Goal-oriented Composition of Services

Sebastian Nanz and Terkel K. Tolstrup

Informatics and Mathematical Modelling
Technical University of Denmark
{nanz,tkt}@imm.dtu.dk

Abstract. One fundamental issue in service-oriented computing concerns the question whether services can be composed in a manner that allows them to achieve their individual goals. In this paper we use a variant of interface automata as an abstraction of the input/output behaviour of services, which are themselves represented as terms in the π-calculus extended with an action for expressing service collaboration. In this setting, the question whether two or more services can meaningfully compose is then reduced to checking a simple property of the product automaton of the involved interfaces.

1 Introduction

Service-oriented computing [17] has evolved from component-based software development as an effective approach to building distributed applications. A service can be described as a process that can be addressed and used by other services on a network, based on its published interface that identifies the capability it provides. Two of the main research problems arising from this approach are thus concerned with the design of this interface: in order for it to enable the *discovery* of services that may achieve a certain computational task; and, to facilitate the *composition* of services.

Web Services [1] are the example of the services paradigm that is currently developed furthest. Here, the static interface of a service can be described in the *Web Services Description Language (WSDL)* [4], an XML-based format which contains a definition of the messages and ports that are involved in a communication with a service. This description limits severely the behavioural complexity the service can implement, since interactions can only be described with a limited variety of message exchange patterns. In an advanced service model, services could offer complex functionality that likewise may result in more complex interactions with other services. For such a model to be successful, it has to be supported by interface descriptions that allow to check the compatibility of services with respect to their communication behaviour, e.g. to ensure that services do not deadlock waiting for each other's input.

In this paper we suggest a variant of interface automata [5] as a means to provide this interface information. Transitions in an automaton are labelled with the types of ports and message formats or internal actions, and describe the input/output behaviour of the service. We view these automata as abstractions

$P ::=$	0	nil process	$\pi ::=$	$x\tilde{y}$	reception
	$\sum_{i\in I} \pi_i.P_i$	guarded sum		$\overline{x}\tilde{y}$	sending
	$P_1 \mid P_2$	parallel composition		$\mathsf{com}\, x$	collaboration initiation
	$(\mathsf{new}\ x)\ P$	restriction		τ	unobservable action
	$!P$	process replication			

Table 1. Syntax of the polyadic π-calculus with service collaborations

of processes which implement the actual service. Such processes are written in a variant of the polyadic π-calculus [15] that enriches the usual syntax with an action to express the initiation of a service collaboration. We define a type system to statically describe the conformance of the abstract interface with its implementing process. We prove for typed processes whose interface automata compose *optimistically* [5] (i.e. there is *some* sequence of interactions to lead to a final state) that the process composition also evolves in a way that the goals of the collaboration can be achieved.

The remainder of this paper is structured as follows. In Section 2 we describe our extension of the π-calculus syntax and semantics that allows us to describe service collaborations, and we introduce the running example of the paper. Section 3 presents interface automata and their use as process abstractions. Also, a non-standard semantics is introduced which describes the execution of interface-conformant processes. In Section 4 we complement this dynamic view of conformance with static type checking, and in Section 5 we discuss goal-oriented compositions of interface automata. We present related work in Section 6 and conclude in Section 7.

2 Modelling of Service Composition

The π-calculus [15] is a fundamental process algebraic approach to describing concurrent systems whose configuration may change during the computation. We use the π-calculus in its polyadic form as a basis for the description of services with complex interactive behaviour, adding an action which explicitly describes the agreement of two processes to collaborate.

2.1 A Process Model for Services

The polyadic π-calculus models two entities: *processes* and *names*. Processes interact by synchronising on channels where they exchange a sequence of data values; both channels and data are uniformly described by names. Names are (unstructured) values drawn from the infinite set $\mathcal{N}$.

The syntax of processes is shown in Table 1 and its entities can be informally described as follows. The terminated process is represented by 0. The

$$
\begin{array}{rr}
P \mid 0 \equiv P & (\mathsf{new}\ x)\ (P \mid Q) \equiv P \mid (\mathsf{new}\ x)\ Q \text{ if } x \notin \mathit{fn}(P) \\
P \mid Q \equiv Q \mid P & (\mathsf{new}\ x)\ 0 \equiv 0 \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (\mathsf{new}\ x)\ (\mathsf{new}\ y)\ P \equiv (\mathsf{new}\ y)\ (\mathsf{new}\ x)\ P \\
!P \equiv !P \mid P &
\end{array}
$$

Table 2. Structural congruence of the π-calculus

term $\sum_{i \in I} \pi_i.P_i$ models an external choice, that allows one action π_i to be executed and to continue with P_i. In our syntax variant, there are the four types of actions; we describe first the three classical ones: an input process $x\tilde{y}.P$ receives a sequence of names along channel x and substitutes it for $\tilde{y}$ in P; an output process $\overline{x}\tilde{y}.P$ sends $\tilde{y}$ along channel x; and an internal action τ executes without interaction.

As an extension of the standard syntax, we add an action $\mathsf{com}\, x$ which describes the readiness of a process to compose with some other process on channel x; in the term $\mathsf{com}\, x.P$, the action $\mathsf{com}\, x$ binds x in P. Two processes running in parallel $\mathsf{com}\, x.P \mid \mathsf{com}\, y.Q$, which are both ready to compose, can then evolve to $(\mathsf{new}\ z)\ (P[z/x] \mid Q[z/y])$, i.e. collaborate using a fresh name z as a common channel. The intuition is that executing a com-action describes a handshake taking place between two processes. While this could be expressed also with standard syntax, having an explicit syntax element available enables us to identify the starting points of a collaboration, which is important for our analysis.

Processes are composed in parallel by $P_1 \mid P_2$, and replication $!P$ represents an infinite number of copies of P. The expression $(\mathsf{new}\ x)\ P$ creates a new name with scope P.

As in the original π-calculus, we present a formal semantics based on a structural congruence and a reaction relation. The structural congruence is the least equivalence relation that is generated by the rules in Table 2, and is standard.

Also the reaction rules of Table 3 are standard, with the exception of the rule for composition which implements the semantics explained informally above. In addition, we introduce a typing environment Γ that maps names into a finite set of *channel types*, and require that collaborations can only take place if both collaboration partners expect the same channel type. We omit to associate Γ explicitly with the reaction rules, as Γ is defined globally.

2.2 Example: Web Auctions

In order to illustrate our π-calculus extension as a formalism to describe service interactions, we present an example from the area of web-based auctions.

$$
\begin{array}{l}
\mathit{Buyer} \triangleq\ !\mathsf{com}\, \mathit{buyer}. \\
\qquad (\mathsf{new}\ \mathit{bid})\ (\mathsf{new}\ \mathit{item})\ \overline{\mathit{buyer}}(\mathit{bid}, \mathit{item}, \mathit{buyer}). \\
\qquad\quad (\mathit{buyer}(\mathit{lost}, \mathit{item}).0 + \mathit{buyer}(\mathit{won}, \mathit{item}, \mathit{product}).0)
\end{array}
$$

$$\tau.P + M \to P \qquad \frac{|\tilde{y}| = |\tilde{z}|}{(x\tilde{y}.P + M) \mid (\overline{x}\tilde{z}.Q + N) \to P[\tilde{z}/\tilde{y}] \mid Q}$$

$$\frac{\Gamma(x) = \Gamma(y) \qquad z \text{ fresh}}{(\mathsf{com}\, x.P + M) \mid (\mathsf{com}\, y.Q + N) \to (\mathsf{new}\ z)\ (P[z/x] \mid Q[z/y])}$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'} \qquad \frac{P \to P'}{(\mathsf{new}\ x)\ P \to (\mathsf{new}\ x)\ P'}$$

Table 3. Reaction rules of the π-calculus with service collaborations

A buyer can place a bid on a certain item, using a channel *buyer* that will be determined by a new collaboration. We imagine that the buyer might both try to directly collaborate with the web auction or through an auction agent, both of which are services as well. The buyer immediately places his maximum bid. As the result of the auction, the buyer expects to receive either the message to have lost the auction, or to have won, and then get the product.

$$\begin{aligned} \mathit{Auction} \triangleq\ & !\mathsf{com}\, \mathit{auction}. \\ & \mathit{auction}(\mathit{bid}, \mathit{item}, \mathit{buyer}). \\ & \quad !(\overline{\mathit{auction}}(\mathit{higherbid}, \mathit{item}). \\ & \qquad (\mathit{auction}(\mathit{bid}, \mathit{item}, \mathit{buyer}).0 + \tau.0) \\ & \quad + \\ & \quad \overline{\mathit{auction}}(\mathit{won}, \mathit{item}, \mathit{product}).0) \end{aligned}$$

The auction service waits for bids on items it receives via the *auction* channel which is also determined by a new collaboration. On this channel, it can announce two outcomes. First, it can send a message to the bidder that it has received a higher bid than the current bid. It will then give the bidder the possibility to raise its bid by sending another bid message on the same channel, or it terminates after executing τ, meaning that the auction is over and lost by the bidder. Second, it can output the message that the bidder has won the item, and send along the product.

Note that throughout the examples we resolve + with external choice, which would be problematic for example in the case of a buyer missing the $\mathit{buyer}(\mathit{lost}, \mathit{item}).0$ branch: the buyer could then always force the winning branch of the auction. In this case we could however modify our examples by using internal choice, i.e. by prefixing the actions in sums with τ.

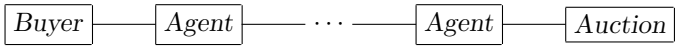
Finally, the auction agent can collaborate with two other services in order to bid on behalf of a service in a certain auction. With its first collaborator, it expects a bidding instruction via the *client* channel. On reception it will start a second collaboration on the *bidding* channel, in order to pass on this bidding instruction. The agent then expects either to be outbid and receive the *higherbid* message, or to win the item and receive the product. In the first case, it will decide according to its bidding algorithm either to place another bid (and

restart, using replication), or communicate on the *client* channel that this item is considered lost. In the second case, it will send the product on to the service that placed the original bidding instruction.

$$
\begin{array}{l}
Agent \triangleq\ !\mathsf{com}\, client. \\
\qquad client(bid, item, client). \\
\qquad\quad \mathsf{com}\, bidding. \\
\qquad\qquad (\mathsf{new}\ bid')\ \overline{bidding}(bid', item, client). \\
\qquad\qquad\quad !(bidding(higherbid, item). \\
\qquad\qquad\qquad (\overline{client}(higherbid, item).0 \\
\qquad\qquad\qquad + \\
\qquad\qquad\qquad (\mathsf{new}\ bid')\ \overline{bidding}(bid', item, client).0) \\
\qquad\qquad\quad + \\
\qquad\qquad\quad bidding(won, item, product). \\
\qquad\qquad\qquad \overline{client}(won, item, product).0)
\end{array}
$$

In terms of composition, we may confirm by inspection that it should be possible for *Buyer* to use *Agent* in order to bid conveniently on a certain item offered by *Auction*. However, it is hard to establish this compatibility automatically when working directly with process descriptions. We therefore propose in the following section interface automata as a process abstraction, that enables the automatic inference of this result.

Furthermore, if we assume that all collaboration channels exhibit the same channel type, i.e. $\Gamma(buyer) = \Gamma(auction) = \Gamma(client) = \Gamma(bidding)$, collaborations turn out to be completely promiscuous: in addition to the already described collaborations, *Buyer* might potentially bid directly using the *Auction* service, or – as depicted below – auction agents might rely on other auction agents (maybe with advanced algorithms for specific types of auctions) in order to do the bidding.



As a matter of fact, entities may also try to compose with themselves, for example *Buyer* with *Buyer*. Not all these compositions would lead to successful computations (clearly, *Buyer* with *Buyer* would not), but using our approach, we will be able to select the meaningful ones.

3 Automata-based Abstractions of Processes

Interface theory provides an approach to describing the interfaces of components, where each component is represented by its input and output behaviour, and interface composition is the key operator. One is usually interested to have two properties on interfaces, namely that a component conforms to its interface, and that composed components have compatible interfaces. In the following we shall use a variant of a popular interface theory, *interface automata* [5], as an abstraction that describes the behaviour of services.

3.1 Interface Automata

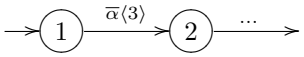
Interface automata [5] are finite state transition systems in a concurrent setting. An interface automaton describes a computational component by its input, output, collaboration, and internal actions. The automata synchronise on communications and are interleaved on internal actions.

Definition 1 (Interface Automaton). *An* interface automaton A *is a 5-tuple* $(S, \Sigma, \delta, s_0, F)$ *where* S *is a finite set of states,* $\Sigma = \Sigma^O \cup \Sigma^I \cup \Sigma^C \cup \{\tau\}$ *an alphabet with output actions* Σ^O*, input actions* Σ^I*, collaboration actions* Σ^C*, and the internal action* τ*,* $\delta : S \times \Sigma \to S$ *a transition function,* $s_0 \in S$ *an initial state, and* F *a set of final states.*

Note that interface automata are deterministic, and that we can thus use equations like $\delta(s, \sigma) = s'$ to describe the transition function; given an interface automaton A with transition function δ, we may then also write $A(s, \sigma) = s'$.

Interface automata distinguish themselves from I/O automata [13,14] by not being *input-enabled*, i.e. it is not required that the transition function δ is defined on all combinations of states and input actions. Instead one takes an *optimistic* approach by assuming that the environment never generates unmatched inputs. Interface automata come with a theory for interface conformance and the composition of components. In our variant approach, we redefine these terms in order to have interface automata serve as abstractions for π-processes, and to describe their goal-oriented composition. For this reason we have added in Definition 1 a notion of final states (that correspond to goals), which is not present in the original approach. In the following, we describe informally how processes are abstracted by automata, and elaborate on this using our running example; Section 3.2 establishes the formal connection.

The main idea of the abstraction is to have the actions of a process matched by transitions in the interface automata. The alphabet Σ of the interface automaton is defined and interpreted as follows: in the case of input and output actions, we take $\alpha\langle k\rangle$ and $\overline{\alpha}\langle k\rangle$, respectively, where α is the type and k the arity of the channel; in the case of collaboration actions, we take $\alpha\langle \mathsf{c}\rangle$; and for τ actions simply τ. For example, an output process $\overline{x}\tilde{y}.P$ with $\Gamma(x) = \alpha$ and $|\tilde{y}| = 3$ can be described by the following automaton, where the behaviour of P is described starting with state 2:



We use the meta-variable σ to range over the elements of Σ, and may sometimes write $\overline{\sigma}$ to denote $\overline{\alpha}\langle k\rangle$ when σ is given by $\alpha\langle k\rangle$.

We shall abstract each parallel process and each process in a sum by its own interface automaton. If a process splits into several processes, e.g. in the case of $\tau.(xy.P \mid uv.Q)$ or $\tau.(xy.P + uv.Q)$, we abstract this by having a branch for each

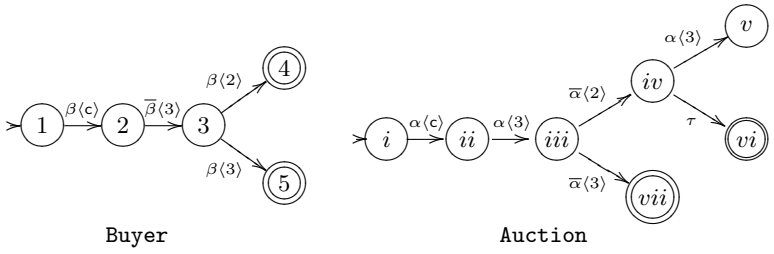
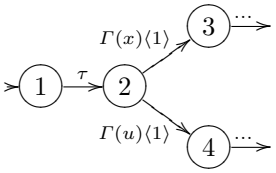


Fig. 1. Interface automata for the processes *Buyer* and *Auction*

process in the automaton:



An automaton abstracting process replication $!P$ is given by the automaton abstracting P. The introduction of new names is likewise ignored in the abstraction.

We take a state in the abstraction to be a final state if the corresponding process has reached a termination point 0, and if in addition the goal of the interaction has been reached. Such goals should match the identified *functional* goals [11] of the service. Final states are thus annotations that depend on the intended process semantics. We could however generate them automatically from a process by having an additional syntax element 0_g for terminated goal states that otherwise behaves like 0.

Example. We illustrate the abstraction of processes by interface automata by considering our running example. The buyer process is ready to collaborate in the start state and, after the handshake, accepts communication over the channel *buyer* on which it outputs a bid message (arity 2) and accepts input that is either win (arity 3) or lose (arity 2). Assuming β as type of the buyer channel, the abstracting automaton `Buyer` is given in Figure 1. Note that both the win and the lose situation determine goals of the buyer's collaboration, and are thus represented by final states. The process abstraction of *Auction* can be argued for similarly, where α is the type of the *auction* channel.

The interface automaton for *Agent* is depicted in Figure 2. The automaton communicates with the client, i.e. the *Buyer* or another *Agent*, on the channel *client* with type γ_1, and bids on an *Auction* (possibly through another *Agent*) using the channel *bidding* with type γ_2. The intuition behind these automata is that they can compose by connecting all channels of a given type, thus one way of composing the example automata would be connecting $\beta \leftrightarrow \alpha$, this correspond

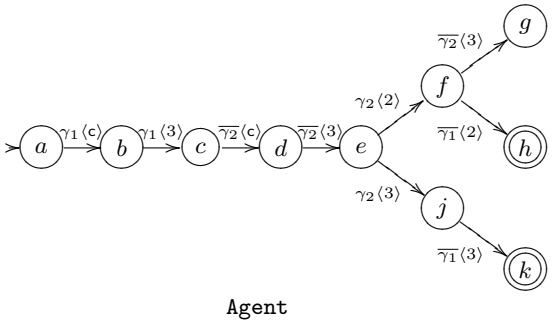


Fig. 2. Interface automaton for the process *Agent*

to connecting the *Buyer* and *Auction* directly. Another composition would result from connecting $\beta \leftrightarrow \gamma_1$ and $\gamma_2 \leftrightarrow \alpha$, here the *Buyer* is connected to the *Agent*, which in turn is connected to the *Auction*.

3.2 Interface Semantics

In order to formalise the conformance of an interface automaton with a process in our π-calculus variant, we propose a non-standard semantics, called *interface semantics*, that describes the behaviour of tagged processes. A *tagged process* $[P]_{A,s}$ relates the process behaviour P with the interface automaton A abstracting it, together with the state s the automaton is currently in. For example, we have argued earlier that the `Buyer` automaton of Figure 1 describes the behaviour of the process *Buyer* in Section 2.2. This means we can use the tagging $[Buyer]_{\texttt{Buyer},1}$. Once *Buyer* has agreed to a collaboration and sent its first bid, the tagging corresponds to

$$[buyer(lost, item).0 + buyer(won, item, product).0]_{\texttt{Buyer},3},$$

meaning that we are now in state 3 of the describing automaton.

The interface semantics checks this agreement of processes and their tags explicitly. We first present in Table 4 a structural congruence $\equiv_{\mathtt{t}}$ for tagged processes. The creation of new names is abstracted away by our automaton model and therefore the new name construct can leave the tagged process. We explicitly require a scope extrusion rule in the tagged semantics since it cannot be inferred from the standard congruence in all cases. Parallel processes using the same tag are equivalent to a tagging of their composition. And if processes are equivalent using the standard congruence $\equiv$, they are equivalent under the same tag.

The equivalence rule for new name creation shows that we can have both tagged and untagged elements in this semantics. We use calligraphic lettering to express this situation:

$$\mathcal{Q} ::= [P]_{A,s} \mid \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid (\mathsf{new}\ x)\ \mathcal{Q}$$

$$
\begin{aligned}
[(\mathsf{new}\ x)\ P]_{A,s} &\equiv_{\mathtt{t}} (\mathsf{new}\ x)\ [P]_{A,s} \\
(\mathsf{new}\ x)\ ([P]_{A,s} \mid [Q]_{B,t}) &\equiv_{\mathtt{t}} [P]_{A,s} \mid (\mathsf{new}\ x)\ [Q]_{B,t} \text{ if } x \notin \mathit{fn}(P) \\
[(P \mid Q)]_{A,s} &\equiv_{\mathtt{t}} [P]_{A,s} \mid [Q]_{A,s} \\
[P]_{A,s} &\equiv_{\mathtt{t}} [Q]_{A,s} \text{ if } P \equiv Q
\end{aligned}
$$

Table 4. Structural congruence for tagged processes

Using this syntactic convention, we can now describe the reaction rules in the interface semantics (Table 5). The rule for τ expresses that a process, tagged with automaton A at state s, can only execute a τ-action if the automaton has a corresponding τ-transition from s to a state s'; the resulting process is $[P]_{A,s'}$. Likewise, in the case of an interaction where the input process is tagged with (A, s) and the output process with (B, t), we require that these automata contain transitions labelled with the channel type $\Gamma(x)$, the communication direction, and the arity $|\tilde{z}|$ of the corresponding message. As an example, the buyer and auction processes

$$[\overline{buyer}(bid, item, buyer).P]_{\mathtt{Buyer},2} \mid [buyer(bid, item, buyer).Q]_{\mathtt{Agent},b}$$

could interact by connecting the channels β and γ_1, because $\Gamma(buyer) = \beta$ and in automaton `Buyer` there is a transition labelled $\overline{\beta}\langle 3\rangle$ from state 2 and in the `Agent` automaton there is a matching transition $\gamma_1\langle 3\rangle$ from state b.

In the collaboration rule note that we require collaboration channels to be of the same type. The remaining rules for parallelism, structural congruence, and name creation are straightforward.

The following result for the interface semantics is straightforward: if processes can be executed under some tagging, they can be executed in the standard semantics. In order to formulate the theorem we introduce the notation $\lfloor \mathcal{P} \rfloor$ which strips all tags off the processes of $\mathcal{P}$:

$$
\begin{aligned}
\lfloor [P]_{A,s} \rfloor &= P \\
\lfloor \mathcal{Q}_1 \mid \mathcal{Q}_2 \rfloor &= \lfloor \mathcal{Q}_1 \rfloor \mid \lfloor \mathcal{Q}_2 \rfloor \\
\lfloor (\mathsf{new}\ x)\ \mathcal{Q} \rfloor &= (\mathsf{new}\ x)\ \lfloor \mathcal{Q} \rfloor
\end{aligned}
$$

Then the theorem can be presented in the following concise form:

Theorem 1. *If* $\mathcal{P} \rightarrow_{\mathtt{t}} \mathcal{P}'$ *then* $\lfloor \mathcal{P} \rfloor \rightarrow \lfloor \mathcal{P}' \rfloor$.

Proof. The result is proved by induction over the inference of $\mathcal{P} \rightarrow_{\mathtt{t}} \mathcal{P}'$, using Table 5 and Table 3. □

4 Interface Conformance

In this section we present an approach for checking conformance between processes and their abstractions, which are given as interface automata. We use a

$$\frac{A(s,\tau) = s'}{[\tau.P + M]_{A,s} \rightarrow_{\mathsf{t}} [P]_{A,s'}}$$

$$\frac{A(s, \Gamma(x)\langle|\tilde{y}|\rangle) = s' \quad B(t, \overline{\Gamma(x)}\langle|\tilde{z}|\rangle) = t' \quad |\tilde{y}| = |\tilde{z}|}{[x\tilde{y}.P + M]_{A,s} \mid [\overline{x}\tilde{z}.Q + N]_{B,t} \rightarrow_{\mathsf{t}} [P[\tilde{z}/\tilde{y}]]_{A,s'} \mid [Q]_{B,t'}}$$

$$\frac{A(s, \Gamma(x)\langle \mathsf{c} \rangle) = s' \quad B(t, \Gamma(y)\langle \mathsf{c} \rangle) = t' \quad \Gamma(x) = \Gamma(y) \quad z \text{ fresh}}{[\mathsf{com}\, x.P + M]_{A,s} \mid [\mathsf{com}\, y.Q + N]_{B,t} \rightarrow_{\mathsf{t}} (\mathsf{new}\ z)\ ([P[z/x]]_{A,s'} \mid [Q[z/y]]_{B,t'})}$$

$$\frac{\mathcal{P} \rightarrow_{\mathsf{t}} \mathcal{P}'}{\mathcal{P} \mid \mathcal{Q} \rightarrow_{\mathsf{t}} \mathcal{P}' \mid \mathcal{Q}} \qquad \frac{\mathcal{Q} \equiv_{\mathsf{t}} \mathcal{P} \quad \mathcal{P} \rightarrow_{\mathsf{t}} \mathcal{P}' \quad \mathcal{P}' \equiv_{\mathsf{t}} \mathcal{Q}'}{\mathcal{Q} \rightarrow_{\mathsf{t}} \mathcal{Q}'}$$

$$\frac{\mathcal{P} \rightarrow_{\mathsf{t}} \mathcal{P}'}{(\mathsf{new}\ x)\ \mathcal{P} \rightarrow_{\mathsf{t}} (\mathsf{new}\ x)\ \mathcal{P}'}$$

Table 5. Reaction rules with interface semantics

type system for specifying whether an abstraction conforms to a process. The judgements are of the following form:

$$\Gamma, s \vdash P : A$$

Here Γ is a typing environment (see Section 2.1), s is the state in which the conformance check starts, and A is the smallest interface automaton that conforms to the process P. The typing rules for processes are given in Table 6 and for actions in Table 7.

The rules match the intuition behind the abstractions we have introduced informally in Section 3.1. A nil-process can be abstracted by a single state s. In the rule for replication, we only require the conformance of the replicated process and the automaton. The conformance of the automaton associated with parallel processes follows from the union of the automata that conform to each process. The introduction of new names is abstracted away.

The rule for summation makes use of the auxiliary judgement for actions, and requires that for every occurring action π_i, we can find an outgoing edge abstracting it and leading to some state s_i, and a conformance check taken from s_i will take care of the continuation process P_i. The judgements for actions simply ensure that actions are directly matched by transitions in the automaton.

Type soundness. Before stating the soundness of the type system we first introduce a convention. Note that if we have two typings $\Gamma, s \vdash P : A$ and $\Gamma, s \vdash P : B$, the automata A and B have isomorphic structure but may differ in the names of the states they contain. When relating two automata, we will therefore assume in the following that their states are already renamed in the proper manner.

The following two simple properties express that conformance is preserved under substitution and structural congruence.

$$\Gamma, s \vdash 0 : s \qquad \frac{\Gamma, s \vdash \pi_i : \{s \xrightarrow{\sigma} s_i\} \quad \Gamma, s_i \vdash P_i : A_i}{\Gamma, s \vdash \sum_{i \in I} \pi_i.P_i : \bigcup_i (\{s \xrightarrow{\sigma} s_i\} \cup A_i)}$$

$$\frac{\Gamma, s \vdash P : A}{\Gamma, s \vdash !P : A} \qquad \frac{\Gamma, s \vdash P_1 : A \quad \Gamma, s \vdash P_2 : B}{\Gamma, s \vdash P_1 \mid P_2 : A \cup B} \qquad \frac{\Gamma, s \vdash P : A}{\Gamma, s \vdash (\mathsf{new}\ x)\ P : A}$$

Table 6. Type checking of processes

$$\Gamma, s \vdash x\tilde{y} : \{s \xrightarrow{\Gamma(x)\langle|\tilde{y}|\rangle} s'\} \qquad \Gamma, s \vdash \overline{x}\tilde{y} : \{s \xrightarrow{\overline{\Gamma(x)}\langle|\tilde{y}|\rangle} s'\}$$

$$\Gamma, s \vdash \mathsf{com}\, x : \{s \xrightarrow{\Gamma(x)\langle \mathsf{c} \rangle} s'\} \qquad \Gamma, s \vdash \tau : \{s \xrightarrow{\tau} s'\}$$

Table 7. Type checking of actions

Lemma 1. *If $\Gamma, s \vdash P : A$ and $\Gamma(x) = \Gamma(y)$ then $\Gamma, s \vdash P[y/x] : A$.*

Lemma 2. *If $\Gamma, s \vdash P : A$ and $P \equiv Q$ then $\Gamma, s \vdash Q : A$.*

In order to formulate the theorem we introduce the notation $\mathsf{E}(\mathcal{Q})$ for the set of tagged processes in $\mathcal{Q}$:

$$\begin{array}{ll} \mathsf{E}([P]_{A,s}) & = \{[P]_{A,s}\} \\ \mathsf{E}(\mathcal{Q}_1 \mid \mathcal{Q}_2) & = \mathsf{E}(\mathcal{Q}_1) \cup \mathsf{E}(\mathcal{Q}_2) \\ \mathsf{E}((\mathsf{new}\ x)\ \mathcal{Q}) & = \mathsf{E}(\mathcal{Q}) \end{array}$$

Furthermore we write $A \simeq_s B$ whenever the same states are reachable from a state s in both A and B, i.e. if the following holds

$$A \simeq_s B \ \text{ iff } \ \forall\, \omega.\ A^*(s, \omega) = B^*(s, \omega)$$

where as usual $\delta^*(s, \sigma\omega) = \delta^*(\delta(s, \sigma), \omega)$.

The subject reduction result states that conformance of processes to the associated automata is preserved under the operational semantics.

Theorem 2 (Subject Reduction). *If $[P_i]_{A_i,s_i} \in \mathsf{E}(\mathcal{P})$ and $\Gamma, s_i \vdash P_i : B_i$ such that $A_i \simeq_{s_i} B_i$ and $\lfloor \mathcal{P} \rfloor \to P'$ then there exists a $\mathcal{P}'$ and indices k and j such that $P' = \lfloor \mathcal{P}' \rfloor$ and $[Q_j]_{A_j,t_k} \in \mathsf{E}(\mathcal{P}')$ and $\Gamma, t_k \vdash Q_j : B_j$ and $A_k \simeq_{t_k} B_j$.*

Proof. The result follows from induction in the shape of $\mathcal{P}$, matching the conditions from the type system to those of the interface semantics while applying Lemmas 1 and 2. □

5 Goal-oriented Compositions

In the previous sections we have seen how interface automata can be used as abstractions for processes. The aim of this development is to obtain a strong

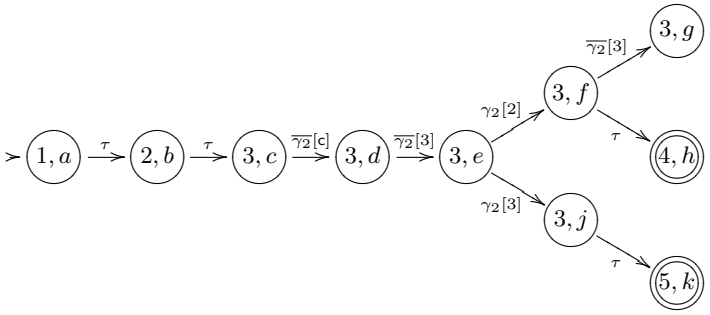


Fig. 3. Composed Automaton: `Buyer` $\otimes_{\beta \leftrightarrow \gamma_1}$ `Agent`

compositionality result for processes from the composition of interface automata, which we define in this section. We define the composition of interface automata as the following product automaton:

Definition 2 (Composition of Interface Automata). *The* composition $A_1 \otimes_{\alpha_1 \leftrightarrow \alpha_2} A_2$ *of two interface automata* $A_1 = (S_1, \Sigma_1, \delta_1, s_1, F_1)$ *and* $A_2 = (S_2, \Sigma_2, \delta_2, s_2, F_2)$ *with* $\sigma_1 = \alpha_1\langle k \rangle \in \Sigma_1^I$ *and* $\sigma_2 = \alpha_2\langle k \rangle \in \Sigma_2^I$ *is defined by*

$$A_1 \otimes_{\alpha_1 \leftrightarrow \alpha_2} A_2 = (S_1 \times S_2, (\Sigma_1 \cup \Sigma_2) \setminus \{\sigma_1, \overline{\sigma_1}, \sigma_2, \overline{\sigma_2}\}, \delta, (s_1, s_2), F_1 \times F_2)$$

where δ *is given by:*

1. *If* $\delta_1(s_1, \sigma_1) = s_1'$ *and* $\delta_2(s_2, \overline{\sigma_2}) = s_2'$ *then* $\delta((s_1, s_2), \tau) = (s_1', s_2')$
2. *A symmetrical rule to 1.*
3. *If* $\delta_1(s_1, \sigma) = s_1'$ *and* $\sigma \notin \{\sigma_1, \overline{\sigma_1}, \sigma_2, \overline{\sigma_2}\}$ *then* $\delta((s_1, s_2), \sigma) = (s_1', s_2)$ *for all* $s_2 \in S_2$
4. *A symmetrical rule to 3.*
5. *If* $\delta_1(s_1, \alpha_1\langle \mathsf{c} \rangle) = s_1'$ *and* $\delta_2(s_2, \alpha_2\langle \mathsf{c} \rangle) = s_2'$ *then* $\delta((s_1, s_2), \tau) = (s_1', s_2')$

Note that although $\otimes_{\alpha_1 \leftrightarrow \alpha_2}$ is a binary operator, we can of course compose arbitrarily many automata together by composing them in sequence, i.e. $A_1 \otimes_{\alpha_1 \leftrightarrow \beta_1} A_2 \otimes_{\alpha_2 \leftrightarrow \beta_2} \cdots \otimes_{\alpha_n \leftrightarrow \beta_n} A_n$, where we assume left-associativity of the composition operator.

In contrast to the approach of [5] where all shared channels are used for composition, we explicitly parametrise the composition operator $\otimes_{\alpha_1 \leftrightarrow \alpha_2}$ with the channels α_1 and α_2 that get connected in the resulting system. These channels are subsequently removed from the alphabet of the resulting automaton, and this automaton contains a τ-transition instead (rules 1 and 2). Channels that are not mentioned in the composition parameter are kept in the result automaton (rules 3 and 4). Also matching collaborations are replaced by a single τ-transition (rule 5). The product thus coincides with the composition of input-enabled automata, such as I/O-automata [14], however, some steps present in A or B may not be present in the product, as not all inputs have to be matched by outputs.

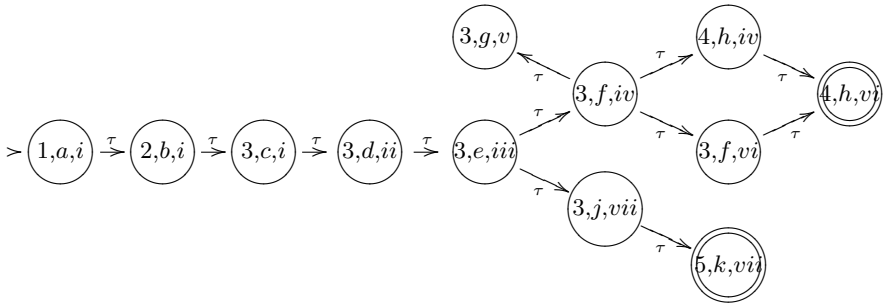


Fig. 4. Composed Automaton: `Buyer` $\otimes_{\beta \leftrightarrow \gamma_1}$ `Agent` $\otimes_{\gamma_2 \leftrightarrow \alpha}$ `Auction`

As shown in Definition 2, we also extend the common notion of composition to take the final states of the automata into account by saying that a composition is *goal-oriented* or meaningful if a final state is reachable, and hence the individual goals of the services are preserved.

Definition 3 (Goal-oriented Composition). *A composition* $A_1 \otimes_{\alpha_1 \leftrightarrow \alpha_2} A_2$ *is said to be* goal-oriented *if it contains a reachable final state.*

A composition is said to be *closed* whenever all the necessary interactions are available in the composed processes (and hence no interaction from the environment is needed) and furthermore a final state is reachable in the product automaton.

Definition 4 (Closedness). (A, s) *is said to be* closed *if, starting from* s*, there is a final state reachable following only* τ *transitions.*

Observe that the traditional definition of closedness is more restrictive than ours, as we only require the *existence* of one path consisting of internal actions, and not every transition to be internal. This allows us to be more flexible because a service may have more than one option to achieve some acceptable goal, and we require that only one is reachable. As an example, consider a scenario where a service acting on behalf of a traveller wishes to compose with either a train ticket service or a plane ticket service. Here it is the case that although the traveller has goals representing both successful train and plane ticket reservations, the agencies provide only one kind of tickets.

Example. As an example of goal-oriented composition, Figure 3 shows the product automaton `Buyer` $\otimes_{\beta \leftrightarrow \gamma_1}$ `Agent`. The automata is connected on the channels β and γ_1, resulting in τ-transitions replacing these in the product automaton. The γ_2 channel present in the `Agent` remains unconnected, hence making further composition possible. Indeed, the automaton can be composed further with `Auction`, as shown in Figure 4. Observe that the resulting composition is closed, as there exists paths containing only τ-transitions that reach final states.

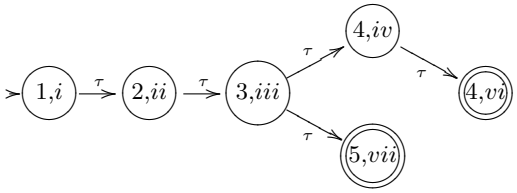


Fig. 5. Composed Automaton: `Buyer` $\otimes_{\beta \hookrightarrow \alpha}$ `Auction`

The composition of the buyer and auction automaton, illustrated in Figure 5, shows how transitions might be lost during composition. Here the buyer places his maximum bid immediately and, when outbid, accepts defeat. The auction however is willing to take another bid from the buyer after the first one is outbid. The composition succeeds because there are still goal states present in the composed automaton; in fact the resulting automaton is closed.

Properties. The main result we establish in this section tells us under which conditions the composition of services leads to a situation where common goals are achieved. More precisely it expresses the following: if the product automaton of process-conformant automaton is closed, then there exists an execution of the processes such that they reach their common goal as specified in the automaton.

Theorem 3. *For* $i = 1, \ldots, n$, *let* $[P_i]_{A_i, s_i} \in \mathsf{E}(\mathcal{P})$ *and* $\Gamma, s_i \vdash P_i : A_i$ *be a typing, and let* $A = A_1 \otimes_{\alpha_1 \hookrightarrow \beta_1} A_2 \otimes_{\alpha_2 \hookrightarrow \beta_2} \cdots \otimes_{\alpha_n \hookrightarrow \beta_n} A_n$ *be the composition of all* A_i. *If* $(A, (s_1, \ldots, s_n))$ *is closed, then* $\mathcal{P} \mid \mathcal{Q} \rightarrow_{\mathfrak{t}}^{*} \mathcal{P}' \mid \mathcal{Q}'$ *and* $[P_i']_{A_i, s_i'} \in \mathsf{E}(\mathcal{P}')$ *and* $(s_1', \ldots, s_n')$ *is a final state in* A.

Note that the property expressed in this theorem can be seen as a relaxation of the common notion of liveness by exploiting the optimistic approach: we ensure that from the state we compose in we can always choose a right path to end up in a desired state (i.e. such a path will always exist).

6 Related Work

Previous work on service oriented computing has primarily focused on web services, for which the *Web Services Description Language (WSDL)* [4] has been influential in describing the static interfaces of services. Another line of work has focused on the choreography of web services, for which the *Web Services Choreography Description Language (WS-CDL)* [10] is a recent attempt at a standard. In this paper we deal with the orthogonal topic of composing services. Several approaches to composing systems based on process algebras and automata have been proposed. However, many of these [16,20,19] have limitations that do not allow them to describe systems in a modular manner, and checking compatibility cannot be performed with a feasible complexity. Canal et al. [3,2] use

roles to define protocol specifications, making them modular, yet the computational complexity of composing systems remains NP-hard. One main benefit of choosing interface automata over other alternatives such as process algebras or input-enabled automata is that the compatibility of services can be checked in linear time [5].

Another approach that deals with these issues is session types. The use of session types [8,9] was introduced to describe structured communication. Gay and Hole [6,7] introduced subtypes for compatibility and conformance testing of processes. Vallecillo et al. [18] continued the investigation of composition of compatibility of session types, applying their approach in the commercial environment CORBA. The present work differs from session types in a core point, namely the fact that goal-oriented composition allows us to express when composition is meaningful. Another novel feature is the ability to compose services component-wise, allowing arbitrarily large dynamic composition scenarios, rather than focusing on two sessions being dual or complementary.

Recently, Larsen et al. [12] have presented an interface theory, *modal I/O automata*, that adds modalities to interface automata such that requirements of the system can be directly modelled. As modal I/O automata are more general than the interface automata, it would be interesting to see what benefits can be gained by lifting our abstractions to this approach.

Goal-orientation is an important term in requirements engineering [11]. In this field the term is used to describe techniques to identify and refine requirements guided by goals. In our work we use goals to guide the composition of services. Hence our work is orthogonal to these techniques.

7 Conclusion

We have extended the π-calculus with an explicit action for service collaborations, and have presented an interface automata-based abstraction of these processes in order to reason about the meaningfulness of the arising process compositions. A type checking algorithm has been provided for ensuring the conformance between a process and its abstracting automaton, and we have extended the theory of composition of interface automata by introducing goal conditions. The notion of closedness of compositions could then be relaxed in a way that only the required interactions needed to be part of a composition, thus establishing a very flexible notion of compositionality.

In future work we would like to investigate the flexibility of having optional as well as required goal states, resulting in an even more fine grained specification of meaningful service composition. Furthermore, we could strengthen the semantics of the collaboration action by transforming it from a mere annotation of the start of a collaboration to an action which checks compositionality before executing, allowing us to express statically when a process is safe with respect to goal-oriented composition.

Acknowledgements. This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.
2. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Transactions on Software Engineering*, 29(3):242–260, 2003.
3. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, 2001.
4. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL). `http://www.w3.org/TR/wsdl`, Mar. 2001.
5. L. de Alfaro and T. A. Henzinger. Interface automata. In *9th Intl. Symposium on Foundations of Software Engineering (FSE'01)*, pages 109–120. ACM, 2001.
6. S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 74–90. Springer, 1999.
7. S. J. Gay and M. Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
8. K. Honda. Types for dynamic interaction. In *4th Intl. Conference on Concurrency (CONCUR'93)*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
9. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
10. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/`, Nov. 2005.
11. A. v. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *5th IEEE Intl. Symposium on Requirements Engineering (RE'01)*, pages 249–262. IEEE Computer Society, 2001.
12. K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In *16th European Symposium on Programming Languages and Systems (ESOP'07)*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
13. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th Annual Symposium on Principles of Distributed Computing (PODC'87)*, pages 137–151, 1987.
14. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
16. O. Nierstrasz. Regular types for active objects. In *8th Annual Conference Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 1–15. ACM, 1993.
17. M. P. Singh and M. N. Huhns. *Service-oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Ltd, 2005.
18. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundamenta Informaticae*, 73(4):583–598, 2006.
19. H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 23(2):143–170, 2003.
20. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.