Abstract Processes in Orchestration Languages*

Maria Grazia Buscemi[1] and Hernán Melgratti[2]

[1] IMT Lucca Institute for Advanced Studies, Italy
m.buscemi@imtlucca.it
[2] FCEyN, University of Buenos Aires, Argentina
hmelgra@dc.uba.ar

Abstract. Orchestrators are descriptions at implementation level and may contain sensitive information that should be kept private. Consequently, orchestration languages come equipped with a notion of *abstract processes*, which enable the interaction among parties while hiding private information. An interesting question is whether an abstract process accurately describes the behavior of a concrete process so to ensure that some particular property is preserved when composing services. In this paper we focus on compliance, i.e, the correct interaction of two orchestrators and we introduce two definitions of abstraction: one in terms of traces, called *trace-based abstraction*, and the other as a generalization of symbolic bisimulation, called *simulation-based abstraction*. We show that simulation-based abstraction is strictly more refined than trace-based abstraction and that simulation-based abstraction behaves well with respect to compliance.

1 Introduction

An *orchestrator* describes the execution flow of a single party in a composite service. The execution of an orchestrator takes control of service invocation, handles service answers and data flow among the different parties in the composition. Since orchestrators are descriptions at implementation level and may contain sensitive information that should be kept private to each party, orchestration comes equipped with the notion of *abstract processes*, which enable the interaction of parties while hiding private information. Essentially, abstract processes are partial descriptions intended to expose the protocols followed by the actual, concrete processes. Typically, abstract processes are used for slicing the interactions of a concrete process over a fixed set of ports. Consider the following scenario in which an organization sells goods that are produced by a different company. The process that handles order requests can be written as follows (we use CCS [15] extended with value-passing and arithmetic operations).

$$C_1 \stackrel{def}{=} order(desc).\overline{askProd}\langle desc\rangle.answProd(cost).\overline{reply}\langle cost \times 1.1\rangle$$

The process C_1 starts by accepting an order (i.e., a message on port *order*). Then, the received order is forwarded to the actual producer (message $\overline{askProd}\langle desc\rangle$) to obtain a quotation (message on port *answProd*). Finally, the client request is answered by sending the production cost incremented by a 10% (message $\overline{reply}\langle cost \times 1.1\rangle$). We

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA.

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 301–315, 2009.

can define an abstract process that at the same time hides the sensible details of the organization (e.g., the source of the offered goods and the percentages earned) and gives enough information to the client for allowing interaction. In fact, it would be enough for a client to know that orders are placed with a message in port *order* and the quotation is received on port *reply*. For instance, we can define the following abstract process (where τ stands for a silent, hidden action) showing the interaction of C_1 with a client.

$$A_{C_1} \stackrel{def}{=} order(desc).\tau.\tau.\overline{reply}\langle cost\rangle$$

Abstract processes can be also used to hide particular values and internal decisions made by concrete processes. Consider the following process used for authorizing loans.

$$C_2 \stackrel{def}{=} request(amount, salary).\texttt{if}\ (salary > amount/50salary)\ \texttt{then}\ \overline{refuse}\langle\rangle\ \texttt{else}\ \overline{approved}\langle\rangle$$

Note that a loan is approved only when the requested amount is at most 50 times the solicitor's salary. Suppose also that the bank does not want to publicly declare this policy. This can be achieved by providing an abstract processes where some values are *opaque*, i.e., not specified. We denote opaque elements with $\Box$. Then, the abstract process of C_2 can be written as follows.

$$A_{C_2} \stackrel{def}{=} request(amount, salary).\texttt{if}\ salary > \Box\ \texttt{then}\ \overline{refuse}\langle\rangle\ \texttt{else}\ \overline{approved}\langle\rangle$$

The conditional process in A_{C_2} has to be thought of as an internal, non-deterministic choice in which the bank may decide either to approve or to refuse the application. That is, the client cannot infer from A_{C_2} the actual decision that the bank will take.

Then, the main question is whether an abstract process is a *suitable* abstraction of a concrete one or, symmetrically, when a concrete process is a *proper* instantiation of an abstract one. Suitable and proper mean that the abstraction relation should preserve some particular property about composition. In this paper we will focus on *compliance* [12], which specifies whether two partners are able to complete their interaction. So, in terms of compliance, a suitable abstraction means that whenever a pair of services are compliant, we can substitute a service with a more concrete one (according to the abstraction relation) and the composition is still compliant.

In this work we give a formal definition of compliance and propose two alternative definitions for the *abstraction* relation. Our first characterization of abstraction relies on a notion of abstraction of the traces of a process: a process P is an abstraction of a process Q with respect to a set of visible names V if the set of traces of P coincides with the set of traces of Q after the removal of all hidden names. As expected, testing whether two processes belong to this relation requires comparing infinite sets of traces. Hence, we give an efficient version of trace abstraction that only requires checking finitely-many symbolic traces and we prove that the two trace-based relations coincide.

The second notion of abstraction that we propose states that the abstract process and the concrete process must be able to simulate each other behaviors when hiding a given set of names in the concrete process. In general, this notion is not a bisimulation. For example, a process $P = \tau.\texttt{if}\ a = \Box\ \texttt{then}\ \overline{y}\langle a\rangle\ \texttt{else}\ \overline{z}\langle a\rangle$ is the abstraction of a process $Q = x(u).\texttt{if}\ a = u\ \texttt{then}\ \overline{y}\langle a\rangle\ \texttt{else}\ \overline{z}\langle a\rangle$ when hiding x but, of course, P and Q are not bisimilar since P has more computations. We show that our simulation-based relation

is strictly finer than trace-based abstraction. Finally, we show that this second notion of abstraction preserves compliance.

Related works. The problem of giving suitable abstractions of the behaviour of a concrete system is not new. In fact, different flavours of the same general problem have been studied in the literature ([15,2,1,6,8], just to name a few). Session types [11,7,9] and, more recently, contracts [12,5,4] provide a framework for checking whether a client is compliant with a service and whether a process can be "safely" replaced with another one (a detailed comparison among session types and contracts can be found in [13]). Our proposal shares aims with the above approaches but there are three main differences with those models. First, our abstraction relations are neither trace inclusion nor simulation. Hence, $a \leq a+b$ for $+$ an external choice does not hold in our case (roughly speaking, we do not allow abstract processes to exhibit more behaviors than their associated concrete processes). Indeed, if we hide b it holds $a+\tau \leq_{\{a\}} a+b$. Second, the main focus of contracts and session types is on the interplay between external and internal choice, while the abstraction relations we define, which specify hiding data and turning external choice and conditional statements into internal choice, have no immediate counterpart in the those models. Third, our processes include actions that not only record the type of communication but also the transmitted data and two branching structures, if-then-else and guarded choice, that do not tightly match internal and external choice in contracts or branching and choice in session types.

2 Concrete Processes

The computation model we describe is highly inspired by the composition model of WS-BPEL, which can be roughly described as follows: a composite service can be though as the parallel composition of several orchestrators that interact by exchanging XML-documents using one of the basic actions, i.e., invoke a service operation ($<$ `invoke` $>$), receive a message ($<$ `receive` $>$), and reply to a previous invocation ($<$ `reply` $>$). An orchestrator is a program built up from basic actions that are composed into sequences ($<$ `sequence` $>$), parallel flows ($<$ `flow` $>$), conditional statements ($<$ `switch` $>$), iteration blocks ($<$ `while` $>$), and in choice statements ($<$ `pick` $>$). Moreover, an orchestrator is not intended to use primitives $<$ `invoke` $>$, $<$ `receive` $>$ and $<$ `reply` $>$ to synchronize with itself. For this reason, we divide the presentation of the computation model in two parts: (i) the language of *concrete processes* (introduced in this section), which is intended to model the behavior of a single orchestrator; and (ii) the language of *concrete business processes* (presented in § 8) that focuses on the interaction among several orchestrators.

The remaining of this section is devoted to the presentation of the language of concrete processes, which is a version of value-passing CCS [16] with input guarded choices and conditional statements but without recursion.

2.1 Syntax

We assume an infinite denumerable set of names $\mathcal{N}$, ranged over by η, that is partitioned into a set of port names $\mathcal{X}$, ranged over by $x, y, z, \ldots$, a set of data variables $\mathcal{V}$, ranged

over by $u, v, \ldots$, and a set of data constants $\mathcal{C}$, ranged over by $a, b, c, \ldots$. We let $m, n, \ldots$ range over $\mathcal{V} \cup \mathcal{C}$. We write $\tilde{\eta}$ for a tuple of names. *Substitutions*, ranged over by σ, are partial maps from $\mathcal{V}$ onto $\mathcal{V} \cup \mathcal{C}$. Domain and co-domain of σ, noted $dom(\sigma)$ and $cod(\sigma)$, are defined as usual. By $m\sigma$ we denote $\sigma(m)$ if $m \in dom(\sigma)$, and m otherwise.

Definition 1 (Concrete processes). *The set of* concrete processes P *is given by the following grammar:*

$$P \;::=\; 0 \;\big|\; P|P \;\big|\; \tau.P \;\big|\; \overline{x}\langle\widetilde{m}\rangle.P \;\big|\; x_1(\widetilde{v_1}).P + \ldots + x_n(\widetilde{v_n}).P \;\big|\; \texttt{if } m = n \texttt{ then } P \texttt{ else } P$$

As usual, 0 stands for the inert process, $P|P$ for the parallel composition of processes, $\tau.P$ for the process that performs a silent action and then behaves like P, $\overline{x}\langle\widetilde{m}\rangle.P$ for the process that sends the message m over the port x and then becomes P. The process $x_1(\widetilde{v_1}).P_1 + \ldots + x_n(\widetilde{v_n}).P_n$ denotes an external choice in which some process $x_i(\widetilde{v_i}).P_i$ is chosen when the corresponding guard $x_i(\widetilde{v_i})$ is enabled. The conditional process $\texttt{if } m = n \texttt{ then } P \texttt{ else } P'$ behaves either as P if m and n are syntactically equivalent, or as P' otherwise. For convenience, here we restrict to equality constraints. However, more complex constraints could be "encoded" under certain conditions. Hereafter, we adopt the usual convention of omitting trailing 0's.

In $x_1(\widetilde{v_1}).P_1 + \ldots + x_n(\widetilde{v_n}).P_n$, the data variables v_i are bound, for all i. We use the standard notions of *free* and *bound* names of processes, noted respectively as $fn(P)$ and $bn(P)$, and α-conversion on bound names. Without loss of generality, we assume that the sets of free and bound names are disjoint and that the bound names of a process are all distinct from each other. As usual, a process P is *closed* if $fn(P) \cap \mathcal{V} = \emptyset$.

2.2 Operational Semantics

The operational semantics, as usual, is given in two steps: the definition of a *structural congruence*, which rearranges processes into adjacent positions, and a notion of *labeled transition relation* that captures computation on processes.

We define *structural congruence*, $\equiv$, as the least congruence over concrete processes closed with respect to α-conversion and satisfying the following rules:

$$P|0 \equiv P \qquad P_1|P_2 \equiv P_2|P_1 \qquad (P_1|P_2)|P_3 \equiv P_1|(P_2|P_3)$$

Let *actions* α range over *silent move*, *free input* and *free output*:

$$\alpha ::= \tau \mid x\langle\widetilde{a}\rangle \mid \overline{x}\langle\widetilde{a}\rangle.$$

As usual, for $\alpha \neq \tau$, $subj(\alpha)$ and $obj(\alpha)$ denote the subject and the object of α respectively. For X a process or an action, $X\sigma$ denotes the expression obtained by replacing in X each data variable $u \in fn(X)$ with $u\sigma$, possibly α-converting to avoid name capturing.

The labeled transition relation $\xrightarrow{\alpha}$ over concrete closed processes is the least relation satisfying the inference rules in Table 1. The transition rules for processes are the standard ones for value passing CCS. We only add rules (IF) and (ELSE) for handling conditional statements of processes. As mentioned before, we do not include here the standard communication rule, because our model allows synchronizations only among different orchestrators (see Section 8).

Table 1. LTS for concrete processes

$$\text{(TAU)}\quad \tau.P \xrightarrow{\tau} P \qquad \text{(OUT)}\quad \overline{x}\langle\widetilde{a}\rangle.P \xrightarrow{\overline{x}\langle\widetilde{a}\rangle} P \qquad \text{(IN)}\quad x_1(\widetilde{v_1}).P_1+\ldots+x_n(\widetilde{v_n}).P_n \xrightarrow{x_i\langle\widetilde{a}\rangle} P_i\{\widetilde{a}/\widetilde{v_i}\}$$

$$\text{(IF)}\quad \frac{P \xrightarrow{\alpha} P'}{\texttt{if } a=a \texttt{ then } P \texttt{ else } Q \xrightarrow{\alpha} P'} \qquad \text{(ELSE)}\quad \frac{Q \xrightarrow{\alpha} Q' \quad a\neq b}{\texttt{if } a=b \texttt{ then } P \texttt{ else } Q \xrightarrow{\alpha} Q'}$$

$$\text{(PAR)}\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{(STR)}\quad \frac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad Q' \equiv P'}{P \xrightarrow{\alpha} P'}$$

The following result shows that the labeled transition relation is well-defined. The proof is by induction on the structure of P and on the transition rules.

Proposition 1. *Let P be a closed process. If $P \xrightarrow{\alpha} Q$ then Q is a closed process.*

3 Abstract Processes

Abstract processes are defined by using the primitives of the concrete processes plus the possibility of having *opaque* definitions. An opaque element hides the precise value of an element: for instance, an opaque assignment to a data variable hides the assigned value. We denote an opaque element by the special name $\Box$, and we assume $\Box \notin \mathcal{N}$.

The definition of *abstract processes* is analogous to the definition of concrete processes, but making $a,b,\ldots$ range over $\mathcal{C} \cup \{\Box\}$, $m,n,\ldots$ range over $\mathcal{V} \cup \mathcal{C} \cup \{\Box\}$, and $x,y,\ldots$ range over $\mathcal{X} \cup \{\Box\}$. Hence, opaque names can appear either as subjects of input and output prefixes, values of output prefixes, or parts of conditions in `if _ then _ else _` processes, but not as a bound variables. We let $P,Q,R\ldots$ range over abstract processes.

The rules in Table 1 remain unchanged. We assume for the rule (IN) that every a_j can take the value $\Box$ and, hence, $P_i\{\widetilde{a}/\widetilde{v_i}\}$ is still a process. Rules (IF) and (ELSE) consider only the cases in which the condition does not contain opaque elements. For the case of opaque values we add the rules in Table 2 and the structural congruence axioms below.

$$\Box(\widetilde{v}).P \;\equiv\; \tau.P\{\widetilde{\Box}/\widetilde{v}\} \qquad\qquad \overline{\Box}\langle\widetilde{a}\rangle.P \;\equiv\; \tau.P.$$

Note that a conditional statement becomes a non-deterministic choice when at least one value in the condition is opaque, while a guarded choice becomes an internal choice

Table 2. Additional LTS rules for processes

$$\text{(CHOICE-1)}\quad \frac{P_1 \xrightarrow{\alpha} P_1' \quad \Box \in \{a,b\}}{\texttt{if } a=b \texttt{ then } P_1 \texttt{ else } P_2 \xrightarrow{\alpha} P_1'} \qquad \text{(CHOICE-2)}\quad \frac{P_2 \xrightarrow{\alpha} P_2' \quad \Box \in \{a,b\}}{\texttt{if } a=b \texttt{ then } P_1 \texttt{ else } P_2 \xrightarrow{\alpha} P_2'}$$

$$\text{(CHOICE-3)}\quad x_1(\widetilde{v_1}).P_1+\ldots+\Box(\widetilde{v_i}).P_i+\ldots+x_n(\widetilde{v_n}).P_n \xrightarrow{\tau} P_i\{\widetilde{\Box}/\widetilde{v_i}\}$$

when the subject of the input guard is the opaque name. For instance, consider the process $R \equiv \Box(v_1).P + x(v_2).Q$. A possible move for R is $R \xrightarrow{x\langle a\rangle} Q\{^a/_{v_2}\}$, where the input guard is executed. Another possibility is $R \xrightarrow{\tau} P\{^{\Box}/_{v_1}\}$, where R makes an internal choice.

We define the notion of traces over processes as usual. A *trace* t is a sequence of actions $\alpha_1.\cdots.\alpha_n$. The set $Tr(P)$ of traces of a process P is defined as follows:

$$Tr(P) = \{t \mid P \xrightarrow{\alpha_1} P_1 \ldots P_{n-1} \xrightarrow{\alpha_n} P_n \wedge t = \alpha_1.\cdots.\alpha_n\}.$$

4 Symbolic Semantics

This section gives a definition of the symbolic semantics of concrete and abstract processes as a symbolic labeled transition relation over processes. Labels have two components: a symbolic action λ and a Boolean condition M over the set of data variables and data constants $\mathcal{V} \cup \mathcal{C}$ that must hold for the α-transition to be enabled.

We let *symbolic actions* λ range over the *silent move*, *input* and *free output* and we let *conditions* M range over a language of Boolean formulas:

$$\lambda ::= \tau \mid x(\widetilde{v}) \mid \overline{x}\langle\widetilde{m}\rangle \qquad M ::= \mathit{true} \mid \mathit{false} \mid m = n \mid m \neq n \mid M \wedge M \mid M \vee M.$$

The notions of *free* names $fn(\cdot)$, *bound* names $bn(\cdot)$, and α-conversion over actions and conditions are as expected, considering that the occurrences of the names v_i's are bound in $x(\widetilde{v})$ and that conditions have no bound names. By $M\sigma$ we mean the condition obtained by simultaneously replacing in M each data variable $v \in fn(M)$ with $v\sigma$. A condition M is *ground* if M does not contain data variables. The *evaluation* $Ev(M)$ of a ground condition M into the set $\{\mathit{true},\mathit{false}\}$ is defined by extending in the expected homomorphical way the following clauses:

$$\begin{array}{lll} Ev(\mathit{true}) = \mathit{true} & Ev(a = a) = \mathit{true} & Ev(a = b) = \mathit{true} \text{ if } \{a,b\} \cap \Box \neq \emptyset \\ Ev(\mathit{false}) = \mathit{false} & Ev(a = b) = \mathit{false} \text{ if } a,b \neq \Box & Ev(a \neq b) = \mathit{true} \text{ if } \{a,b\} \cap \Box \neq \emptyset \end{array}$$

A substitution σ *respects* M, written $\sigma \models M$, if $M\sigma$ is ground and $Ev(M\sigma) = \mathit{true}$. A condition M is *consistent* if there is a substitution σ such that $\sigma \models M$. A condition M *logically entails* a condition N, written $M \Rightarrow N$, if, for every σ, $\sigma \models M$ implies $\sigma \models N$. For instance, $v = a \wedge u \neq b \wedge v = u \Rightarrow a \neq b$ and $\mathit{true} \Rightarrow u = a \vee u \neq a$. For λ a symbolic action and σ a substitution such that every data variable in λ belongs to $dom(\sigma)$, we write $\lambda\sigma$ to denote the following action:

$$\lambda\sigma \stackrel{\text{def}}{=} \begin{cases} \tau & \text{if } \lambda = \tau \\ \overline{x}\langle a_1,\ldots,a_k\rangle & \text{if } \lambda = \overline{x}\langle n_1,\ldots,n_k\rangle \text{ and } a_i = n_i\sigma \text{ for } i = 1,\ldots,k \\ x\langle a_1,\ldots,a_k\rangle & \text{if } \lambda = x(v_1,\ldots,v_k) \text{ and } a_i = \sigma(v_i) \text{ for } i = 1,\ldots,k \end{cases}$$

By $\lambda = \lambda'$ we denote the following condition:

$$\lambda = \lambda' \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } \lambda = \lambda' = \tau \text{ or } \lambda = \lambda' = x(\widetilde{v}) \\ \widetilde{m} = \widetilde{n} & \text{if } \lambda = \overline{x}\langle\widetilde{m}\rangle \text{ and } \lambda' = \overline{x}\langle\widetilde{n}\rangle \\ \mathit{false} & \text{otherwise} \end{cases}$$

For M a condition and $D = \{M_1, \ldots, M_n\}$ a finite set of conditions, D is a *M-decomposition* if $M \Rightarrow M_1 \vee \ldots \vee M_n$. For instance, $\{u = a, u \neq a\}$ is a *true*-decomposition.

The symbolic labeled transition relation $\xrightarrow{M,\lambda}$ over concrete processes is the least relation satisfying the inference rules in Table 3. The additional symbolic rules for processes are given in Table 4. Each symbolic rule is the counterpart of a rule in Table 1. Intuitively, the condition M in the label M,λ of a transition collects the Boolean constraints on the free data variables of the source process necessary for action λ to take place. For instance, the rules for prefixes say that each prefix can be consumed unconditionally. Differently from rule (IN) in Table 1, input variables are *not* instantiated immediately (rule (S-IN)). Rules (S-IF) and (S-ELSE) make the equalities or inequalities of the conditional statements explicit. As an example, the process $P \equiv x(v).\texttt{if } v = a \texttt{ then } \overline{y}\langle v\rangle \texttt{ else } 0$, after a first step, can make a transition under condition that variable v is equal to a:

$$P \xrightarrow{true,x(v)} \texttt{if } v = a \texttt{ then } \overline{y}\langle v\rangle \texttt{ else } 0 \xrightarrow{v=a,\overline{y}\langle v\rangle} 0$$

Remark that the present rules are simpler than those given in [3] for the pi-calculus because our calculus is a value-passing CCS plus conditional statements and, thus, logical conditions do not affect channel names.

Proposition 2. *Let P be a closed process.*

- *If $P \xrightarrow{\alpha} Q$ then there exist R, M, λ, and $\sigma \models M$ s.t. $P \xrightarrow{M,\lambda} R$, $\alpha = \lambda\sigma$ and $Q = R\sigma$.*
- *If $P \xrightarrow{M,\lambda} Q$ then there exists $\sigma \models M$ such that $P \xrightarrow{\alpha} Q\sigma$ and $\alpha = \lambda\sigma$.*

5 Notion of Abstraction

This section informally presents our notion of abstraction by introducing the ideas that are formalized in the following sections. The abstraction relation is parametric with respect to the names that should be shown by the abstract process. For instance, given the concrete process $P \equiv \overline{x}\langle a\rangle.\overline{y}\langle b,c\rangle$ and the set $V = \{y,a,b\}$ of visible names, we require the abstract process (i) to show every interaction that takes place over channel y, (ii) to hide every interaction occurring over a channel different from y, (iii) to show every occurrence of the data values a and b in visible interactions and (iv) to hide every occurrence of a data value different form a and b. Hence, we expect the abstraction of P to be $Q \equiv \tau.\overline{y}\langle b,\Box\rangle$. Note that the output action on the hidden channel x is mimicked by the silent movement τ (independently from the fact that a is a visible name) and the output $\overline{y}\langle b,c\rangle$ over the channel y is represented in the abstraction as $\overline{y}\langle b,\Box\rangle$, where the hidden value c has been replaced by the opaque element.

A side-effect of hiding concrete elements is the introduction of non-determinism at the abstract level. This may happen either because decisions become opaque or because input guarded choices become internal choices, as shown by the following example.

Example 1. Consider the following two processes

$$P \equiv x(u).\texttt{if } u = a \texttt{ then } \overline{y}\langle b\rangle \texttt{ else } \overline{z}\langle c\rangle \quad Q \equiv x(u).\texttt{if } u = \Box \texttt{ then } \overline{y}\langle b\rangle \texttt{ else } \overline{z}\langle c\rangle$$

We expect Q to be an abstraction of P when a is a hidden name.

Table 3. Symbolic LTS for concrete processes

$$\text{(S-TAU)}\quad \tau.P \xrightarrow{true,\tau} P \qquad \text{(S-OUT)}\quad \overline{x}\langle\widetilde{m}\rangle.P \xrightarrow{true,\overline{x}\langle\widetilde{m}\rangle} P \qquad \text{(S-IN)}\quad x_1(\widetilde{v_1}).P_1+\ldots+x_n(\widetilde{v_n}).P_n \xrightarrow{true,x_i(\widetilde{v_i})} P_i$$

$$\text{(S-PAR)}\quad \frac{P \xrightarrow{M,\lambda} P' \quad bn(\lambda)\cap fn(Q)=\emptyset}{P \mid Q \xrightarrow{M,\lambda} P' \mid Q} \qquad \text{(S-IF)}\quad \frac{P \xrightarrow{M,\lambda} P' \quad m=n\wedge M \text{ consistent}}{\texttt{if } m=n \texttt{ then } P \texttt{ else } Q \xrightarrow{m=n\wedge M,\lambda} P'}$$

$$\text{(S-STR)}\quad \frac{P\equiv Q \quad Q \xrightarrow{M,\lambda} Q' \quad Q'\equiv P'}{P \xrightarrow{M,\lambda} P'} \qquad \text{(S-ELSE)}\quad \frac{Q \xrightarrow{M,\lambda} Q' \quad m\neq n\wedge M \text{ consistent}}{\texttt{if } m=n \texttt{ then } P \texttt{ else } Q \xrightarrow{m\neq n\wedge M,\lambda} Q'}$$

Table 4. Additional symbolic rules for processes

$$\text{(S-CHOICE-1)}\quad \frac{P \xrightarrow{M,\lambda} P' \quad \Box\in\{m,n\}}{\texttt{if } m=n \texttt{ then } P \texttt{ else } Q \xrightarrow{M,\lambda} P'} \qquad \text{(S-CHOICE-2)}\quad \frac{Q \xrightarrow{M,\lambda} Q' \quad \Box\in\{m,n\}}{\texttt{if } m=n \texttt{ then } P \texttt{ else } Q \xrightarrow{M,\lambda} Q'}$$

$$\text{(S-CHOICE-3)}\quad x_1(\widetilde{v_1}).P_1+\ldots+\Box(\widetilde{v_i}).P_i+\ldots+x_n(\widetilde{v_n}).P_n \xrightarrow{true,\tau} P_i$$

In addition, non-determinism is a valid abstraction only when either alternative is actually present in the concrete process, namely abstraction must reflect real choices. For instance, let P below be a concrete process and R be a process obtained from P by turning the conditional statement into a non-deterministic choice:

$$P' \equiv \texttt{if } a=a \texttt{ then } P_1 \texttt{ else } P_2 \quad R \equiv \texttt{if } \Box=\Box \texttt{ then } P_1 \texttt{ else } P_2$$

We expect R not to be an abstraction of P', since $a=a$ is always true and P' can never evolve to P_2. In fact, a suitable abstraction of P' is simply P_1.

6 Trace-Based Abstraction

We present a relation of abstraction based on *symbolic traces*. Roughly, for V a set of ports and data names that must be kept visible, a process P is an abstraction of a process Q with respect to V if the set of traces of P coincides with the set of concrete traces derived by the symbolic traces of Q "up to" the names not in V.

Definition 2 (symbolic traces). *A* symbolic trace *s is a sequence of symbolic actions $\lambda_1.\cdots.\lambda_n$. The set $STr(P)$ of symbolic traces of a process P is defined as follows:*

$$STr(P) = \{\, \langle M,s\rangle \mid P \xrightarrow{M_1,\lambda_1} P_1 \ldots P_{n-1} \xrightarrow{M_n,\lambda_n} P_n \text{ and } M=M_1\wedge\ldots\wedge M_n \text{ and } s=\lambda_1.\cdots.\lambda_n \text{ and } bn(\lambda_i)\cap bn(\lambda_j)=\emptyset \text{ for all } i \text{ and } j \text{ s.t. } i\neq j \,\}$$

For $s=\lambda_1.\cdots.\lambda_n$ and σ such that every data variable in s belongs to $dom(\sigma)$, $s\sigma$ stands for $\lambda_1\sigma.\cdots.\lambda_n\sigma$. For instance, if $s=x(u).z(v).\overline{y}\langle v\rangle$ and $\sigma=\{a/u,b/v\}$, $s\sigma = x\langle a\rangle.z\langle b\rangle.\overline{y}\langle b\rangle$. Given a process, we can recover its concrete traces by instantiating its symbolic traces, as stated by the following definition.

Definition 3 (derived concrete traces). *The set $DCTr(P)$ of derived concrete traces of a process P is defined as follows:*

$$DCTr(P) = \{s\sigma \mid \langle M, s\rangle \in STr(P) \text{ and } \sigma \models M \text{ and } s\sigma \text{ is a trace}\}.$$

Consider the process $P \equiv x(v).\texttt{if } v = a \texttt{ then } \overline{y}\langle v\rangle \texttt{ else } \overline{z}\langle v\rangle$ shown in Ex. 1. The sets of symbolic traces and of derived concrete traces of P are as follows.

$$STr(P) = \{\langle true, x(v)\rangle, \langle v = a, x(v).\overline{y}\langle v\rangle\rangle, \langle v \neq a, x(v).\overline{z}\langle v\rangle\rangle\}$$
$$DCTr(P) = \{x\langle a\rangle, x\langle b\rangle, \cdots, x\langle a\rangle.\overline{y}\langle a\rangle, x(b).\overline{z}\langle b\rangle, x(c).\overline{z}\langle c\rangle, \ldots\}.$$

Note that $DCTr(P)$ is equal to the set $Tr(P)$ of traces of P (shown in Ex. 1). The following proposition formally states the equivalence of these two alternative characterizations of the concrete traces of a process.

Proposition 3. *Let P be a closed process. The sets $Tr(P)$ and $DCTr(P)$ coincide.*

As stated before, an abstract process hides names used by the concrete process. The following definitions describe the effect of hiding names in conditions, actions, and symbolic traces. Hereafter, V stands for a set of names that are kept visible.

We write $M_{|V}$ for the abstraction of a condition M with respect to a set V. The effect of the abstraction $M_{|V}$ is defined inductively as expected, once it is set that:

$$(m = n)_{|V} = true \text{ if } \{m,n\} \setminus V \neq \emptyset \qquad (m \neq n)_{|V} = true \text{ if } \{m,n\} \setminus V \neq \emptyset$$

Note that the operator $_{|V}$ makes a condition weaker, since all constraints involving hidden names are removed. For instance, given the condition $M \equiv (v = a \wedge v = w \wedge u \neq b)$, the abstraction of M when hiding a is $M_{|\{v,w,u,b\}} \equiv (true \wedge v = w \wedge u \neq b)$.

The abstraction of a symbolic action λ with respect to a set V, written $\lambda_{|V}$, is defined by the following expression, with $\{\Box/m_1, \ldots, \Box/m_n\}$ a partial map from $\mathcal{V} \cup \mathcal{C}$ to $\{\Box\}$.

$$\lambda_{|V} = \begin{cases} \lambda\{\Box/m_1, \ldots, \Box/m_n\} & \text{if } subj(\lambda) \in V \text{ and } m_i \in (obj(\lambda) \setminus V), \text{ for } i = 1, \ldots, n \\ \tau & \text{if } subj(\lambda) \notin V \text{ or } \lambda = \tau \end{cases}$$

Abstraction on actions is naturally extended to sequences of symbolic actions as below.

$$s_{|V} = \begin{cases} \lambda_{|V} & \text{if } s = \lambda \\ \tau s'_{|V \setminus \widetilde{u}} & \text{if } s = x(\widetilde{u})\, s' \text{ and } x \notin V \\ x(\widetilde{u})\, s'_{|V \cup \widetilde{u}} & \text{if } s = x(\widetilde{u})\, s' \text{ and } x \in V \\ \lambda_{|V}\, s'_{|V} & \text{if } s = \lambda s' \text{ and } \lambda \neq x(\widetilde{u}) \end{cases}$$

Note that any input action over a hidden channel (second line in the above definition) is mapped to a silent action and all received names are considered hidden when abstracting the remaining part of the trace. Differently, when abstracting an input action over a visible name (third line in the above definition) all received names are considered visible for the rest of the trace. For instance, when considering the trace $s = x(u).y(v).\overline{z}\langle u, v\rangle$ and the set $V = \{x, z\}$ of visible names, the abstraction of s when considering V is

$s_{|V} = x(u).\tau.\overline{z}\langle u, \Box\rangle$. The set of abstract symbolic traces of P is obtained as the abstraction of any symbolic trace of P:

$$STr(P)_{|V} = \{\langle M_{|V}, s_{|V}\rangle \mid \langle M, s\rangle \in STr(P)\}$$

Since $STr(P)_{|V}$ is a set of symbolic traces, we can define the set of the associated concrete traces. We call this set the abstraction of the derived concrete traces of a process P with respect to a set V, written $DCTr(P)_{|V}$ and defined as follows.

$$DCTr(P)_{|V} = \{s\sigma \mid \langle M, s\rangle \in STr(P)_{|V} \text{ and } \sigma \models M \text{ and } s\sigma \text{ is a trace}\}.$$

Consider the process $P = x(v).\texttt{if } v = a \texttt{ then } \overline{y}\langle v\rangle \texttt{ else } \overline{z}\langle v\rangle$ introduced in Ex. 1. The abstraction of $STr(P)$ when hiding a and the corresponding derived concrete traces are:

$$\begin{aligned} STr(P)_{|\{x,y,z\}} &= \{\langle true, x(v)\rangle, \langle true, x(v).\overline{y}\langle v\rangle\rangle, \langle true, x(v).\overline{z}\langle v\rangle\rangle\} \\ DCTr(P)_{|\{x,y,z\}} &= \{x\langle a\rangle, x\langle b\rangle, \cdots, x\langle a\rangle.\overline{y}\langle a\rangle, x\langle a\rangle.\overline{z}\langle a\rangle, x(b).\overline{y}\langle b\rangle, x(b).\overline{z}\langle b\rangle, \ldots\}. \end{aligned}$$

Note that the set $DCTr(P)_{|\{x,y,z\}}$ coincides with the set of concrete traces of $Q \equiv x(v).\texttt{if } v = \Box \texttt{ then } \overline{y}\langle v\rangle \texttt{ else } \overline{z}\langle v\rangle$ shown in Ex. 1, which is a suitable abstraction of P when hiding a. Below we formally define the notion of trace abstraction.

Definition 4. *A closed process Q is a* trace abstraction *of a closed process P with respect to a set $V \subseteq \mathcal{N}$ such that $fn(Q) \subseteq V$, written $Q \ltimes^V P$, if $Tr(Q) = DCTr(P)_{|V}$.*

As mentioned before, the equation $Tr(Q) = DCTr(P)_{|V}$ holds for P and Q as defined in Ex. 1 and for $V = \{x, y, z\}$. Therefore, $Q \ltimes^V P$ for $V = \{x, y, z\}$.

Remark 1. The abstraction condition cannot be obtained directly by abstracting concrete traces, i.e., condition $Tr(Q) = DCTr(P)_{|V}$ is different from requiring either $Tr(Q) = Tr(P)_{|V}$ or $Tr(Q)_{|V} = Tr(P)_{|V}$, where $Tr(P)_{|V}$ stands for the set obtained by abstracting every trace in $Tr(P)$ with respect to V. For instance, when considering the processes P and Q of Ex. 1, their sets of concrete traces are as follows

$$\begin{aligned} Tr(P) &= \{x\langle a\rangle, x\langle b\rangle, \cdots, x\langle a\rangle.y\langle a\rangle, x\langle b\rangle.z\langle b\rangle, x\langle c\rangle.z\langle c\rangle, \ldots\} \\ Tr(Q) &= \{x\langle a\rangle, x\langle b\rangle, \cdots, x\langle a\rangle.y\langle a\rangle, x\langle a\rangle.z\langle a\rangle, x\langle b\rangle.y\langle b\rangle, x\langle b\rangle.z\langle b\rangle, \ldots\} \end{aligned}$$

and their direct abstractions as below.

$$\begin{aligned} Tr(P)_{|V} &= \{x\langle\Box\rangle, x\langle b\rangle, \cdots, x\langle\Box\rangle.y\langle\Box\rangle, x\langle b\rangle.z\langle b\rangle, x\langle c\rangle.z\langle c\rangle, \ldots\} \\ Tr(Q) \quad &= \{x\langle\Box\rangle, x\langle b\rangle, \cdots, x\langle\Box\rangle.y\langle\Box\rangle, x\langle\Box\rangle.z\langle\Box\rangle, x\langle b\rangle.y\langle b\rangle, x\langle b\rangle.z\langle b\rangle, \ldots\} \end{aligned}$$

Checking the abstraction relation introduced before is hard, since it requires to compare infinite sets of traces. Because of this, we provide an alternative characterization of trace abstraction that requires to consider finitely-many symbolic traces.

We start by introducing some auxiliary notions that will allow us to compare sets of symbolic traces. Consider the sets $S_1 = \{\langle true, s\rangle\}$ and $S_2 = \{\langle u = a, s\rangle, \langle u \neq a, s\rangle\}$. They describe the same behavior since, for any substitution $\sigma \models true$, either $\sigma \models u = a$ or $\sigma \models u \neq a$. Hence, the set of concrete traces derived from S_1 and S_2 coincide. The following definition formally states when two sets of symbolic traces describe the same behavior.

Definition 5. *Let S_1 and S_2 be sets of pairs of conditions and symbolic traces. We write $S_1 \sqsubseteq S_2$ iff for all $\langle M,s\rangle$ in S_1 there exists an M-decomposition D such that for all N in D there exists $\langle N',s'\rangle$ in S_2 such that $N \Rightarrow N'\sigma_\alpha$ and $s = s'\sigma_\alpha$ for some renaming σ_α of the bound names of s'. We write $S_1 \stackrel{\star}{=} S_2$ when both $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$ hold.*

We define below an alternative (and more efficient) characterization of abstraction in terms of symbolic traces.

Definition 6. *(trace abstraction) A process P is a* symbolic trace abstraction *of a process Q with respect to a set $V \subseteq \mathcal{N}$ s.t. $fn(P) \subseteq V$, written $P \ltimes_s^V Q$, if $STr(P) \stackrel{\star}{=} STr(Q)_{|V}$.*

We remark that the above definition extends the definition of abstraction over processes that are not necessarily closed. The following proposition ensures that symbolic trace abstraction coincides with trace abstraction when restricting to closed processes.

Proposition 4. *Let P and Q be two closed processes. $P \ltimes_s^V Q$ iff $P \ltimes^V Q$.*

7 Abstraction as a Generalized Symbolic Bisimulation

A main challenge of defining a simulation-based abstraction relation is that the application of substitutions when executing concrete processes makes the evaluation of branching statements deterministic while such statements should match non-deterministic choices. As a solution, we propose an abstraction based on a generalization of symbolic bisimulation [10,3]. Symbolic bisimulation is defined on top of a symbolic transition system. Informally, to verify whether two processes P and Q are bisimilar with respect to a given Boolean condition M it is required to find, for each symbolic move of P labeled with $\langle N,\lambda\rangle$, a partition of $N \wedge M$ such that each subcase entails a corresponding symbolic move of Q, and vice-versa for Q and P. First, we give an auxiliary definition that will be used in the subsequent characterization of abstraction.

Definition 7 (visible names). *Given a set of visible names V and a symbolic action λ, the set of visible received names of λ, written $vn(\lambda)_V$, is defined as follows:*

$$vn(\lambda)_V \stackrel{\text{def}}{=} \begin{cases} \widetilde{u} \text{ if } \lambda = x(\widetilde{u}) \text{ and } x \in V \\ \emptyset \text{ otherwise} \end{cases}$$

We will omit the subscript V when it is clear from the context.

Definition 8 (simulation-based abstraction). *The family $\mathcal{R} = \{\mathcal{R}_M^V\}_M$ of process relations is a* family of simulation-based abstraction relations*, indexed over the set of conditions M, iff for all M and $P\mathcal{R}_M^V Q$:*

1. *If $Q \xrightarrow{N,\lambda} Q'$ and $bn(\lambda) \cap fn(P,Q,M) = \emptyset$ then there exists a $M \wedge N$-decomposition D s.t. $\forall M' \in D$ there exists $P \xrightarrow{N',\lambda'} P'$, with $M' \Rightarrow N' \wedge \lambda_{|V} = \lambda'$ and $P'\mathcal{R}_{M'}^{V \cup vn(\lambda)} Q'$.*
2. *if $P \xrightarrow{N,\lambda} P'$ and $bn(\lambda) \cap fn(P,Q,M) = \emptyset$ then there exists a $M \wedge N$-decomposition D s.t. $\forall M' \in D$ there exists $Q \xrightarrow{N',\lambda'} Q'$ with $M' \Rightarrow N'_{|V} \wedge \lambda = \lambda'_{|V}$ and $P'\mathcal{R}_{M'}^{V \cup vn(\lambda')} Q'$.*

A process P is a simulation-based abstraction *of a process Q with respect to a set $V \subseteq \mathcal{N}$, written $P \propto^V Q$, if there is an abstraction relation $\mathcal{R}^V_{true}$ s.t. $P\,\mathcal{R}^V_{true}\,Q$, with $fn(P) \subseteq V$.*

Condition 1 above states that the abstraction P simulates the concrete process Q up to hidden names. Note that we require $\lambda_{|V} = \lambda'$ instead of the standard definition of symbolic bisimulation that imposes the exact matching of action labels. Condition 2 states that the (concrete) process Q can simulate its abstraction P if we forget about the constraints involving hidden values. That is, if P proposes a move with label $\langle N, \lambda \rangle$ we allow Q to mimic the behavior for a more restrictive condition N'. (Actually, N' may contain several additional constraints involving hidden names.) Note that this makes the abstraction relation not symmetric. For instance, consider the two processes below:

$$P \equiv \texttt{if } v = \square \texttt{ then } \overline{y}\langle v \rangle \texttt{ else } \overline{z}\langle v \rangle \quad Q \equiv \texttt{if } v = a \texttt{ then } \overline{y}\langle v \rangle \texttt{ else } \overline{z}\langle v \rangle.$$

It holds that $P \propto^V Q$ for $V = \{v, y, z\}$. Indeed, when considering the transition $P \xrightarrow{true, \overline{y}\langle v \rangle} 0$, we can take $Q \xrightarrow{v=a, \overline{y}\langle v \rangle} 0$ since $true \Rightarrow (v = a)_{|V} \wedge \overline{y}\langle v \rangle = \overline{y}\langle v \rangle_{|V}$. Conversely, $P \not\propto^V Q'$ for $Q' \equiv \texttt{if } a = a \texttt{ then } \overline{y}\langle v \rangle \texttt{ else } \overline{y}\langle v \rangle$ because $P \xrightarrow{true, \overline{z}\langle v \rangle} 0$ but $Q' \not\xrightarrow{M, \overline{z}\langle v \rangle}$.

We remark that the relation $\propto$ is a simulation (since the abstract process simulates the concrete one) but, in general, is not either a bisimulation or a similarity.

Remark 2. The abstraction relation generalizes symbolic early bisimulation [3,10]. If we restrict to concrete processes, i.e., all names are visible (hence $V = \mathcal{N}$), then $\propto^{\mathcal{N}} = \approx_e$. Indeed, the abstraction operator $\_{|V}$ is the identity when $V = \mathcal{N}$.

The following result states that simulation-based abstraction is finer than trace-based abstraction.

Theorem 1. $\propto^V \subset \ltimes^V$.

8 Composition of Orchestrators

This section addresses the problem of composing orchestrators, and the properties that are ensured by the abstraction relation. Basically, an orchestrator is a concrete process P plus the declaration of the operations it provides, which is a set $I \subseteq \mathcal{X}$ of channel names, and a declaration of the operations it invokes, which is a set $O \subseteq \mathcal{X}$.

Definition 9 (business processes). *The set of business processes B is defined by the following grammar:*

$$B ::= (I, O)P \mid B \| B$$

We usually abbreviate $B = (I_1, O_1)P_1 \| \cdots \| (I_n, O_n)P_n$ with $B = \|_{i \leq n}(I_i, O_i)P_i$. We say that a business process $B = \|_{i \leq n}(I_i, O_i)P_i$ is *well-formed* iff the three conditions below hold:

1. For all i, if $x \in fn(P_i)$ and x occurs as subject of an input prefix of P_i, then $x \in I_i$. Similarly, if $x \in fn(P_i)$ and x occurs as subject of an output prefix of P_i then $x \in O_i$.
2. $I_i \cap I_j = \emptyset$ for all $i \neq j$.
3. For all i, $I_i \cap O_i = \emptyset$

Table 5. LTS for business processes

$$\text{(B-TAU)}\ \frac{P \xrightarrow{M,\tau} P'}{(I,O)P \xrightarrow{M,\tau} (I,O)P'} \qquad \text{(B-OUT)}\ \frac{P \xrightarrow{M,\overline{x}\langle\widetilde{m}\rangle} P' \quad x \in O}{(I,O)P \xrightarrow{M,\overline{x}\langle\widetilde{m}\rangle} (I,O)P'}$$

$$\text{(B-IN)}\ \frac{P \xrightarrow{M,x(\widetilde{v})} P' \quad x \in I}{(I,O)P \xrightarrow{M,x(\widetilde{v})} (I,O)P'} \qquad \text{(B-COMM)}\ \frac{B_1 \xrightarrow{M,\overline{x}\langle\widetilde{m}\rangle} B_1' \quad B_2 \xrightarrow{N,x(\widetilde{v})} B_2' \quad M \wedge N \text{ consistent}}{B_1 \| B_2 \xrightarrow{M\wedge N,\tau} B_1' \| B_2'\{\widetilde{m}/\widetilde{v}\}}$$

$$\text{(B-PAR)}\ \frac{B_1 \xrightarrow{M,\lambda} B_1' \quad bn(\lambda) \cap fn(B_2) = \emptyset}{B_1 \| B_2 \xrightarrow{M,\lambda} B_1' \| B_2} \qquad \text{(B-STR)}\ \frac{B \equiv C \quad C \xrightarrow{M,\lambda} C' \quad C' \equiv B'}{B \xrightarrow{M,\lambda} B'}$$

The first condition requires every orchestrator P_i to correctly declare the operations it provides and the operations it invokes. The second condition imposes operations provided by different orchestrators to be named differently. Last condition forbids self-communications in orchestrators. Hereafter, we will assume all business processes to be well-formed. The operational semantics of business processes is defined up-to the structural congruence $\equiv$ over business processes, which is the least congruence over business processes closed with respect to the commutative and associative laws for $\|$ and the structural rules for concrete processes.

Definition 10. *The symbolic labeled transition relation $\xrightarrow{M,\lambda}$ over business processes is the least relation satisfying the inference rules in Table 5.*

Rules (B-TAU), (B-IN), (B-OUT) lift silent, input and output actions performed by the process P to corresponding actions of the business process $(I,O)P$. The other rules are standard.

8.1 Composition Compliance and Abstraction

We now study the notion of compliance among orchestrators and its relation with abstraction. Different notions of compliance have been proposed in the literature (notably weak termination in the context of Workflow Nets [14]). We adopt here the proposal of [14], which requires both the client and the server to complete in every possible interaction. The following definition introduces the notion of compliance up-to a set of visible names.

Definition 11 (business process compliance up-to V). *We say that two business processes $B_1 = (I_1,O_1)P$ and $B_2 = (I_2,O_2)Q$ are compliant with respect to a condition M and a set of visible names V s.t. $I_1 \cap O_2 \subseteq V$, $O_1 \cap I_2 \subseteq V$, written $B_1 \bowtie_M^V B_2$, whenever $B_1 \| B_2$ is well-formed and either $P \equiv Q \equiv 0$ or all the following conditions hold*

- *if $B_1 \xrightarrow{N,\overline{x}\langle\widetilde{m}\rangle} B_1'$ and $x \in V$ and $M \wedge N$ consistent, then $B_2 \xrightarrow{N',x(\widetilde{v})} B_2'$ and $M \wedge N \wedge N'$ consistent and $B_1' \bowtie_{M\wedge N\wedge N'}^V B_2'\{\widetilde{m}/\widetilde{v}\}$*
- *if $B_1 \xrightarrow{N,x(\widetilde{v})} B_1'$ and $x \in V$ and $M \wedge N$ consistent, then $B_2 \xrightarrow{N',\overline{x}\langle\widetilde{m}\rangle} B_2'$ and $M \wedge N \wedge N'$ consistent and $B_1'\{\widetilde{m}/\widetilde{v}\} \bowtie_{M\wedge N\wedge N'}^V B_2'$*

- *if* $B_1 \xrightarrow{N,\lambda} B_1'$ *and* ($\lambda = \tau$ *or* $subj(\lambda) \notin V$) *and* $M \wedge N$ *consistent, then* $B_1' \bowtie^V_{M \wedge N} B_2$
- *if* $B_2 \xrightarrow{N,\lambda} B_2'$ *and* ($\lambda = \tau$ *or* $subj(\lambda) \notin V$) *and* $M \wedge N$ *consistent, then* $B_1 \bowtie^V_{M \wedge N} B_2'$

We write $B_1 \bowtie^V B_2$ *to denote* $B_1 \bowtie^V_{true} B_2$, *and* $B_1 \bowtie B_2$ *for* $B_1 \bowtie^{\mathcal{N}} B_2$.

Note that above relation requires V to include all channels through which B_1 and B_2 may synchronize. Then, the notion of compliance up-to V ensures that the interaction among two business processes B_1 and B_2 completes provided with the fact that any other action involving the synchronization of either B_1 or B_2 with a third party will take place at the right moment. Furthermore, remark that the above relation is asymmetric. For instance, for $B_1 = (\emptyset, \{x\})\overline{x}\langle c\rangle.\overline{x}\langle a\rangle$ and $B_2 = (\{x\}, \emptyset)x(v).\texttt{if}\ v = \Box\ \texttt{then}\ x(z)\ \texttt{else}\ y(z)$, it holds $B_1 \bowtie B_2$ but not the converse $B_2 \bowtie B_1$.

Next result ensures "safe replacement", i.e. that substituting an abstract process P with a more concrete one R, i.e. $P \propto^V R$ for some set of visible names V, we still obtain a compliant composition if we ignore the interactions that take place over channels that are not in the abstraction, namely that are not in V.

Theorem 2. *If* $P_1 \propto^V Q$ *and* $(I_1, O_1)P_1 \bowtie (I_2, O_2)P_2$, *then* $(I, O)Q \bowtie^V (I_2, O_2)P_2$.

9 Future Work

In this paper we have studied a notion of abstraction for orchestration languages. It would be interesting to extend our approach by including recursion. Although being less expressive, several models of orchestration consider finite fragments of process calculi because they allow for the description of usual scenarios: most instances of business interactions are finite in practice. Nevertheless, our main contribution, i.e., the definition of abstraction as symbolic bisimulation and the composition result extend to a recursive form of process like $\texttt{rec}\ K\ \texttt{in}\ P$ with the usual operational semantics: (i) the definition of abstraction remains unchanged, since it is given in terms of transitions independently from the form of the process, (ii) abstractions will preserve all non terminating computations of the concrete process, because the abstraction relation requires the abstract process to exhibit "at least the same computations as the concrete processes", hence (iii) the substitution of an abstraction by a concrete process in a compliant composition will preserve termination. Extending the results presented in §6 would be more involved since we would lose the property of having finite sets of finite symbolic traces.

We also plan a formal study of the relationship with session types and contracts. As remarked in the introduction, even if these approaches have similarities with ours, a precise comparison seems not to be immediate.

References

1. Alur, R., Henzinger, T., Kupferman, O., Vardi, M.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
2. Arun-Kumar, S., Natarajan, V.: Conformance: A precongruence close to bisimilarity. In: STRICT. Springer Workshops in Computer Series, pp. 148–165. Springer, Heidelberg (1995)

3. Boreale, M., De Nicola, R.: A symbolic semantics for the pi-calculus. Information and Computation 126(1), 34–52 (1996)
4. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
5. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: Necula, G.C., Wadler, P. (eds.) POPL 2008, pp. 261–272. ACM, New York (2008)
6. de Alfaro, L., Henzinger, T.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
7. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
8. Fournet, C., Hoare, C.A.R., Rajamani, S., Rehof, J.: Stuck-free conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004)
9. Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica 42(2), 191–225 (2005)
10. Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science 138, 353–389 (1995)
11. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
12. Laneve, C., Padovani, L.: The must preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
13. Laneve, C., Padovani, L.: The pairing of contracts and session types. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 681–700. Springer, Heidelberg (2008)
14. Massuthe, P., Schmidt, K.: Operating guidelines - an automata-theoretic foundation for the service-oriented architecture. In: QSIC, pp. 452–457. IEEE Computer Society, Los Alamitos (2005)
15. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
16. Milner, R.: Communication and Concurrency. Prentice Hall International, Englewood Cliffs (1989)