Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services (Long Version*, version 2, January, 7th, 2008)**

Tarek Melliti[1], Pascal Poizat[1,2], and Sonia Ben Mokhtar[2]

[1] IBISC FRE 2873 CNRS – Université d'Évry Val d'Essonne, France
tarek.melliti@ibisc.univ-evry.fr
[2] INRIA/ARLES project-team, France
{pascal.poizat,sonia.ben_mokhtar}@inria.fr

Abstract. Services are developed separately and without knowledge of all possible use contexts. They often mismatch or do not correspond exactly to the end-user needs, making direct composition without mediation impossible. In such a case, software adaptation can support composition by producing semi-automatically new software pieces called adaptors. Adaptation proposals have addressed the signature and behavioural service interface levels. Yet, taking also into account the semantic level is mandatory to enable the fully-automatic retrieval of adaptors from service interfaces. We propose a new adaptation technique that, compared to related work, supports both behavioural and semantic service interface levels, works system-wide, and generates automatically distributed adaptors.
keywords: Model-Based Adaptation, Behavioural Adaptation, Semantic Adaptation, Services, Input Output Labelled Transition Systems.

1 Introduction

Service Oriented Architectures (SOA) [21] have introduced a new organizing of software, based on *services*, self describing and loosely coupled interacting software components that support the rapid and low-cost composition of distributed applications. An important issue in SOA is *service composition and its automation* [21, 16], either to fulfill a user task or to have services collaborating in added-value composite services. Techniques that support the composition in component or service based systems rely on four interface description levels: signature (operations), behaviour (protocols), non functional (time, QoS) and semantics [22]. In SOA, *service composition* takes place after services have been discovered. It is often assumed that discovered services conform at the different interface levels, and first, at the signature one where syntactic matching is used to put in correspondence required and provided functionalities. Approaches

---

* A shorter version of this paper has been published in the proceedings of FASE'08.

that support the behavioural (called *conversation*) and semantics levels assume one-to-one functionality correspondences [5, 3]. These assumptions do not yield in practice in open heterogeneous environments where services are developed by different organizations.

Software adaptation [11] has provided solutions for component interoperability through the computation – from component interfaces and user-defined adaptation specifications called *mappings* – of *adaptors* that operate in-between components to ensure their correct[3] composition at the signature and behavioural levels [11, 6, 2, 19], and more recently at the non-functional level [24]. Yet, while *automatic adaptation* is highly desirable for SOA where systems are composed from dynamically discovered services, component adaptation techniques do not support the semantic level and therefore require a mapping to be given by a designer to deal for example with message name mismatch between services. Moreover, distributed adaptation is an important issue in domains such as pervasive computing, due to the use of small-resource devices and ad-hoc networks (no centralized server being available to execute the adaptor). Our objective is to overcome limitations of both semantic service composition – supporting complex dependencies between services or functionalities – and software adaptation – enabling one to obtain automatically distributed (local) adaptor models directly from service interface models, without requiring some mapping to be given, by supporting the semantic level in the adaptation process.

In the sequel, we present first a formalizing of service behavioural interfaces with associated semantic information (Sect. 2). We then develop our automatic service composition and adaptation technique (Sect. 3). Related work is discussed in Section 4 and we end with conclusions, including limitations of our work and perspectives.

2   A Model of Semantic Service Specifications

In this section we present our service model including service interfaces and service conversations. It is then extended with semantic information to enable automatic composition and adaptation.

Example 1 (Presentation). In the sequel we consider a simple pervasive system with four services. PDA is a service on top of a PDA which stores music files (mp3 or ogg). It is used to transmit music to be played at a given volume. dBMeter is a sensor which is used to transmit (in dB) the noise level in the room. HF is a Hi-Fi system which can play ogg files at a given volume which is adjusted according to the required one and the ambient noise level. Finally, Trans is a service that translates mp3 files into ogg files at a certain compression level, depending on the ambient noise level.

2.1   Service Interfaces (Interface Signature Level)

Service interfaces are used to advertise service provided functionalities to potential service clients. They are described, in the case of Web services, using WSDL[4]

---

[3] In the sense of deadlock freeness.
[4] http://www.w3.org/TR/wsdl for WSDL 1.1.

as a set of *provided operations*, each described using a signature, *i.e.*, typed in/out arguments, as well as the corresponding XML messages carrying them. We abstract the WSDL elements we use for adaptation in our model as follows. We define M as the set of XML messages and *opNames* as the set of operation names, over which we range respectively using $m, m_1, \ldots$ and *op*. The symbols "?" and "!" used with messages denote respectively input and output, *e.g.*, $?m$ means receiving a message of type m. O is the set of operations, over which we range using o. An operation $o \in O$ can be either one-way $o = opName[?m]$ (solicitation) or $o = opName[!m]$ (notification); or two-way $o = opName[?m, !m]$ (request-response) or $o = opName[?m, ?m]$ (notification-response). Ω is the set of service names, over which we range using ω, w or w depending of the context. $\perp$ denotes an undefined message. $Input : O \rightarrow M \cup \{\perp\}$ returns the input message of an operation: $Input(o) = m$ if $o = op[?m]$, $o = op[?m, !m']$ or $o = op[!m', ?m]$); $\perp$ otherwise. $Output : O \rightarrow M \cup \{\perp\}$ returns the output message of an operation: $Output(o) = m$ if $o = op[!m]$, $o = op[?m', !m]$ or $o = op[!m, ?m']$; $\perp$ otherwise.

Example 2 (Service Interfaces). dBMeter (w1) has an operation to give the dB level, infodB$[?m11_a, !m11_b]$. HF (w2) has an operation play$[?m21]$ to play music. It also has an operation over which the ambient noise level can be given, ambiance$[?m22]$. Finally, an operation output$[!m23]$ outputs sound and volume. Trans (w3) has an operation, trans$[?m31_a, !m31_b]$ to translate music files. PDA (w4) provides no operation. This kind of pure client is used to model some user task the service composition is built for. It is implemented with additional operations corresponding to the interaction with (usually) a user interface. We name messages in correspondence with the service they correspond to, *e.g.*, m$\underline{31}_a$ for Trans (w$\underline{3}$), see also the operations on the left of Figure 1.

2.2   Service Conversations (Interface Behavioural Level)

In addition to the set of operations specified in the service interface, an elementary service described using BPEL4WS[5] (or BPEL for short) defines a long-run interaction protocol, called *business protocol*, where operations are invoked according to ordering and time constraints. These protocols mix both internal and external behaviours, while only the latter ones (conversations) are relevant for automatic composition: they constitute the service behavioural interface. Our solution relies on the derivation of Abstract BPEL[5] conversations from BPEL processes. This process, which abstracts from the service internal activities, is automated by a tool [18] and only briefly presented here since, for our present purpose, only its Abstract BPEL end-result is important.

Basic activities. The most relevant activities for service composition are *communication activities*: *invoke*[o] (invocation of operation o), *receive*[o] (reception of an operation o invocation, forbidden for notification), and *reply*[o] (response sending for an operation o invocation, forbidden for solicitation). They specify the communication constraints between a service and its *partners*, *i.e.*, the set of services/clients that interact with it. We distinguish two kinds of invocations: if the operation owner is known or if it has to be instantiated at run time. In the first case, invocation is assumed to be internal and is hidden. In the second

---

[5] http://www.ibm.com/developerworks/library/specification/ws-bpel for BPEL 1.1.

case, we rely on a set of free variables $\overline{X} = \{x_1, x_2, \ldots\}$ to refer to service own-ers, with a fresh variable for each *invoke*, written $invoke[o](x_i)$. Moreover, as a composition involves several services, we ensure disjointness of fresh variables by indexing them with the service name, *e.g.*, $invoke[o](x_{i_\omega})$. *Time activities* are used for example to define timeouts or watchdogs. They can be reduced to a time passing activity (which we will denote *time*) and the use of *scope* (see below). Finally, *empty* represents a void activity and *terminate* a terminated one. All other BPEL basic activities – *e.g.*, those locally executed by services, mainly providing data handling facilities – are hidden in interfaces.

Structured activities are control flow constructors. Each one defines an order with which activities are activated or executed, and can be applied to either basic or structured activities (both over which we range using $P, Q, R, \ldots$). We support here a simple subset of BPEL: parallel execution ($flow[\{P_{i,i\in\{1,\ldots,n\}}\}]$) with joint links ignored for simplicity, conditional execution ($switch[\{(\_, P_i)_{i\in\{1,\ldots,n\}}\}]$) and loops ($while(\_, P)$) where we assimilate conditions to internal non-deterministic choice (choice is performed internally, without external control over it). Finally, $scope(P, EH)$ encapsulates P with an event handler,

$$EH = [\{(m_i, P_i)_{i\in\{1,\ldots,n\}}\}, (d, Q), \{(e_j, R_j)_{j\in\{1,\ldots,m\}}\}]$$

An event handler is made up of sets of events, one for each type of event – an event can be a received message (m_i), a time event related to a duration (d) or a raised exception (e_j) – and associated activities: *scope* behaves as P if none of the events happens and as a given sub-activity (some P_i, Q, or R_j) if the corresponding event happens. Other constructs could be supported provided it is possible to translate them into Timed Input Output Labelled Transition System (TIOLTS) as in Section 2.4.

     We may now introduce our formal definition of a service.

Definition 1 (Service). *A* service *is a tuple* $\langle \omega, O^\bullet, O^\circ, B(\overline{X}) \rangle$ *where* $\omega \in \Omega$ *is the service name (used as an abstraction of, e.g., XML name spaces),* $O^\bullet \subseteq O$ *is a WSDL interface that defines the service's set of provided operations,* $O^\circ \subseteq O$ *is a set of required operations, and* $B(\overline{X}) \in ABP$, *where* ABP *denotes the set of Abstract BPEL processes, is the service conversation, defined over a set of free variables* $\overline{X} = \{x_1, \ldots, x_n\}$. *Moreover,* $B(\overline{X})$ *respects:* $o \in O^\circ$ *for every* $invoke[o](x)$ *and* $o \in O^\bullet$ *for every* $receive[o]$ *and every* $reply[o]$.

Due to the assumed uniqueness of services names, we do not distinguish service names from the corresponding service definition, *i.e.*, for a service $\langle \omega, O^\bullet, O^\circ, B(\overline{X}) \rangle$ we may write $\omega = \langle O^\bullet, O^\circ, B(\overline{X}) \rangle$. We further define for a service ω: $O_\omega = O^\bullet \cup O^\circ$, $In_\omega = (\bigcup_{o \in O_\omega}\{Input(o)\}) \setminus \{\bot\}$, $Out_\omega = (\bigcup_{o \in O_\omega}\{Output(o)\}) \setminus \{\bot\}$, $M_\omega = In_\omega \cup Out_\omega$, and $M_\omega^{IO} = \{?m \mid m \in In_\omega\} \cup \{!m \mid m \in Out_\omega\}$.

Example 3 (Service Specifications). We can now give more detail about our services. dBMeter and HF have no required operations, and their behavioural interfaces are re-spectively receive[infodB] ; reply[infodB] ; empty and receive[play] ; receive[ambiance] ; reply[output] ; empty. Trans has a required operation, getNoise[?m32$_a$,!m32$_b$] and

its behaviour is switch[(_,receive[trans] ; invoke[getNoise](x_{1w3}) ; reply[trans] ; empty), (_,terminate)] (it may terminate directly if not used). Finally, PDA has one required operation for each kind of music file, mp3Play[?m41$_a$,!m41$_b$] and oggPlay[?m42$_a$,!m42$_b$], and its behaviour is switch[(_,invoke[mp3Play](x_{1w4})),(_,invoke[oggPlay](x_{2w4}))] ; empty.

2.3   Semantic Information (Interface Semantic Level)

To support automatic composition, service descriptions must be extended with descriptive semantic information. A number of research efforts have been conducted for Web service semantic annotation, but SAWSDL[6] has become the W3C recommendation for the semantic annotation of WSDL documents. In this section we introduce a formal model for representing the descriptive semantics of a service described using SAWSDL complemented with BPEL. This integrated formal model allows reasoning on service compatibility at three levels at the same time (signature, behavioural and semantic levels).

Definition 2 (Semantic Structure). *A semantic structure $\mathcal{I}$ is a couple $(\mathcal{U}, \mathcal{R})$ where $\mathcal{U}$ is a set of units of sense (*UoS*), over which we range using u, and $\mathcal{R} \subseteq 2^{\mathcal{U}} \times \mathcal{U}$ is a relation where $(U, u) \in \mathcal{R}$ denotes that given a set U of UoS, one can obtain u.*

These structures support partner collaboration and can be related to concrete ontologies referenced in the SAWSDL service description, where units of sense correspond to the ontology concepts and properties, while $\mathcal{R}$ can be used to encapsulate relations such as the "subclassOf" one, *i.e.*, $\forall u, u' \in \mathcal{U}$, u' subclassOf $u \Rightarrow (\{u'\}, u) \in \mathcal{R}$. A semantic structure may result from ontology integration, *e.g.*, following [7], and support different semantic structures for different partners.

Example 4 (Semantic Structure). Elements of $\mathcal{U}$ are mfile (music file), ogg (ogg file), mp3 (mp3 file), vol (volume), noise (noise information), dB (noise in dB), sound (sound return), and info (information feedback). The relations between them are: ({ogg},mfile) and ({mp3},mfile) (ogg and mp3 are music files), ({dB},noise) (noise information can be retrieved from dB), and ({sound,vol},info) (sound and volume build an information).

A service receives requests, process them and sends answers back. Yet, to process a request, behind its representation (the message format), a set of information, UoS, is required. In turn, replies contain (possibly new) UoS. For instance, in order to play music, the HF service requires an ogg file and a volume, and outputs sound and volume. Moreover, for Web services, this information has to be XML-formatted (and published in the service SAWSDL interface). To support the automatic use of such a service by a partner, one has to ensure that all required information is known by the partner, format the request message and package the information into it, call the service, get the response, and finally process it to extract the set of information that is returned back. This principle is at the core of our adaptor behaviours and is first supported through semantic matching functions and a formal definition of partnership.

---

[6] http://www.w3.org/TR/sawsdl/

Definition 3 ((Semantic) Matching Function). *A* matching function *for a service ω over a semantic structure $\mathcal{I} = (\mathcal{U}, \mathcal{R})$ is a function $SM_{\omega,\mathcal{I}} : M_\omega \rightarrow 2^{\mathcal{U} \times Xpath(M_\omega)}$ with $Xpath(M_\omega)$ the set of Xpath expressions defined over M_ω.*

These functions are used to associate to each message the set of UoS it corresponds to, together with a syntactic expression ($Xpath$) that makes it possible to relate these UoS with the message XML tree.

Example 5 (Matching Functions). We have dbMeter: $(m11_a, \emptyset)$ and $(m11_b, \{(dB,\_)\})$ for infodB (dB output) – HF: $(m21, \{(ogg,\_),(vol,\_)\})$ and $(m23, \{(sound,\_),(vol,\_)\})$ for play and output (inputs ogg file and volume, outputs sound and volume), $(m22, \{(dB,\_)\})$ for ambiance (inputs ambient noise level) – Trans: $(m31_a, \{(mp3,\_)\})$ and $(m31_b, \{(ogg,\_)\})$ for trans (inputs mp3, outputs ogg), $(m32_a, \emptyset)$ and $(m32_b, \{(noise,\_)\})$ for getNoise (noise returned back) – PDA: $(m41_a, \{(mp3,\_),(vol,\_)\})$ and $(m41_b, \{(info,\_)\})$ for mp3Play, $(m42_a, \{(ogg,\_),(vol,\_)\})$ and $(m42_b, \{(info,\_)\})$ for oggPlay (file and volume sent, information expected in the end). Xpath information is omitted ($\_$).

We introduce hereafter the formal definition of *partners* and *partnerships* as a set of services collaborating on top of a semantic structure. We suppose an enumerable set Id over which partners are indexed (it can be naturals or the set of partners' names).

Definition 4 (Partner and Partnership). *A* partner *over a semantic structure $\mathcal{I}$ is a tuple $\rho_\omega = \langle \omega, \mathcal{I}, SM_{\omega,\mathcal{I}} \rangle$ where ω is a service and $SM_{\omega,\mathcal{I}}$ is a matching function for ω over $\mathcal{I}$. A* partnership *over a semantic structure $\mathcal{I}$ is a set of partners $\Upsilon_{Id} = \{\langle \omega_i, \mathcal{I}, SM_{\omega_i,\mathcal{I}} \rangle_{i \in Id}\}$ over $\mathcal{I}$. When clear, suffixes are omitted.*

2.4   Operational Semantics of Semantic Services

We present now the formal semantics of partners using operational semantics to favor operational issues such as algorithms and tools.

Configurations. To support automatic semantic composition, the operational semantics of a partner $\langle \omega, (\mathcal{U}, \mathcal{R}), SM \rangle$ should be defined through its evolution over time, directed by its behaviour, of hypotheses on the UoS it holds. This can be described using *configurations* $(P, \mathcal{H})$ where $P \in ABP$ is the current process representing the partner and $\mathcal{H} \in 2^{\mathcal{U}}$ is its current semantic environment. $\mathcal{H}^{\mathcal{R}^*}$ denotes the closure of $\mathcal{H}$ over $\mathcal{R}$ and $\mathcal{H} \rightsquigarrow_{\mathcal{R}} u$ that the u UoS can be obtained from $\mathcal{H}$: $\mathcal{H} \rightsquigarrow_{\mathcal{R}} u$ iff $u \in \mathcal{H}^{\mathcal{R}^*}$. When clear from the context (remind that all partners in a partnership share a common $\mathcal{R}$) this is simply noted $\mathcal{H} \rightsquigarrow u$. We also suppose that a UoS belongs to a partner configuration until it terminates.

Events. The semantics depends on message communication which is modelled using events ($!m$ and $?m$). We also introduce several specific events. As services may evolve in an unobservable way (*e.g.*, due to a condition abstraction), *tau* is used to denote internal actions. The termination event, $/$, enables the detection of service termination. Time is supported by χ that denotes the passing of one time unit (which stands for any delay). This is compatible with the fact that

the time constraints of a Web service are generally soft, thus this discretization of time is a valid abstraction [18]. We define Exc to be the set of exceptions, $!M = \{!m \mid m \in M\}$, $?M = \{?m \mid m \in M\}$ and $Event = !M \cup ?M \cup \{tau, /, \chi\}$. Moreover, we define *complementarity* as $(?m)^c = !m$, $(!m)^c = ?m$, and $a^c = a$ for all $a \in Event \backslash (!M \cup ?M)$. We introduce hereafter a structural operational semantics (SOS) for our semantic services. In this semantics, we are in the context of a given partner $\rho_\omega = \langle \omega, \mathcal{I}, SM_{\omega, \mathcal{I}} \rangle$.

Basic Activities are denoted by basic processes which are *terminate*, *empty*, *time*, *receive*[o], *reply*[o] and *invoke*[o].

time has one axiom, and **empty** can only terminate:

$$(time, \mathcal{H}) \xrightarrow{\chi} (time, \mathcal{H}) \qquad\qquad (empty, \mathcal{H}) \xrightarrow{/} (terminate, \emptyset).$$

receive[o]. Upon reception of the corresponding message, its UoS are augmented with the message ones (also for *invoke*, below):

$$(receive[o], \mathcal{H}) \xrightarrow{?\omega.Input(o)} (empty, \mathcal{H} \cup SM(Input(o))$$

reply[o]. The UoS needed to build the message corresponding to a reply have to be obtainable from the ones in the configuration (also for *invoke*, below):

$$(reply[o], \mathcal{H}) \xrightarrow{!\omega.Output(o)} (empty, \mathcal{H}) \text{ if } \mathcal{H} \rightsquigarrow SM(Output(o))$$

invoke[o](x) semantics depends on the form of o:

$$(invoke[o](x), \mathcal{H}) \xrightarrow{!x.Input(o)} (empty, \mathcal{H}) \text{ if } \mathcal{H} \rightsquigarrow SM(Input(o))$$
$$\text{when } o = op[?m]$$

$$(invoke[o](x), \mathcal{H}) \xrightarrow{?x.Output(o)} (empty, \mathcal{H} \cup SM(Output(o)))$$
$$\text{when } o = op[!m]$$

$$(invoke[o](x), \mathcal{H}) \xrightarrow{!x.Input(o)} (invoke[op[!m_2]](x), \mathcal{H}) \text{ if } \rightsquigarrow SM(Output(o))$$
$$\text{when } o = op[?m_1, !m_2]$$

$$(invoke[o](x), \mathcal{H}) \xrightarrow{?x.Output(o)} (invoke[op[?m_2]](x), \mathcal{H} \cup SM(Output(o)))$$
$$\text{when } o = op[!m_1, ?m_2]$$

For one-way operations the process executes the event related to the operation signature and becomes *empty*. For two-way operations, the first event is executed and then the process becomes an *invoke* corresponding to the remaining (now one-way) operation. If *invoke* operates on a partner operation that starts with an input message then the associated event is an output message and *vice versa*. Events are prefixed by partner names.

Structured Activities are supported in a structured way as usual in process algebraic SOS for Web services. Since the basic activities are the main ones for this work, structured activity rules are presented in Appendix A. The modular application of basic and structured rules associates a TIOLTS to each ABP process.
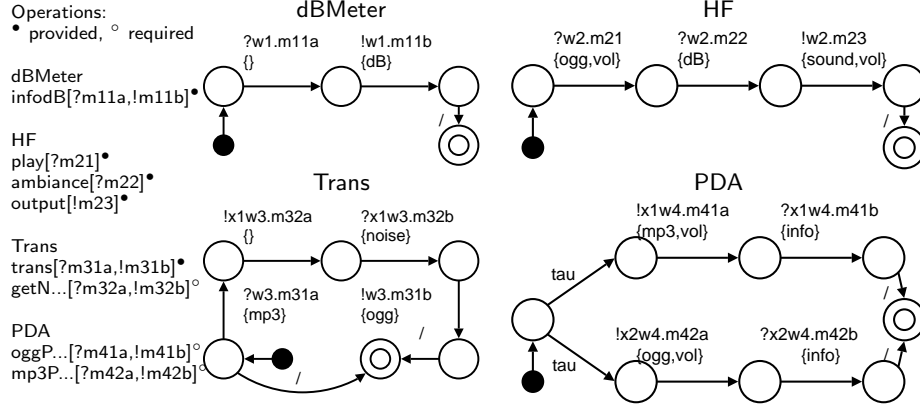
Fig. 1. Partners TIOLTS

Definition 5 (Partner External Behaviour). *The* external behaviour *of a partner* $\rho = \langle \omega = \langle O^\bullet, O^\circ, B(\overline{X}) \rangle, (\mathcal{U}, \mathcal{R}), SM \rangle$ *is a TIOLTS* $\mathcal{L}_\rho = \langle A, S, s_0, F, T(\overline{X}) \rangle$ *where* $A = M_\omega^{IO} \cup \{tau, /, \chi\}$ *is the alphabet,* $S \subseteq ABP \times 2^{\mathcal{U}}$ *is the set of configurations,* $s_0 = (B(\overline{X}), \emptyset)$ *is the initial state* $(s_0 \in S)$, $F = \{(s_f, \mathcal{H}_f) \in S \mid \exists (s, \mathcal{H}) \in S \wedge (s, \mathcal{H}) \xrightarrow{/} (s_f, \mathcal{H}_f) \in T\}$ *is the set of final states, and* $T(\overline{X}) \subseteq S \times A \times S$ *is the transition relation obtained from the SOS rules* $(\overline{X}$ *is the set of the free variables used in the transitions).*

Based on this definition, we develop techniques operating on TIOLTS. Hence, in service and partner structures, TIOLTS will be used for *ABP* processes.

Example 6. Partners TIOLTS are given in Figure 1.

3   Automatic Composition using Adaptation

To be compatible, two partners have to share complementary operations, messages and matching functions, *i.e.*, the UoS sent by one must correspond to UoS required by the second. Moreover, their two behaviours should also be compatible to ensure deadlock freedom. We reuse a relation defined in [18] which intuitively states that to be compatible (denoted using $\sim^c$), two behaviours must be such that (i) at each step of the interaction, each sent message can be received, and (ii) each expected message possibly corresponds to a sent one, regarding the history of past exchanged messages. This is related to bisimulation, yet taking into account the difference between sent and received messages. This is also related to the compatibility relation for interface automata [14] but with support for internal actions and time.

As usual, an observable relation between states is defined as $s \overset{a}{\Rightarrow} s'$ iff $s \xrightarrow{tau^* a tau^*} s'$ and $s \overset{\epsilon}{\Rightarrow} s'$ iff $s \xrightarrow{tau^*} s'$. Moreover, $s \overset{a}{\Longrightarrow}_i s'$ stands for $(s, a, s') \in T_i$ and w denotes a word.

Definition 6 (Behavioural Compatibility [18]). *Two TIOLTS $\mathcal{L}_i = \langle A_i, S_i, s_{0_i}, F_i, T_i(\overline{X}) \rangle$, $i \in \{1, 2\}$, are compatible iff there is a relation $\sim^c \subseteq S_1 \times S_2$ such that $s_{0_1} \sim^c s_{0_2}$ and for every s_1, s_2 such that $s_1 \sim^c s_2$:*

- *$\forall a \in Event\backslash ?M$, $\forall i, j \in \{1, 2\}, j \neq i$, if $s_i \stackrel{a}{\Longrightarrow}_i s_i'$ then there is $s_j \stackrel{a^c}{\Longrightarrow}_j s_j'$ with $s_i' \sim^c s_j'$*

- *$\forall m \in M$, $\forall i, j \in \{1, 2\}, j \neq i$, if $s_i \stackrel{?m}{\Longrightarrow}_i s_i'$ then there are $s_j^- \stackrel{w}{\Longrightarrow}_j s_j$, $s_j^- \stackrel{w}{\Longrightarrow}_j s_j^+$, $s_j^+ \stackrel{!m}{\Longrightarrow}_j s_j'$ with $s_i \sim^c s_j^+$ and $s_i' \sim^c s_j'$, and there is $s_j \stackrel{!m'}{\Longrightarrow}_j s_j'$.*

Definition 7 (Partner Compatibility). *Two partners $\rho_i = \langle \omega_i = \langle O_i^\bullet, O_i^\circ, \mathcal{L}_{\omega_i} \rangle, \mathcal{I}, SM_i \rangle$, $i \in \{1, 2\}$, are compatible, noted $\rho_1 \sim^c \rho_2$, iff $O_1^\bullet = O_2^\circ$, $O_2^\bullet = O_1^\circ$, $SM_1 = SM_2$ and $\mathcal{L}_{\omega_1} \sim^c \mathcal{L}_{\omega_2}$.*

It is not realistic to suppose that compatibility yields from scratch in a context where services are designed by different parties. To release these strict composition constraints, a one-to-one correspondence between names could be supported parameterizing the compatibility relation by a name correspondence mapping (hence solving simple name mismatch), *i.e.*, for two services ω_1 and ω_2, defined over $M_{\omega_1} \times M_{\omega_2}$. However, it is not straightforward to support in such a way more complex correspondences and semantic information, as the UoS required for a partner's received message may correspond to several partners' sent messages. For example, ogg and vol for m21 in HF may come either directly from PDA (using $m42_a$) or using both PDA ($m41_a$) and Trans ($m31_b$). The possible need for message reordering and cyclic dependencies between required/produced UoS are also important locks. Recent advances in software adaptation [6, 19] may help there, provided they are extended to support semantic information.

To solve these issues, we propose an approach were an adaptor is generated for each partner, which will only communicate through it. We process in three steps: (i) the generation of a compatible (correct) service client (CSC) for each partner, which allows to interact with the partner without changing its protocol, (ii) using the CSCs, the definition of a central global adaptor that defines all the valid interactions between partners, and finally (iii) the transformation of CSCs into local adaptors using the global adaptor protocol. The final desired architecture (adapted system, see Fig. 2) is such that there is compatibility (i) between each partner and its local adaptor (hiding its communications with the other adaptors) and (ii) between the adaptors (hiding their communications with their partners). This yields the deadlock freedom of the global system.

3.1 Step i: Generation of the Correct Service Clients (CSCs)

We compute, for each partner, a CSC which acts as a perfect environment for it and ensures compatibility by construction. This means that the CSC acts for its partner as if it provides all required operations and is always ready to consume (resp. send) its partner sent (resp. received) messages. This idea originates from controller synthesis and has been applied both in software adaptation [2] and in service compatibility checking [18]. We reuse the latter as step (i) is independent
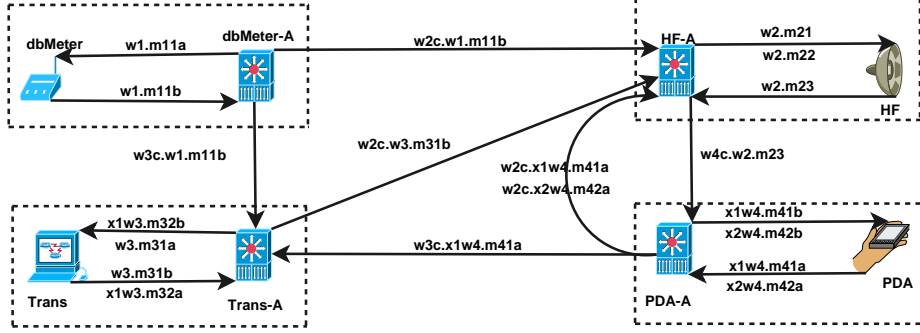
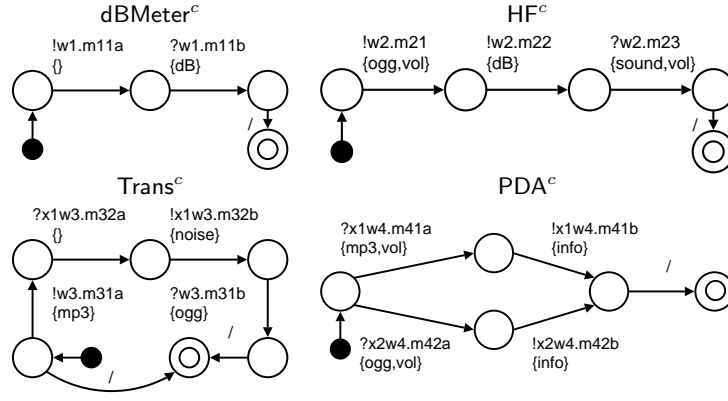**Fig. 2.** Adapted System Architecture

Fig. 3. CSC TIOLTS

from the semantic information. Two tasks are performed on a partner TIOLTS to build its CSC TIOLTS: messages are complemented (exchanging directions) and the resulting TIOLTS is determinized. We fail when the CSC TIOLTS is non-deterministic on *tau* transitions or message sending as this ambiguity yields non implementable behaviours [18].

Definition 8 (Correct Service Client (CSC)). *The correct service client (CSC) for a partner* $\rho = \langle \omega = \langle O^\bullet, O^\circ, \mathcal{L}_\omega = \langle A, S, s_0, F, T(\overline{X}) \rangle \rangle, (\mathcal{U}, \mathcal{R}), SM \rangle$ *is a partner* $\rho^c = \langle \omega^c = \langle O^\circ, O^\bullet, \mathcal{L}_\omega^c \rangle, (\mathcal{U}, \mathcal{R}), SM \rangle$ *where* $\mathcal{L}_\omega^c$ *(*$\mathcal{L}_\omega \sim^c \mathcal{L}_\omega^c$*) is computed using the [18] synthesis algorithm. When clear,* $\mathcal{L}_\omega^c$ *is also written* $\mathcal{L}_\rho^c$*.*

Example 7. The CSC TIOLTSs are given in Figure 3.

At the time, a CSC does not really provide the required operations to its partner, but only the corresponding received and/or send messages. The CSC must now

be *extended* with implementations in terms of calls (resp. replies) to the other CSCs. Take Trans^c in Figure 3 for example. Its provided operation for noise information ($\mathsf{getNoise}$, called with message $\mathsf{m32}_a$ and noise result returned with message $\mathsf{m32}_b$) should be implemented by getting this noise UoS from some other CSC (here, $\mathsf{dBMeter}^c$, using the relation between dB and noise). Moreover, the extension of a CSC should also support additional message exchanges for some messages sent with UoS by this CSC (*e.g.*, for Trans^c, this means getting the $\mathsf{mp3}$ UoS from some other CSC (here, PDA^c) before sending it to Trans using $\mathsf{m31}_a$). All these extensions correspond to the message exchanges in between local adaptors in Figure 2. This is the objective of the next steps, where a global structure (called global adaptor) is first built (step ii) before being used to extend the CSCs into local adaptors (step iii).

3.2   Step ii: Generation of a Global Adaptor (GA)

Composing partners correctly in a partnership means to have them exchange messages in a way that guarantees that each partner fulfills its role in the composition until all terminate. The global adaptor (GA) is a composition of the CSC defined such that the orderings imposed by CSC protocols are respected, The GA is ready to receive a message from any CSC at any time (augmenting correspondingly its UoS, Def. 9,(1)), and can only send a message when it has all the required information at its disposal (Def. 9,(2)). From one partner's point of view, this ensures that a given operation is run only when the other partners have sent all required information. Moreover, / is synchronized to ensure correct termination and time passing is weakly synchronized. The GA corresponds to a form of free product of the CSCs restricted by constraints over UoS.

The GA has access to UoS originating from different messages of different partners. We need to distinguish them in order to support step (iii): when adaptation is distributed, we must be able to detect which messages were used to obtain a UoS. Therefore, for a partnership $\Upsilon_{Id} = \{\langle \omega_i, \mathcal{I}, SM_{\omega_i} \rangle_{i \in Id}\}$, in the GA configurations the semantic information will be taken in $2^{\mathcal{E}}$ with $\mathcal{E} = \bigcup_{i \in Id}\{(m, u) \mid m \in M_{\omega_i} \wedge u \in SM_{\omega_i}(m)\}$. Given some $E \subseteq \mathcal{E}$, we also define projections on the message, $\pi^{msg}(E) = \{m \mid (m, u) \in E\}$, and on the UoS, $\pi^{uos}(E) = \{u \mid (m, u) \in E\}$.

Definition 9 (Global Adaptor Generation). *Let $\Upsilon_{Id} = \{\rho_{i, i \in Id}\}$ be a partnership with a corresponding set of CSC, $\{\rho_i^c = \langle \omega_i = \langle O_i^\bullet, O_i^\circ, \mathcal{L}_i \rangle\rangle, (\mathcal{U}, \mathcal{R}), SM_i)_{i \in Id}\}$ where for each i in Id, $\mathcal{L}_i = \langle A_i, S_i, s_{0_i}, F_i, T_i \rangle$. The global adaptor for Υ_{Id} is the TIOLTS $\mathcal{A}_{\Upsilon_{Id}} = \langle A, S, s_0, F, T \rangle$ where $A = \bigcup_{i \in Id} A_i$, $S \subseteq (\Pi_{i \in Id}(S_i)) \times 2^{\mathcal{E}}$, $s_0 = (\Pi_{i \in Id}(s_{0_i}), \emptyset)$, $F = \{(s_1, ..., s_n) \in S \mid \forall i \in Id, \ s_i \in F_i\}$, and T is defined as: $\forall s = ((s_1, \ldots, s_n), E) \in S$,*

- *if $\exists j \in Id, (s_j, ?m, s_j') \in T_j$ then $(s, ?m, s') \in T$ with $s' = ((s_1, \ldots, s_j', \ldots, s_n), E \cup \{(m, u) \mid u \in SM_i(m)\})$;*   *(1)*
- *if $\exists j \in Id, (s_j, !m, s_j') \in T_j$ and $\pi^{uos}(E) \rightsquigarrow SM_i(m)$ then $(s, !m, s') \in T$ with $s' = ((s_1, \ldots, s_j', \ldots, s_n), E)$;*   *(2)*
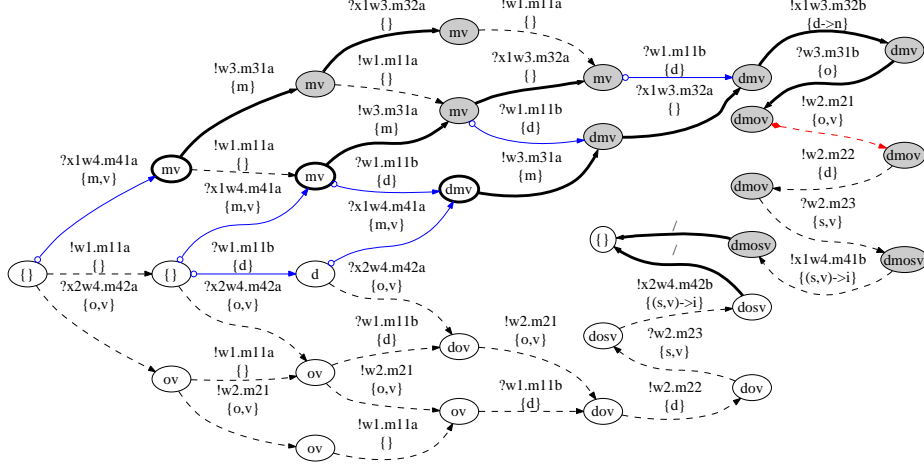
Fig. 4. The GA TIOLTS

- if $\exists j \in Id, (s_j, tau, s'_j) \in T_j$ then $(s, tau, s') \in T$ with $s' = ((s_1, \ldots, s'_j, \ldots, s_n), E)$;
- if $\forall j \in Id, (s_j, /, s'_j) \in T_j$ then $(s, /, s') \in T$ with $s' = ((s'_1, \ldots, s'_n), \emptyset)$;
- let $J = \{j \in Id \mid \exists (s_j, \chi, s'_j) \in T_j\}$, if $J \neq \emptyset$ then $(s, \chi, s') \in T$ with $s' = ((s'_1, \ldots, s'_j, \ldots, s'_n), E)$ where for every $j \in Id$, $(s_j, \chi, s'_j) \in T_j$ if $j \in J$ and $s'_j = s_j$ otherwise.

Postprocessing is performed on the GA removing recursively transitions leading to deadlock states (non-final states without outgoing transitions). In such a case, compatibility will only yield on a subset of partners' interactions. The alternative is to abort the adaptation process.

Example 8. The GA is given in Figure 4 where states are labelled with UoS initials (*e.g.*, d for dB) and transitions with messages and corresponding UoS initials (obtained by receptions and needed for emissions). To understand postprocessing, let us suppose Trans had not been available. The grey states would not have been computed and the three bold states would have been removed (in three steps), being deadlocks. Compatibility between PDA and its adaptor would then only yield for the lower (ogg) branch in Figure 1. Arrow shapes and colors are used in the sequel to explain step (iii).

The GA computation is exponential. Yet, additional criteria (wrt deadlock freedom) could be used to filter *on-the-fly* the GA during its computation and have a better complexity, as discussed in perspectives (Sect. 5).

3.3   Step iii: Generation of the Local Adaptors

The GA defines all valid interactions. Using it, we can generate for each partner ρ a *local adaptor* (hereafter adaptor for short) by extending its CSC ρ^c in three steps: (a) defining, for each message m sent by ρ^c to ρ, the set of potential messages from other CSC used to construct m, and receiving in ρ^c these

potential messages; (b) defining, for each message m received from ρ in ρ^c, the set of interested CSC that need UoS from m to send some message to their own partner, and sending in ρ^c the UoS to these interested CSC; (c) updating the ρ^c alphabet and operations. In the sequel, we are in the context of $\rho_i^c = \langle \omega_i = \langle O_i^\bullet, O_i^\circ, \mathcal{L}_i^c \rangle \rangle, (\mathcal{U}, \mathcal{R}), SM_i \rangle$, the CSC of a partner ρ_i in a partnership $\Upsilon_{Id} = \{\rho_{i,i \in Id}\}$ and with the associated GA $\mathcal{A}_{\Upsilon_{Id}} = \langle A, S, s_0, F, T \rangle$.

(a) Reception of required messages from other adaptors. For all m such that $!m \in A_i^c$, we define $Const_m$, the set of messages involved (directly or by means of $\mathcal{R}$) in the construction of m according to $\mathcal{A}_{\Upsilon_{Id}}$, as follows: $Const_m = \{?m' \in A \backslash A_i^c \mid \exists (s, \mathcal{H}) \in S, u \in \mathcal{U}, \text{ with } (m', u) \in \mathcal{H} \wedge \exists ((s, \mathcal{H}), !m, -) \in T \wedge u \in SM(m) \cup \underset{u' \in SM(m)}{\uplus} \mathcal{R}^{-1}(u')\}$. This corresponds to a form of GA backward analysis, from the sending of m by ρ_i^c to ρ_i, back to the messages that made the UoS (u) required for building m available. Yet, it is possible to look only one step back thanks to UoS union in Def. 9,(1). Note that when a UoS may come from several messages (and/or several CSC) then we non-deterministically choose one (*i.e.*, at most one application of (c2) in Def. 10 for each transition).

Example 9. Trans^c sends two messages to Trans: m31$_a$ (the initial call to Trans, requiring a mp3 UoS) and m32$_b$ (the sending of the noise UoS required by Trans to encode mp3 into ogg). In the GA (see the bullet/blue transitions), we see that the mp3 UoS for m31$_a$ comes from m41$_a$ ($Const_{w3.m31_a} = \{?x1w4.m41_a\}$) and that the noise UoS for m32$_b$ is obtained from a dB UoS which comes from m11$_b$ ($Const_{x1w3.m32_b} = \{?w1.m11_b\}$). See Table 2 in Appendix B for details on the construction of the other adaptors.

We define the ρ_i^c reception-extended alphabet as $\mathcal{AL}_i = A_i^c \cup \underset{!m \in A_i^c}{\uplus} Const_m$. From it, we can derive the expected behaviour of ρ_i^c. It corresponds to the ρ_i^c role in the GA, and therefore, to the projection of the GA on $\mathcal{AL}_i$. The projection must be selective, to avoid useless interactions. For example, let us consider a version of dBMeter (w1) with an operation (infodB[!m11$_b$]) that sends the dB UoS periodically. ?w1.m11$_b$ belongs to the Trans^c (w3^c) reception-extended alphabet. Yet, Trans^c should receive an instance of this message only if it is useful in a future state. Therefore, we first define for a TIOLTS $\langle A, S, s_0, F, T \rangle$ a precedence relation over states, $\preceq = \{(s, s') \mid (s, a, s') \in T\}^+$. Then we may define the projection that enables to retrieve the required receptions to be added in an adaptor.

Definition 10. *Let $\mathcal{A}_{\Upsilon_{Id}} = \langle A, S, s_0, F, T \rangle$ be a global adaptor TIOLTS of a partnership $\Upsilon_{Id} = \{\rho_{i,i \in Id}\}$ and $\mathcal{AL}_i \subseteq A$ be the alphabet of the ith (local) adaptor. The projection of $\mathcal{A}_{\Upsilon_{Id}}$ over $\mathcal{AL}_i$ is a TIOLTS $Pr(\mathcal{A}_{\Upsilon_{Id}})_{\mathcal{AL}_i} = \langle \mathcal{AL}_i \cup \{tau\}, S, s_0, F, T_{\mathcal{AL}_i} \rangle$ where $T_{\mathcal{AL}_i}$ is defined as $\forall (s, a, s') \in T$:*

- *(c1) if $a \in A_i^c$ then $(s, a, s') \in T_{\mathcal{AL}_i}$;*
- *(c2) if $a = ?m \in \mathcal{AL}_i \backslash A_i^c, \exists !m' \in A_i^c, \exists (s'', !m', -) \in T, a \in Const_{m'}, s' \preceq s''$, then $(s, ?\rho_i^c.m, s') \in T_{\mathcal{AL}_i}$ (prefixing by ρ_i^c ensures new private name)*
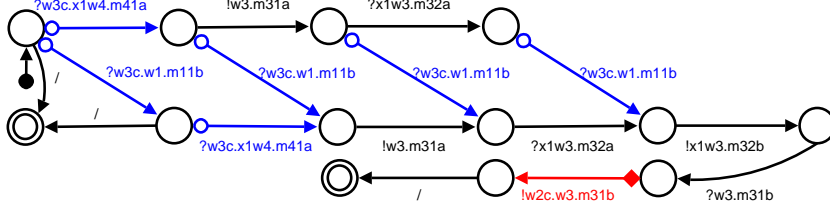- *(c3) otherwise, $(s, tau, s') \in T_{\mathcal{AL}_i}$ (removed using tau-reduction)*

Fig. 5. Trans Adaptor TIOLTS

Example 10. Projection may be explained on Trans^c using Figures 4 and 5 (without the diamond/red transition). Case c1 (bold transitions) corresponds to the original CSC messages. Case c2 (bullet/blue transitions) corresponds to receptions added in $\mathcal{AL}_i$, provided they are useful in the future, *e.g.*, not all $?\mathsf{w1}.\mathsf{m11}_b$ transitions are in this case. *Wrt* the GA, in projections case c2 labels are prefixed with the CSC identifier ($\mathsf{w3c}$ for Trans^c) to ensure private communication between adaptors. Finally, case c3 (dashed transitions) corresponds to non useful messages (*taus*). In the projection they are reduced.

(b) Emission of messages to interested adaptors. An adaptor ρ_i^c must forward any message m received from its partner ρ_i to the adaptors that need it. These may change for each instance of m. For each $(s, ?m, s') \in T_{\mathcal{AL}_i}$ such that $?m \in A_i^c$, we define a set of Interested Adaptors $IA_{(m,s)} = \{\rho_j^c | i \neq j \wedge \exists\ !m' \in !A_j^c$ with $m \in Const_{m'} \wedge \exists s''(s'', !m', -) \in T \wedge s' \preceq s''\}$. This corresponds to a GA forward analysis, from the reception of m by ρ_i^c, to the messages m' of other adaptors (ρ_j^c) that need an UoS in m ($m \in Const_{m'}$). For each $(s, ?m, s') \in T_{\mathcal{AL}_i}$ such that $?m \in A_i^c$, we add in the adaptor $Pr(A_{\Upsilon_{Id}})_{\mathcal{AL}_i}$ TIOLTS the interleaving of emissions to the elements of $IA_{(m,s)}$ as follows. Let $l = |IP(m,s)|$. First we add states, $S = S \cup \underset{j \in \{1, \ldots, l\}}{\times} \{s'_{0j}, s'_{1j}\}$, then, transitions, $T_{\mathcal{AL}_i} = T_{\mathcal{AL}_i} \setminus (s, ?m, s') \cup (s, ?m, s'_0 = (s'_{01}, \ldots, s'_{0l})) \cup \underset{j \in \{1, \ldots, l\}}{\times} (s'_{0j}, !\rho_j^c.m, s'_{1j})$. Finally, we unify s' and $(s'_{10}, \ldots, s'_{1l})$. This would correspond to flow branches in a BPEL implementation.

Example 11. Trans^c receives $\mathsf{m32}_a$ (no UoS) and $\mathsf{m31}_b$ (ogg) from its partner. We see in the GA that only HF^c (with $\mathsf{m21}$, reading an ogg file) is interested in $\mathsf{m31}_b$. The adding of emissions (diamond/red transitions) is demonstrated on the Trans adaptor (Fig. 5). Bullet/blue transitions were added in step iii(i). The final architecture with the exchanged messages is given in Figure 2 and the other adaptors in Appendix B.

(c) Updating the adaptors alphabet and operations. The initial alphabet of an adaptor ρ_i^c (A_i^c) is extended in previous steps with messages for receptions (the $?\rho_i^c.m$) and emissions (the $!\rho_j^c.m$). As far as operations are concerned, we proceed as follows. For each reception $?\rho_i^c.m$ added in step iii(a), we add in $O_i^\bullet$ an operation $op_{\rho_i^c.m}[?m]$. For each emission $!\rho_j^c.m$ added in step iii(b), we add in O_i° an operation $op_{\rho_j^c.m}[!m]$. The adding of receptions and emissions in CSCs is transparent for their partners thanks to prefixing by CSC identifiers. Moreover, the fact that the added emissions are deterministic (*i.e.*, leads to the same

state) preserves in adaptors the CSC correctness by construction. Our approach does not impose any constraint on resulting adaptors but for communication, data extraction from messages, and message construction (the latter two being supported by the Xpath part in matching functions), which makes our approach realistic for Web services adaptation.

4   Related Work

Behavioural component adaptation is now mature, but its application to SOA is recent. Mismatch patterns [4] or adaptation operators [15] may support adaptation but are not fully automatic as adaptation contracts must be defined. There are few fully automatic adaptation techniques. Service adaptation is performed using matching between service execution trees of BPEL processes in [8]. The matching process generates a workflow in an intermediary language (YAWL) which is then translated into BPEL. A form of adaptation may also be supported using ontology crossing [7]. Ontology matching is used in [20] to obtain behavioural correspondences and to compute a client-side adaptor that supports service replacement. Workflow analysis is used in [9] to build server-side adaptors that can be deployed as new services. All these approaches work with one client and (one or) several services, not system-wide. Yet, they build local adaptors and contrast with the centralized ones in most adaptation works [11, 19].

A technique for the distribution of a centralized orchestrator into different topologies of decentralized orchestrators is presented in [17] and extended in [12] to support data flow constraints and a filtering mechanism to select the topologies that satisfy the constraints. These techniques do not address adaptation and require a centralized orchestrator to be given while our goal is to obtain distributed orchestrators (in our case, adaptors) directly from the service descriptions. In [23, 1, 2], the authors extend their earlier works on component adaptation to support respectively incremental local adaptation and the distribution of centralized adaptors. They do not support the semantic interface level and therefore either require a mapping to be given or cannot deal with message name mismatch between services.

5   Conclusion

Automatic service composition at run-time is one of the most challenging issues, as it enables to provide the end-users with added-value functionalities composed out of services existing in their environment. Automatic service composition approaches usually assume that services have been previously developed to be integrated, and the proposed composition processes are limited to simple correspondences between service functionalities. To overcome these limitations, we have proposed an approach which integrates solutions from behavioural adaptation into the service composition process. The distinctive features of this approach are possibly complex correspondences between service functionalities, the

integration of descriptive semantics in the adaptation process, and distributed adaptors as opposed to a centralized one.

As explained before, the adaptation process is exponential due to the GA computation. To increase scalability, a first perspective concerns the *on-the-fly* filtering of the GA using a composition specification. This may be achieved by translating first this specification into an LTS and then taking this LTS into account in the GA computation [2, 19]. Filtering with data flow constraints [12] could also be used to enhance the GA computation.

We have made the hypothesis that services are equipped with semantic annotations. This common assumption in semantic (Web) services enables the automatic composition and adaptation. Yet, semantic annotation may be difficult and error prone and may lead to the impossibility to compose and adapt services correctly. This is a general issue in adaptation, *e.g.*, in semi-automatic adaptation where a mapping is used to replace the semantic annotations and where wrong mappings yield empty adaptors. The solutions that have been proposed consist in performing verification using the service and the adaptor models [19], or to build interactive tools that help building correct mappings [10]. While the later one is not compatible with automatic adaptation, the first one would, provided composition properties are given. The approach we have presented is compatible with this as it is based on related formal models (LTS).

Another perspective concerns the automation of adaptor implementation. Some results are available for centralized adaptors in BPEL [8] or WF/.NET [13]. An issue is to restrict the services from engaging in a forbidden communication. A solution could be to rely on additional middleware messages [1].

References

1. M. Autili, M. Flammini, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems. In *Proc. of EWSA'06*.
2. M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In *Proc. of ICSE'07*.
3. S. Ben Mokhtar, N. Georgantas, and V. Issarny. COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. *Journal of Systems and Software*, 80(12), 2007.
4. B. Benatallah, F. Casati, D. Grigori, H. R. Motahari Nezhad, and F. Toumani. Developing Adapters for Web Services Integration. In *Proc. of CAiSE'05*.
5. B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
6. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1), 2005.
7. A. Brogi, S. Corfini, J. F. Aldana, and I. Navas. Automated Discovery of Compositions of Services Described with Separate Ontologies. In *Proc. of ICSOC'06*.
8. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*.

9. A. Brogi and R. Popescu. Service Adaptation through Trace Inspection. *Int. J. Business Process Integration and Management*, 2(1):9–16, 2007.

10. J. Cámara, G. Salaün, and C. Canal. Clint: A Composition Language Interpreter. In *Proc. of FASE'08*.

11. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006. Special Issue on Software Adaptation.

12. G. Chafle, S. Chandra, V. Mann, and M. Gowri Nanda. Orchestrating Composite Web Services Under Data Flow Constraints. In *Proc. of ICWS'05*.

13. J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*.

14. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*.

15. M. Dumas, M. Spork, and K. Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proc. of BPM'06*.

16. S. Dustdar and W. Schreiner. A Survey on Web services Composition. *Int. J. Web and Grid Services*, 1(1):1–30, 2005.

17. M. Gowri Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In *Proc. of OOPSLA'04*.

18. S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling Web Services Interoperability. In *Proc. of ICEIS'04*.

19. R. Mateescu, P. Poizat, and G. Salaün. Behavioral Adaptation of Component Compositions based on Process Algebra Encodings. In *Proc. of ASE'07*.

20. H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*.

21. M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.

22. P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *Proc. of WCAT'04*.

23. P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*.

24. M. Tivoli, P. Fradet, A. Girault, and G. Goessler. Adaptor Synthesis for Real-Time Components. In *Proc. of TACAS'06*.

A   Structured Activities Semantic Rules

SOS rules for structured activities are given in Table 1 where $EH^d = [\{(m_i, P_i)_{i \in \{1,\ldots,n\}}\}, (d, Q), \{(e_j, R_j)_{j \in \{1,\ldots,m\}}\}]$.

$$\forall\, a \in Event \backslash \{/\}, \quad \frac{(P,\mathcal{H}) \xrightarrow{a} (P',\mathcal{H}')}{(P;Q,\mathcal{H}) \xrightarrow{a} (P';Q,\mathcal{H}')}$$

$$\forall\, a \in Event, \quad \frac{(P,\mathcal{H}) \xrightarrow{/} \text{ and } (Q,\mathcal{H}) \xrightarrow{a} (Q',\mathcal{H}')}{(P;Q,\mathcal{H}) \xrightarrow{a} (Q',\mathcal{H}')}$$

$$\forall i \in \{1,\ldots,n\}, \ (switch[\{(\_, P_i)_{i \in \{1,\ldots,n\}}\}], \mathcal{H}) \xrightarrow{tau} (P_i, \mathcal{H})$$

$$(while(\_, P), \mathcal{H}) \xrightarrow{tau} (P \ ; \ while(\_, P), \mathcal{H})$$

$$(while(\_, P), \mathcal{H}) \xrightarrow{tau} (empty, \mathcal{H})$$

$$\forall\, a \in Exc \bigcup \{tau\}, \quad \frac{\exists j \in \{1,\ldots,n\}, \ (P_j, \mathcal{H}) \xrightarrow{a} (P'_j, \mathcal{H}')}{(flow[\{P_{i,i \in \{1,\ldots,n\}}\}], \mathcal{H}) \xrightarrow{a} (flow[\{P_{i,i \in \{1,\ldots,n\}\backslash\{j\}}\} \bigcup \{P'_j\}], \mathcal{H}')}$$

$$\forall\, a \in Event \backslash \{\chi, /\}, \quad \frac{\exists j \in \{1,\ldots,n\}, \ (P_j, \mathcal{H}) \xrightarrow{a} (P'_j, \mathcal{H}')}{(flow[\{P_{i,i \in \{1,\ldots,n\}}\}], \mathcal{H}) \xrightarrow{a} (flow[\{P_{i,i \in \{1,\ldots,n\}\backslash\{j\}}\} \bigcup \{P'_j\}], \mathcal{H}')}$$

$$\frac{\forall i \in \{1,\ldots,n\}, \ (P_i, \mathcal{H}) \xrightarrow{/} (P'_i, \mathcal{H}'_i)}{(flow[\{P_{i,i \in \{1,\ldots,n\}}\}], \mathcal{H}) \xrightarrow{/} (terminate, \emptyset)}$$

$$\frac{\exists J \neq \emptyset, \ J \subseteq \{1,\ldots,n\}, \ \forall i \in J, \ (P_i, \mathcal{H}) \xrightarrow{\chi} (P'_i, \mathcal{H}'_i) \text{ and } \forall i \in \{1,\ldots,n\} \backslash J, \ (P_i, \mathcal{H}) \xrightarrow{/}}{(flow[\{P_{i,i \in \{1,\ldots,n\}}\}], \mathcal{H}) \xrightarrow{\chi} (flow[\{P'_i\}_{i \in J} \bigcup \{P_i\}_{i \in \{1,\ldots,n\} \backslash J}], \bigcup_i \mathcal{H}'_i)}$$

$$\frac{(P,\mathcal{H}) \xrightarrow{/}}{(scope(P,EH), \mathcal{H}) \xrightarrow{/} (terminate, \emptyset)}$$

$$\frac{(P,\mathcal{H}) \xrightarrow{\chi} (P',\mathcal{H}')}{(scope(P,EH^d), \mathcal{H}) \xrightarrow{\chi} ((scope(P,EH^{d-1}), \mathcal{H}')} \text{ if } d > 1$$

$$\frac{(P,\mathcal{H}) \xrightarrow{\chi} (P',\mathcal{H}')}{(scope(P,EH^1), \mathcal{H}) \xrightarrow{\chi} (Q, \mathcal{H}')}$$

$$\forall i \in \{1,\ldots,m\}, \forall (?m_i, P_i) \in EH \quad \frac{\forall a \in Exp \bigcup \{tau,/\}, \ (P \not\xrightarrow{a})}{(scope(P,EH), \mathcal{H}) \xrightarrow{?m_i} (P_i, \mathcal{H} \cup SM(m_i))}$$

$$\forall j \in \{1,\ldots,m\}, \forall (e_j, R_j) \in EH \quad \frac{(P,\mathcal{H}) \xrightarrow{e_j} (p', \mathcal{H}')}{(scope(P,EH), \mathcal{H}) \xrightarrow{tau} (R_j, \mathcal{H}')}$$

$$\forall e \in Exc, (e, -) \notin EH, \quad \frac{(P,\mathcal{H}) \xrightarrow{e}}{(scope(P,EH), \mathcal{H}) \xrightarrow{e} (empty, \mathcal{H}')}$$

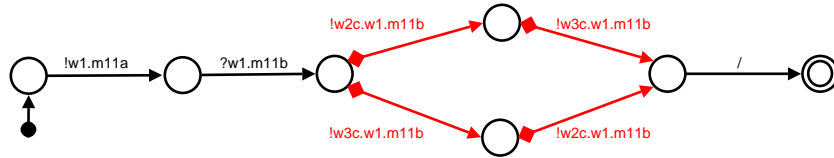**Table 1.** Operational Semantics for Structured Activities

B   Additional Figures and Tables

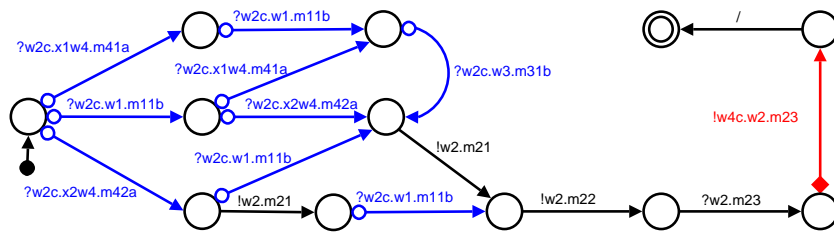| partner	message	required UoS	candidate provider messages (partner)
w1	$m11_a$	$\emptyset$	$\emptyset$
w2	m21	{o, v}	{$m42_a$ (w4), $m41_a$ (w4), $m31_b$ (w3)}
	m22	{d}	{$m11_b$ (w1)}
w3	$m31_a$	{m}	{$m41_a$ (w4)}
	$m32_b$	{d} ({d}$\rightsquigarrow$n)	{$m11_b$ (w1)}
w4	$m41_b$	{s, v} ({s, v}$\rightsquigarrow$i)	{m23 (w2)}
	$m42_b$	{s, v} ({s, v}$\rightsquigarrow$i)	{m23 (w2)}

Table 2. Dependency between messages (used in step iii(a))

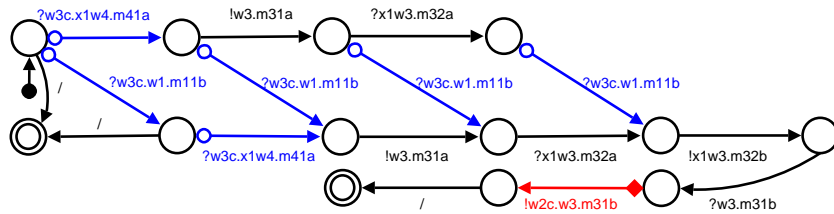| partner	message	provided UoS	candidate requester messages (interested partner)
w1	$m11_b$	{d}	{m22 (w2), $m32_b$ (w3)}
w2	m23	{s, v}	{$m41_b$ (w4), $m42_b$ (w4)}
w3	$m32_a$	$\emptyset$	$\emptyset$
	$m31_b$	{o}	{m21 (w2)}
w4	$m41_a$	{m, v}	{m21 (w2), $m31_a$ (w3)}
	$m42_a$	{o, v}	{m21 (w2)}

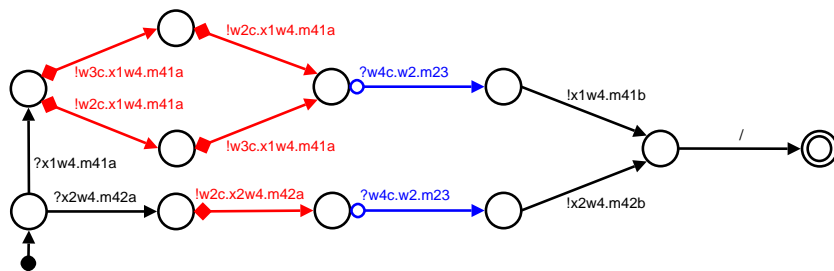**Table 3.** Interested partners (used in step iii(b))

dBMeter Adaptor TIOLTS

HF Adaptor TIOLTS

Trans Adaptor TIOLTS

PDA Adaptor TIOLTS