $\bullet \in \{\leq, \times, \otimes\}$, we define:

$$(F \bullet_L G)[\ell] \quad \overset{\mathit{def}}{=} \quad F[\ell] \bullet G[\ell] \quad \text{for all } \ell \in L$$

To ease the notation, we omit the subscript L when understood from the context. We may then generalize the syntax of SCCs to account for the application of filters on the locations underlying the SCC.

Filtered SCCs

$$\begin{array}{lll} C & ::= & \ldots & \text{as in section 2} \\ & \mid & F \rhd_L C & \text{composite } L\text{-filter application} \end{array}$$

The dynamics of filtered SCCs derives directly by combining the structural transitions in the last block of rules in Table 1 with the ones for filtered locations in Table 2. Notice, in particular, that when writing $F \rhd C$ we may safely assume that C is not filtered, as nested filter applications like $F_1 \rhd F_2 \rhd C$ may equivalently be represented as the application of the intersection filter $(F_1 \otimes F_2) \rhd C$.

Relevance and weak compliance Being behavioral transformers for the components of an SCC, filters also help recover a notion of compliance for SCCs that are not compliant according to our current definition. Intuitively a composition C is weakly compliant if it *can be made* compliant by filtering away all actions that may bring it to a deadlock or a livelock. Say that a composite filter *fixes* C if $F \rhd C$ is compliant.

Definition 3.2 [Weak Compliance] A composition C is *weakly compliant*, written $C\Downarrow$, if there exists a composite filter F that fixes C.

Table 2 Dynamics of Filtered SCC's

Transitions for filters

$$\alpha.f \xmapsto{\alpha} f \qquad \frac{f\left\{{}^{\texttt{rec}(X)\,f}/_{X}\right\} \xmapsto{\alpha} f'}{\texttt{rec}(X)\,f \xmapsto{\alpha} f'} \qquad \frac{f \xmapsto{\alpha} f_\alpha \qquad g \xmapsto{\alpha} g_\alpha}{f \otimes g \xmapsto{\alpha} f_\alpha \otimes g_\alpha}$$

$$\frac{f \xmapsto{\alpha} f_\alpha \qquad g \xmapsto{\alpha} g_\alpha}{f \times g \xmapsto{\alpha} f_\alpha \times g_\alpha} \qquad \frac{f \xmapsto{\alpha} f_\alpha \qquad g \not\xmapsto{\alpha}}{f \times g \xmapsto{\alpha} f_\alpha} \qquad \frac{f \not\xmapsto{\alpha} \qquad g \xmapsto{\alpha} g_\alpha}{f \times g \xmapsto{\alpha} g_\alpha}$$

Transitions for filtered locations

$$\frac{[\sigma]_n \xrightarrow{\alpha} [\sigma']_n \qquad f \xmapsto{\alpha} f'}{f \triangleright [\sigma]_n \xrightarrow{\alpha} f' \triangleright [\sigma']_n} \qquad \frac{[\sigma]_n \xrightarrow{\tau} [\sigma']_n}{f \triangleright [\sigma]_n \xrightarrow{\tau} f \triangleright [\sigma']_n} \qquad \frac{[\sigma]_n \checkmark}{f \triangleright [\sigma]_n \checkmark}$$

In the next section we present an algorithm that given a composition C infers a composite filter F that fixes C, whenever such F exists. The algorithm is so structured as to guarantee two important properties on the inferred filter. On the one hand, the filter is as permissive as possible, in that it is the greatest (with respect to the pre-order $\leq$) among the filters that fix C. On the other side, the inferred filter is *relevant*, i.e., minimal in size: for any computation state reached by the SCC via a series of τ transitions (local moves or synchronizations), the filter only enables actions that may be attempted at that state (either directly, or via a local choice), by one of the components of the SCC.

While the notion of relevance is intuitive, the presence of the local moves makes its formal definition somewhat involved. We first introduce some notation to help (i) to distinguish a local move from a synchronization, and (ii) to identify the contribution of every location in a synchronization.

- $C \xhookrightarrow{\tau} C'$ iff $C \xrightarrow{\tau} C'$ because $C \equiv [\sigma]_\ell \parallel C''$, $C' \equiv [\sigma']_\ell \parallel C''$ and $[\sigma]_\ell \xrightarrow{\tau} [\sigma']_\ell$.

- $C \xhookrightarrow{\{a_{n\to m}\}} C'$ iff $C \xrightarrow{\tau} C'$ because $C \equiv C_1 \parallel C_2$, $C_1 \xrightarrow{a_{n\to m}} C_1'$, $C_2 \xrightarrow{\bar{a}_{n\to m}} C_2'$, and $C' \equiv C_1' \parallel C_2'$.

We let φ range over the labels $\{a_{n\to m}\}$ and τ. Then, we define a weak version of the $\overset{\varphi}{\lhook\joinrel\twoheadrightarrow}$ moves that are synchronizations. Simply, define $\overset{\tau}{\lhook\joinrel\twoheadrightarrow} \overset{\mathit{def}}{=} \xhookrightarrow{\tau} \cdots \xhookrightarrow{\tau}$ and then $\overset{\{a_{n\to m}\}}{\lhook\joinrel\twoheadrightarrow} \overset{\mathit{def}}{=} \overset{\tau}{\lhook\joinrel\twoheadrightarrow} \cdot \xhookrightarrow{\{a_{n\to m}\}} \cdot \overset{\tau}{\lhook\joinrel\twoheadrightarrow}$.

Definition 3.3 [RELEVANCE] Let $\mathcal{C}$ be a non-empty set of L-SCCs. A filter f is ℓ-relevant in $\mathcal{C}$, written $f \propto_\ell \mathcal{C}$, if whenever $f \xmapsto{\alpha} \widehat{f}$ one has $\alpha \in \{a_{\_\to\ell}, \bar{a}_{\ell\to\_}\}$ and there exists $\widehat{C} \in \mathcal{C}$ such that[1] $\widehat{C} \overset{\{\alpha\}}{\lhook\joinrel\twoheadrightarrow}$ and $\widehat{f} \propto_\ell \{C' \mid \widehat{C} \overset{\{\alpha\}}{\lhook\joinrel\twoheadrightarrow} C'\}$.

A composite L-filter F is relevant for $\mathcal{C}$, written $F \propto \mathcal{C}$ iff $F[\ell] \propto_\ell \mathcal{C}$ for all $\ell \in L$. Finally, a composite L-filter is relevant for an L-SCC C if $F \propto \{C\}$.

We move on describing the algorith that, given an SCC, synthesizes the $\sqsubseteq$-greatest relevant filter that fixes the SCC.

4. Synthesis of the Maximal Relevant Filter

Given a composition C, the algorithm keeps track of the reachable states of the computation in C so as to filter out the actions that may bring C to a deadlock or livelock. Since C is finite state, the reachable states may be organized into a finite state reduction graph.

A reduction graph is a directed graph $G = (V, E)$ with labeled edges and vertices. The vertices in V represent the reachable states of the underlying composition C. With each $\mathbf{v} \in V$ we associate two fields:

- $\mathit{state}[\mathbf{v}]$ gives the computation state (i.e., the derivative C' of the initial SCC C) associated with $\mathbf{v}$;
- $\mathit{result}[\mathbf{v}]$ is a tag that informs on the possible outcomes of the computation starting off at this state: SUCC, FAIL or UNDEF.

The edges in E represent the reduction steps determining the reachable states: each edge is a triple $(\mathbf{u}, \mathbf{v})_\varphi$ representing the transition $\mathit{state}[\mathbf{u}] \xhookrightarrow{\varphi} \mathit{state}[\mathbf{v}]$. Remarkably, the reduction graph only traces the states reached by means of synchronizations or internal moves, thus disregarding all states reached by labeled transitions.

Reduction graphs are stored in adjacency list representation, so that the set of outgoing edges for each $\mathbf{u} \in V$ can be retrieved as $\mathit{Adj}[\mathbf{u}]$: thus $(\mathbf{u}, \mathbf{v})_\varphi \in E$ iff $(\varphi, \mathbf{v}) \in \mathit{Adj}[\mathbf{u}]$. Also, we write $\mathit{Adj}[\mathbf{u}, \varphi]$ for the set $\{\mathbf{v} \in V \mid (\mathbf{u}, \mathbf{v})_\varphi \in E\}$ of vertices reachable from $\mathbf{u}$ with an edge labeled φ. Vertices with no outgoing edges are called leaves. The graph

[1] This notation is loose: when $\alpha = \bar{a}_{\ell\to\_}$, $\{\alpha\}$ is, in fact, $\{a_{\ell\to\_}\}$

representation includes one additional field, noted $root[G]$, that stores a reference to the node representing the initial state C of the underlying system.

The algorithm involves several steps. The first step builds the reduction graph for the given SCC. The second step propagates the *result* labels to verify whether the SCC admits a filter, i.e., whether its reduction graph contains at least one successful path from the root to a final state. The third step extracts the sub-graph of the successful paths in the reduction graph. The algorithm fails at this stage if the extracted sub-graph is empty. Otherwise the algorithm succeeds by synthesizing the filter from the success sub-graph.

Building the reduction graph. The reduction graph is built in a top-down manner from a given initial SCC term C: the details are defined by the function `BuildGraph`(C) given below. The construction grows the sets of vertices V and of edges E by iteratively exploring the set of states reached from C. At each iteration, the outer **while** loop selects one of the states reached (and not yet visited) to analyze the set of (silent or synchronization) transitions from that state. For each such transition the analysis determines whether the transition leads to a new state (in which case the set V is extended with a new vertex) or to an existing state. In either case, the set E of edges is extended with an edge corresponding to the transition. Once this analysis is concluded, the vertex is included in the (sub)set W of visited vertices. The loop terminates when all the vertices reached have been visited.

Function `BuildGraph`(C)

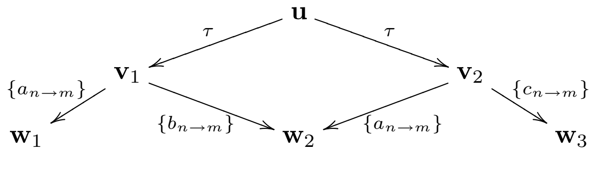
```
Input: A choreography $C$
Output: $G = (V, E)$ the reduction graph of $C$
$\mathbf{r} := \texttt{newVertex}(C)$;
$V := \{\mathbf{r}\}$; $E := \emptyset$; $W := \emptyset$;
while ($V \setminus W \neq \emptyset$) do
    $\mathbf{u} := \texttt{select}(V \setminus W)$;
    if $state[\mathbf{u}]$ ✓ then
        $result[\mathbf{u}] := \textsc{succ}$;
    else if $state[\mathbf{u}] \overset{\varphi}{\not\hookrightarrow}$ then
        $result[\mathbf{u}] := \textsc{fail}$;
    end
    foreach $\widehat{C} : state[\mathbf{u}] \xhookrightarrow{\varphi} \widehat{C}$ do
        $\mathbf{v} := \texttt{select}(\{\mathbf{w} \in V \mid state[\mathbf{w}] = \widehat{C})$;
        if $\mathbf{v}$ = NULL then
            $\mathbf{v} := \texttt{newVertex}(\widehat{C})$;
            $V := V \cup \{\mathbf{v}\}$;
        end
        $E := E \cup \{(\mathbf{u}, \mathbf{v})_\varphi\}$;
    end
    $W := W \cup \{\mathbf{u}\}$;
end
$G := (V, E)$; $root[G] := \mathbf{r}$;
return $G$;
```

The new vertices to be included in the reduction graph are created by the function `NewVertex`(C): given a state C, `NewVertex`(C) creates a new vertex with UNDEF result label and state C. Notice that, as a result of the construction, two vertices in G may be connected by more than one edge, provided that the two edges have different labels. Also note that each computation state is associated with just one vertex in the graph.

Lemma 4.1 If G is a graph generated by the procedure `BuildGraph(C)`, then all the internal vertices of G are labeled as UNDEF. Only the leaf with state $[\mathbf{1}]_{\ell_1} \parallel \ldots \parallel [\mathbf{1}]_{\ell_k}$, if it exists, is labeled as SUCC. All the other leafs are labeled as FAIL.

Let *succ*$[G]$ note the leaf labeled SUCC in G.

Example 4.2 The following graph G is generated by the configuration $P_1 \parallel P_2$, with $P_1 = [(a.d.\mathbf{1} + b.\mathbf{1}) \oplus (a.\mathbf{1} + c.e.\mathbf{1})]_m$ and $P_2 = [\bar{a}@m.\mathbf{1} + \bar{b}@m.\mathbf{1} + \bar{c}@m.\mathbf{1}]_n$.



In particular, $result[\mathbf{w}_1] = result[\mathbf{w}_2] = \textsc{fail}$, $result[\mathbf{u}] = result[\mathbf{v}_1] = result[\mathbf{v}_2] = \textsc{undef}$, and $result[\mathbf{w}_2] = \textsc{succ}$. The node *succ*$[G]$ corresponds to $\mathbf{w}_2$.

Labelling the reduction graph. In order to describe this phase of the algorithm, we introduce some auxiliary definitions. First we define $locs(\varphi)$ to be $\{m, n\}$ in case $\varphi = \{a_{m \to n}\}$, and to be $\emptyset$ in case $\varphi = \tau$. Then, let $G = (V, E)$ be a reduction graph, and $\varphi = \{a_{m \to n}\}$.

- A path $\pi = (\mathbf{u}, \mathbf{u}_1)_{\varphi_1}, \ldots, (\mathbf{u}_{n-1}, \mathbf{v})_{\varphi_n}$ from $\mathbf{u}$ to $\mathbf{v}$ in G is *φ-free* if $locs(\varphi) \cap locs(\varphi_i) = \emptyset$ for all i's.
- A vertex $\mathbf{v}$ is a *φ-free descendant* of $\mathbf{u}$ in G (dually $\mathbf{u}$ is a *φ-free ancestor* of $\mathbf{v}$) if there is a φ-free path from $\mathbf{u}$ to $\mathbf{v}$.
- A vertex $\mathbf{u}$ *yields a conflict on* φ if $\mathbf{u}$ has two distinct φ-free descendants $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $(\mathbf{v}_1, \mathbf{w}_1)_\varphi$ and $(\mathbf{v}_2, \mathbf{w}_2)_\varphi \in E$ and $result[\mathbf{w}_1] \neq result[\mathbf{w}_2] \neq$ UNDEF.
- Finally, a vertex $\mathbf{v}$ has a conflict on φ in G, written $Conflict_G(\varphi, \mathbf{v})$ if $\mathbf{v}$ has a φ-free ancestor yielding a conflict on φ.

The notion of conflict is central in the search for a relevant filter. Intuitively, a graph G represents a compliant

SCC if every path in G that starts form $\mathit{root}[G]$ can be extended to reach $\mathit{succ}[G]$. In order to ensure compliance, filters must then prune the graph by banning all the 'bad' synchronizations that lead to a node that cannot reach $\mathit{succ}[G]$, and by preserving all the 'good' synchronizations that converge to $\mathit{succ}[G]$. Due to the presence of internal choices, the same synchronization can look good at one point, but actually be bad. The definition of conflict captures formally this notion of ambiguous synchronizations, as shown in the following example.

Example 4.3 Consider the graph G in Example 4.2. It is easy to see that $\mathit{Conflict}_G(\{a_{n\to m}\}, \mathbf{u})$. To understand why the situation is tricky, focus on node $\mathbf{v}_2$. In order to guarantee compliance, we have to ban $(\mathbf{v}_2, \mathbf{w}_3)_{\{c_{n\to m}\}}$ and to preserve $(\mathbf{v}_2, \mathbf{w}_2)_{\{a_{n\to m}\}}$. On the other hand, at $\mathbf{v}_1$ we have to prune the edge $(\mathbf{v}_1, \mathbf{w}_1)_{\{a_{n\to m}\}}$ as it leads to a failure. Since $\mathbf{u}$ reaches $\mathbf{v_1}$ and $\mathbf{v_2}$ via τ actions, and a filter has no control on τs, the candidate filter should allow and, at the same time, prohibit the action $a_{n\to m}$ at location m and $\bar{a}_{m\to n}$ at location n. This is clearly impossible, hence the choreography $P_1 \parallel P_2$ cannot be fixed.

After the generation of the reduction graph, the construction goes on by *labelling* the graph in the attempt to propagate the result label from $\mathit{succ}[G]$ back towards $\mathit{root}[G]$. The closure algorithm is implemented by the `LabelGraph(`G`)` procedure which runs the auxiliary procedure `PushLabels(`G`)` twice. `PushLabels(`G`)` is actually the core of the algorithm. It receives a graph G generated by `BuldGraph(`C`)` and updates just the UNDEF vertices of G leaving the label SUCC of $\mathit{succ}[G]$ and the label FAIL of the vertex without outgoing edges unchanged. The procedure determines the result label at each vertex $\mathbf{u}$ based on the result labels at the vertices adjacent to $\mathbf{u}$: specifically, $\mathbf{u}$ is set to FAIL if there exists at least one silent transition from $\mathbf{u}$ to a FAIL vertex; $\mathbf{u}$ is set to SUCC if either there are no silent transitions from $\mathbf{u}$ to a FAIL vertex and there exists a silent transition from $\mathbf{u}$ that lead to a SUCC vertex or there exists one non-silent and non-conflicting transition from $\mathbf{u}$ to a SUCC vertex. The procedure iteratively examines all the vertices in the graph until it reaches a fixed point, i.e., no vertex gets updated in the main loop iteration.

The following lemma ensures that the algorithm terminates.

Lemma 4.4 The following conditions hold during the iterations of the main loop in `PushLabels(`G`)`. For every node $\mathbf{u}$ in G:

- $\mathit{result}[\mathbf{u}]$ changes to FAIL and SUCC at least once;
- $\mathit{result}[\mathbf{u}]$ never changes to UNDEF;
- when $\mathit{result}[\mathbf{u}] =$ FAIL, then the label does not change anymore during the computation.

Moreover, at the end of the loop the variable *done* is `false` if and only if some node $\mathbf{u}$ has changed its status during the current loop. Hence the procedure `PushLabels(`G`)` terminates.

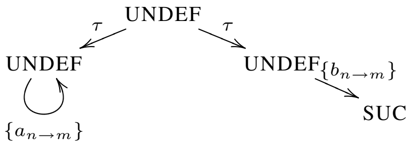
The following lemma states the main property of the procedure `PushLabels(`G`)`.

Lemma 4.5 After the call to `PushLabels(`G`)`, the following conditions hold for every node $\mathbf{u}$ in G:

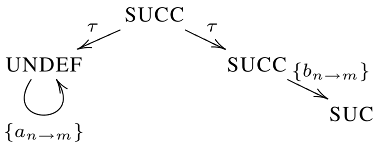
- $\mathit{result}[\mathbf{u}] =$ FAIL iff either there exists no $(\mathbf{u}, \mathbf{v})_\varphi \in E$ such that $\mathit{result}[\mathbf{v}] =$ SUCC and $\neg\mathit{Conflict}_G(\varphi, \mathbf{u})$ or there exists $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $\mathit{result}[\mathbf{v}] =$ FAIL;
- $\mathit{result}[\mathbf{u}] =$ SUCC iff there exists no $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $\mathit{result}[\mathbf{v}] =$ FAIL and there exists either $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $\mathit{result}[\mathbf{v}] =$ SUCC or $(\mathbf{u}, \mathbf{v})_\varphi \in E$ with $\varphi \neq \tau$, $\neg\mathit{Conflict}_G(\varphi, \mathbf{u})$ and $\mathit{result}[v] =$ SUCC;
- $\mathit{result}[\mathbf{u}] =$ UNDEF iff $\mathit{result}[\mathbf{v}] =$ UNDEF for every node $\mathbf{v}$ that is reachable form $\mathbf{u}$.

By Lemma 4.5 and Lemma 4.1, it follows that if $\mathit{result}[\mathbf{u}] =$ UNDEF after a run of `PushLabels(`G`)` then it is impossible to reach any leaf of G from $\mathbf{u}$. Hence $\mathbf{u}$ is inside a cycle of UNDEF vertices. In this case, vertex $\mathbf{u}$ must be marked FAIL. This is accomplished by the `LabelGraph(`G`)` procedure which first runs `PushLabels(`G`)`, then it sets to FAIL every vertex that is still UNDEF. After that, it runs `PushLabels(`G`)` again. To see why this additional run is needed, consider the following example.

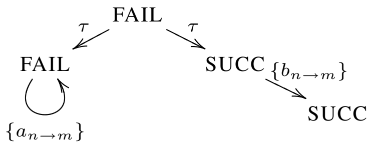
Example 4.6 The SCC $C_2 = [(\texttt{rec}(x)\, a.x) \oplus b.\mathbf{1}]_m \parallel [(\texttt{rec}(x)\, \bar{a}@m.x) + \bar{b}@m.\mathbf{1}]_n$ generates the following reduction graph G:



After the first run of `PushLabels(`G`)`, we update the vertices of G as follows:



Now the UNDEF node must clearly be set to FAIL. Thus, a further run of `PushLabels(`G`)` is needed to propagate the failure to the root.



Thanks to Lemma 4.5 and the fact that at the end of LabelGraph(G) there exists no UNDEF vertex in G, we may easily derive a post condition for LabelGraph(G). We first need an auxiliary definition. Say that a path π in G is *successful* if $\mathit{result}[\mathbf{u}] = \textsc{succ}$ for every node $\mathbf{u}$ in π, otherwise we say that π is *unsuccessful*. A vertex $\mathbf{u}$ is *root-successful* if it is reachable from $\mathit{root}[G]$ via a successful path, otherwise it is *root-unsuccessful*.

Lemma 4.7 Let G be a graph generated by the procedure BuildGraph(C), then after the call LabelGraph(G)

1. for every root-successful node $\mathbf{u}$ in G, there exists a successful path from $\mathbf{u}$ to $\mathit{succ}[G]$.
2. for every root-unsuccessful node $\mathbf{u}$ in G, either there exists no path from $\mathbf{u}$ to $\mathit{succ}[G]$ or there exists a path π from $\mathbf{u}$ to $\mathbf{v}$ with $\mathit{result}[\mathbf{v}]=\textsc{fail}$ and there exist $\mathbf{u}'$, $\mathbf{v}'$ such that $(\mathbf{u}', \mathbf{v}')_\tau \in \pi$.

```
Procedure PushLabels($G$)
  Input: A reduction graph $G = (V, E)$
  Output: The graph $G$ updated
  $done$ := false;
  while $\neg done$ do
      $done$ := true;
      foreach $\mathbf{u} \in V$ do
          $succ$ := false;
          $fail$ := false;
          if $Adj[\mathbf{u}, \tau] \neq \emptyset$ then
              if $\exists \mathbf{v} \in Adj[\mathbf{u}, \tau] : result[\mathbf{v}] = \textsc{fail}$ then
                  $fail$ := true;
              else if $\exists \mathbf{v} \in Adj[\mathbf{u}, \tau] : result[\mathbf{v}] = \textsc{succ}$
              then
                  $succ$ := true;
              end
          else if $\exists (\varphi, \mathbf{v}) \in Adj[\mathbf{u}] \wedge result[\mathbf{v}] =$
          $\textsc{succ} \wedge \neg Conflict(\varphi, \mathbf{u})$ then
              $succ$ := true;
          end
          if $succ \wedge result[\mathbf{u}] \neq \textsc{succ}$ then
              $result[\mathbf{u}] := \textsc{succ}$;
              $done$ := false;
          else if $fail \wedge result[\mathbf{u}] \neq \textsc{fail}$ then
              $result[\mathbf{u}] := \textsc{fail}$;
              $done$ := false;
      end
  end
```

```
Procedure LabelGraph($G$)
  Input: A reduction graph $G = (V, E)$
  Output: The graph $G$ updated
  PushLabels($G$);
  foreach $\mathbf{u} \in V$ do
      if $result[\mathbf{u}] = \textsc{undef}$ then
          $result[\mathbf{u}] := \textsc{fail}$;
      end
  end
  PushLabels($G$);
```

Extracting the success subgraph. Item 1 of Lemma 4.7 hints that the root-successful vertices of G represent the computation of a compliant configuration, as they satisfy the requirements of Definition 2.1. Item 2 of the same lemma hints that if we consider just a root-unsuccessful node, we make this ideal computation break compliance. Given that, the next step of the algorithm computes the sub-graph of G that only includes the root-success vertex, along with the edges which compose the successful paths. This computation is accomplished by the SuccessGraph(G) function below, by simply selecting the set of root-successful states (and the associated transitions), assuming the root itself is successful. The correctness of the procedure is given by the following lemma.

Lemma 4.8 Let $G = (E, V)$ be a graph generated by the procedure BuildGraph(C), and let $G' = (E', V')$ the graph generated by SuccessGraph(G). Then $\mathbf{u} \in V'$ if and only if $\mathbf{u}$ is root-successful in G.

```
Function SuccessGraph($G$)
  Input: A closed reduction graph $G = (V, E)$
  Output: $G' = (V', E')$ the success sub-graph of $G$
  $V' := (result[root[G]] = \textsc{succ})\ ?\ \{root[G]\}\ :\ \emptyset$;
  $E' := \emptyset$;
  $done$ := false;
  while $\neg done$ do
      $done$ := true;
      foreach $(\mathbf{u}, \mathbf{v})_\varphi \in E \setminus E'$ do
          if
          $\mathbf{u} \in V' \wedge result[\mathbf{v}] = \textsc{succ} \wedge \neg Conflict(\varphi, \mathbf{u})$
          then
              $V' := V' \cup \{\mathbf{v}\}$; $E' := E' \cup \{(\mathbf{u}, \mathbf{v})_\varphi\}$;
              $done$ := false
          end
      end
  end
  return $G' = (V', E')$;
```

Synthesizing the filter. The final step of the algorithm synthesizes the filter out of the success graph, in case this is not empty. The filter may in fact simply be extracted from the success graph by projecting the complementary actions of each synchronization step in the graph on to the contributing locations.

```
Function ExtractFilter_L(u, U, G)

Input: $G = (V, E)$ a success graph. $\mathbf{u} \in V, U \subseteq V$
Output: $F$, an $L$-composite filter

$F[\ell] := \mathbf{0}$ for all $\ell \in L$.;
if $state[\mathbf{u}] \checkmark$ then
    return $F$;
end
if $\mathbf{u} \in U$ then
    $rec[\mathbf{u}] := \mathtt{true}$; return $(X_{\mathbf{u}}, \ldots, X_{\mathbf{u}})$;
end
foreach $(\varphi, \mathbf{v}) \in Adj[\mathbf{u}]$ do
    $F_{\mathbf{v}} := \mathtt{ExtractFilter}_L(\mathbf{v}, U \cup \{\mathbf{u}\}, G)$;
    foreach location $\ell \in L$ do
        if $\varphi = \{a_{\ell \to \_}\}$ then
            $F[\ell] := F[\ell] \times \bar{a}_{\ell \to \_}.F_{\mathbf{v}}[\ell]$;
        else if $\varphi = \{a_{\_ \to \ell}\}$ then
            $F[\ell] := F[\ell] \times a_{\_ \to \ell}.F_{\mathbf{v}}[\ell]$;
        else
            $F[\ell] := F[\ell] \times F_{\mathbf{v}}[\ell] j$;
        end
    end
end
if $rec[\mathbf{u}] = \mathtt{true}$ then
    foreach $\ell \in L :\ X_{\mathbf{u}} \in fv(F[\ell])$ do
        $F[\ell] := \mathtt{rec}(X_{\mathbf{u}})\, F[\ell]$;
    end
end
return $F$;
```

Let F be the synthesized filter. By construction, it follows that the success graph of the original composition C coincides with the reduction graph of the composition obtained by filtering C with F. Then, thanks to item 1 of Lemma 4.7 we derive the correctness of the algorithm. We assume that $G = \mathtt{BuildGraph}(C)$, $G' = \mathtt{SuccessGraph}(G)$, and $F = \mathtt{ExtractFilter}_L(root[G], \emptyset, G')$. Then we have

Theorem 4.9 [SOUNDNESS OF THE ALGORITHM] Let C be a service contract composition, and F_C be the filter synthesized by the algorithm. Then $F_C^{Alg} \triangleright C$ is compliant.

By construction, F_C^{Alg} is relevant for C. Furthermore, item 2 of Lemma 4.7 and Lemma 4.8 imply that F_C^{Alg} is maximum among the relevant filters that fix C.

Theorem 4.10 [MAXIMALITY] If a filter F fixes C, and is relevant for C, then $F \leq F_C^{Alg}$.

Corollary 4.11 [COMPLETENESS] Let C be a service-contract composition. If C is weakly compliant, then the algorithm succeeds and extracts the maximum relevant filter.

5. Conformance Validation

Filters have an additional purpose in our theory, as behavioral descriptors: in fact, we employ them to specify the intended roles of a choreography. This is possible as filters are built around actions that provide accurate information on the end-points involved in the operations of message sent/reception they represent. That information is precisely what we need when projecting a choreographic design on the components to define the role (i.e., the expected behavior) of each component.

We then check the conformance of a composition against a choreographic specification as we outline next. We disregard the definition of the choreography specification language, and the details of how a choreography specification may to generate the corresponding specs for each of the component roles. We refer the reader to, e.g., [4], for an example of how that can be accomplished, and assume that choreography specifications are given directly in projected form, as a set $\Phi = \{\Phi(\ell) \mid \ell \in L\}$ of role specifications, with each role specified by a filter. We let $\stackrel{\alpha}{\Longrightarrow}$ denote the weak SSC transition $\xrightarrow{\tau}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}^{*}$, and $\stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow}$ a sequence of such transitions. When $n = 0$, $\stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow}$ stands for $\xrightarrow{\tau}^{*}$.

Definition 5.1 [ROLE SUPPORT] A binary relation $\mathcal{S}$ between (role) filters and service contracts is a role simulation if whenever $(\phi, [\sigma]_\ell) \in \mathcal{S}$ one has:

- if $\phi \stackrel{\alpha}{\longmapsto} \phi'$ there exist $\alpha_1, \ldots, \alpha_n, \sigma'$ such that $[\sigma]_\ell \stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \cdot \xrightarrow{\alpha} [\sigma']_\ell$ with $\alpha \neq \alpha_i$ and $(\phi', [\sigma']_\ell) \in \mathcal{S}$;
- if $[\sigma]_\ell \stackrel{\alpha}{\Longrightarrow} [\sigma']_\ell$ then either $\phi \stackrel{\alpha}{\longmapsto} \phi'$ and $(\phi', [\sigma']_\ell) \in \mathcal{S}$ or $\phi \stackrel{\alpha}{\not\longmapsto}$ and $(\phi, [\sigma']_\ell) \in \mathcal{S}$.

A contract $[\sigma]_\ell$ supports a role ϕ, noted $\phi \lhd [\sigma]_\ell$, if $(\phi, [\sigma]_\ell) \in \mathcal{S}$ with $\mathcal{S}$ role simulation. A composition C supports a specification Φ, written $\Phi \lhd_L C$ iff $\Phi(\ell) \lhd [\sigma_\ell]_\ell$ for all $\ell \in L$.

A role simulation allows the simulating contract to have additional observable behavior over the simulated role, but only if the additional behavior preserves the branching structure of the role.

Definition 5.2 [CONFORMANCE] A contract composition C is Φ-conformant iff F_C^{Alg} exists and $\Phi \lhd (F_C^{Alg} \triangleright C)$.

Requiring $F_C^{Alg} \triangleright C$ to support Φ gives a guarantee of functional completeness for the filtered choreography. On the other hand, the properties of F_C^{Alg} ensure that the additional behavior exposed by $(F_C^{Alg} \triangleright C)$, if any, is safe. To illustrate, let $\phi = a_{m \to \ell}.\phi'$ be the role, and $\sigma = a.\sigma_1 \oplus b.a.\sigma_2$ be the contract at ℓ. Then, $\phi \lhd [\sigma]_\ell$ whenever $\phi' \lhd [\sigma_1]_\ell$ and $\phi' \lhd [\sigma_2]_\ell$. Indeed, when $[\sigma]_\ell$ is a filtered location, we know that the 'spurious' action b does not deadlock (otherwise $b.a.\sigma_2$ would have been filtered away). On the other hand $\phi \not\lhd [a.\sigma_1 \oplus b]_\ell$ as the service at ℓ may deliberately choose not to execute a as required by its role.

6. Related Work

Most of the proposals for verification of web services choreographies/orchestrations are based on encoding services within a mathematically well-founded model, formalizing the properties of interest in terms of logical formulae, and then devising model checking techniques to validate the composite services against the expected properties.

A number of proposals focus on verification of properties for isolated services: [22, 9, 10, 18, 11, 19] is a sample, but far from exhaustive list of papers that follow this line of work. Closer to our present approach are the proposal that address the compliance problem within choreographies. In [20] the problem is attacked by reducing it to a bipartite compatibility problem. The method allows a distributed checking since each partner can check locally its compliance by checking its compatibility with the aggregation of all its partners. Our present concern is more general, as we also devise a technique for adapting the aggregation to solve the behavioral mismatches that hinder compliance.

Component adaptation has itself received much interest in the literature. Among the many approaches to the problem, (see for instance, [2, 6, 12]), the closest to ours is perhaps [14], where the authors develop a method for the automated extraction of an adapter for a service composition out of the LTS of the composition. The adapter is then deployed as an independent component that orchestrates the the execution of its peer components to ensure safety of the execution flow. More recently [15], the approach has been extended and refined with a technique for projecting the global adapter onto the individual system components.

While we share some of the initial motivations and ideas with these two papers, specifically the idea of extracting the adapter from the LTS, our approach is different for a number of design choices and technical results. First, we extract the filters directly from the individual components rather obtaining them by a projection from a global adapter. Secondly, our filter fully preserves the action sequence in the original components, whereas the adapter synthesized in [15] may require to reorder them. Also, our formalization of adapters as filters makes it possible to formulate, and prove formally, a precise characterization of the properties satisfied by the extracted filter (i.e. relevance and maximality). Finally, as we have shown our theory provides a uniform and fairly elegant setting for a combined analysis of compliance and conformance in terms of well-established behavioral techniques.

References

[1] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *WS-FM05*, volume 3670 of *LNCS*, pages 257–271. Springer-Verlag, 2005.

[2] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[3] M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *FSEN'07*, volume 4767 of *LNCS*, pages 207–222. Springer, 2007.

[4] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC'07*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.

[5] M. Bravetti and G. Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *WS-FM'08*, LNCS. Springer, 2008. To appear.

[6] A. Brogi, C. Canal, and E. Pimentel. Component adaptation through flexible subservicing. *Science of Computer Programming*, 63(1):39–56, 2006.

[7] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL'08*, pages 261–272. ACM press, 2008.

[8] G. Deker, O. Kopp, F. Leymann, and M. Weske. Bpel4chor: Extending bpel for modeling choreographies. In *ICWS'07*. IEEE Computer Society, 2007.

[9] A. Ferrara. Web services: a process algebra approach. In *ICSOC'04*, pages 242–251. ACM press, 2004.

[10] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proc. of the 6th International Conference on Entreprise Information Systems*, pages 287–295, 2004.

[11] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proc. of the 14th Australasian Database Conference (ADC'03)*, pages 191–200. ACS, 2003.

[12] P. Inverardi and M. Tivoli. Deadlock free software architectures for com/dcom applications. *Journal of Systems and Software*, 65(3):173–183, 2005.

[13] F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM Software Group, 2002.

[14] R. Mateescu, P. Poizat, and G. Salaün. Behavioral adaptation of component compositions based on process algebra encodings. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–388. ACM Press, 2007.

[15] T. Melliti, P. Poizat, and S. Ben Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *FASE'08*, volume 4961 of *LNCS*, pages 146–162, 2008.

[16] R. Milner. *Communication and Concurrency*, volume 92 of *Prentice Hall International Series in Computer Science*. Prentice Hall, 1989.

[17] OASIS. Web services business process execution language version 2.0. http://www.oasis-open.org/specs/.

[18] B.-H. Schlingloff, A. Martens, , and K. Schmidt. Modeling and model checking web services. *ENTCS*, pages 3–26, 2005.

[19] K. Schmidt and C. Stahl. A petri net semantic for bpel4ws - validation and application. In *Proc. of the Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, pages 1–6, 2004.

[20] M. Tarek, C. Boutrous-Saab, and S. Rampacek. Verifying correctness of web services choreography. In *ECOWS'06*, pages 306–318. IEEE Computer Society, 2006.

[21] S. Thatte. Xlang: Web services for business process design. Technical report, Microsoft Corporation, 2001.

[22] F. van Breugel and M. Koshkina. Dead-path-elimination in bpel4ws. In *ACSD'05*, pages 192–201. IEEE Computer Society, 2005.

[23] W. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. Verbeek. Choreography conformance checking: An approach based on bpel and petri nets. Technical Report BPM-05-25, BPMcenter.org, 2005.

[24] W3C. Web services choreography description language. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217.

An Example

Figure 1 shows an example of a compliant service contract composition. Three services are involved in this composition: one representing a *customer*, another one a *supplier*, and a third one a *bank*. The elementary actions represent business activities that result in messages being sent or received. For example, the action $\overline{\mathtt{Request}}@S$ undertaken by the customer results in a message being sent to the supplier. In this example, every message sending action has a corresponding message receipt action. In this scenario, the customer sends a request for goods to the supplier and then decides whether to pay in cash or by an electronic card, which can be either a debit card or a credit card. The supplier contacts its referring bank and waits for the (internal) decision of customer: in case of cash payment he accepts the money, then it sends the goods and closes the communication with the bank, in case of electronic transaction he checks with the bank the availability of the money and then he sends the goods to the customer.

The SCC *Customer* $\|$ *Supplier* $\|$ *Bank* is compliant since the bank is ready to fulfill both the electronic payments. On the other hand, we lose compliance if we replace *Bank* with *Bank**, as *Bank** only accepts credit cards: as a result the composition gets stuck in case the customer and the supplier agree on a payment by credit card. A run of our algorithm will produce the filter F which fixes the composition.

Figure 1 A Choreography for E-payment

Services : C, S, B

$$
\begin{array}{lcl}
\mathit{Customer} & = & [\overline{\mathtt{Request}}@S.(\,(\overline{\mathtt{PayDebit}}@S.\mathtt{GetProd}.\mathbf{1} + \overline{\mathtt{PayCredit}}@S.\mathtt{GetProd}.\mathbf{1}) \oplus \\
 & & \qquad\qquad \overline{\mathtt{PayCash}}@S.\overline{\mathtt{GetCash}}@S.\mathtt{GetProd}.\mathbf{1}\,)]_C \\
\mathit{Supplier} & = & [\mathtt{Request}.\overline{\mathtt{Request}}@B.(\\
 & & \quad \mathtt{PayDebit}.\overline{\mathtt{CheckDebit}}@B.\mathtt{Done}.\overline{\mathtt{GetProd}}@C.\mathbf{1} + \\
 & & \quad \mathtt{PayCredit}.\overline{\mathtt{CheckCredit}}@B.\mathtt{Done}.\overline{\mathtt{GetProd}}@C.\mathbf{1} + \\
 & & \quad \mathtt{PayCash}.\mathtt{GetCash}.\overline{\mathtt{GetProd}}@C.\overline{\mathtt{Done}}@B.\mathbf{1})]_S \\
\mathit{Bank} & = & [\mathtt{Request}.(\mathtt{CheckDebit}.\overline{\mathtt{Done}}@S.\mathbf{1} + \mathtt{CheckCredit}.\overline{\mathtt{Done}}@S.\mathbf{1} + \mathtt{Done}.\mathbf{1})]_B \\
\mathit{Bank}^* & = & [\mathtt{Request}.(\mathtt{CheckCredit}.\overline{\mathtt{Done}}@S.\mathbf{1} + \mathtt{Done}.\mathbf{1})]_B
\end{array}
$$

Filter : F

$$
\begin{array}{lcl}
F[C] & = & \overline{\mathtt{Request}}_{C\to S}.(\,\overline{\mathtt{PayCredit}}_{C\to S}.\mathtt{GetProd}_{S\to C}.\mathbf{0} \times \overline{\mathtt{PayCash}}_{C\to S}.\overline{\mathtt{GetCash}}_{C\to S}.\mathtt{GetProd}_{S\to C}.\mathbf{0}\,) \\
F[S] & = & \mathtt{Request}_{C\to S}.\overline{\mathtt{Request}}_{S\to B}.(\,\mathtt{PayCredit}_{C\to S}.\overline{\mathtt{CheckCredit}}_{S\to B}.\mathtt{Done}_{B\to S}.\overline{\mathtt{GetProd}}_{S\to C}.\mathbf{0} \times \\
 & & \qquad\qquad \mathtt{PayCash}_{C\to S}.\mathtt{GetCash}_{C\to S}.\overline{\mathtt{GetProd}}_{S\to C}.\overline{\mathtt{Done}}_{S\to B}.\mathbf{0}\,) \\
F[B] & = & \mathtt{Request}_{S\to B}.(\mathtt{CheckCredit}_{S\to B}.\overline{\mathtt{Done}}_{B\to S}.\mathbf{0} \times \mathtt{Done}_{S\to B}.\mathbf{0})
\end{array}
$$