2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

Generation of Service Wrapper Protocols from Choreography Specifications

Gwen Salaün
Department of Computer Science, University of Málaga, Spain
Email: salaun@lcc.uma.es

Abstract

Choreography description languages specify interactions among a set of services from a global point of view. From this description, it is possible to generate either an orchestrator (centralized interactions), or a set of peers or wrappers (distributed interactions). In this paper, we present first a model of service protocols with value passing, and an abstract choreography language to describe their composition and adaptation. Adaptation is useful while composing services to correct existing mismatches which might exist between their interfaces. Given abstract descriptions of services and their choreography, we propose techniques based on encodings into process algebra to generate an orchestrator and a set of wrapper protocols. Generation of wrappers is particularly tackled in this paper because this enables the system deployment in the context of distributed systems, and keeps at the same time a full parallelism of the system execution. Our approach is completely automated by a prototype tool we implemented.

1 Introduction

Service-oriented computing has emerged as a new programming paradigm that aims at designing and implementing software applications which interact via the exchange of messages. The specification of the service composition or *choreography* is often given from a global point of view since it makes easier its writing and validation. From a choreography, either an *orchestrator* can be generated and deployed in a centralized way (on one specific machine), or various peers or *wrappers* can be generated and distributed on the different machines which hold the services involved in the system, or deployed using middleware technologies.

Services are accessed through their interfaces that distinguish four interoperability levels: signature, behaviour or protocol, quality of service and semantics. Here, we consider that services are described using signatures and protocols. Protocols give the application order of method calls and exchanged messages. We particularly focus on these behavioural descriptions because they are crucial to avoid erroneous executions of the system such as deadlocks.

In this paper, we propose an approach to generate service wrapper protocols from an abstract specification of the choreography. The choreography language we rely on makes possible to define connectors between services, but also more complex composition policies by specifying order, choice or iteration of connectors. The set of distributed peers are implemented as service wrappers. The wrapper protocols (as well as the central orchestrator protocol) are obtained in two steps. First, service interfaces, the choreography specification and some constraints indicating how to distribute it are encoded into the LOTOS process algebra [15]. In a second step, wrapper protocols are computed from this encoding using state-of-the-art exploration and reduction techniques that avoid the generation of the full state space corresponding to the LOTOS specification. Our approach is completely automated by a prototype tool we implemented and validated on many examples.

The wrapper generation makes possible their deployment in the context of distributed systems, and keeps at the same time a full parallelism of the system's execution. It also facilitates its implementation by splitting its behavioural description into pieces (the behaviour of a centralized orchestrator can be quite complex when many services are involved). Once the wrapper protocols are generated, they can be implemented using BPEL or Windows Workflow Foundation [9].

The choreography language and orchestrator generation techniques we present here go beyond service connectors specification because they also take *adaptation* into account. Software adaptation [3] aims at correcting mismatches that appear while reusing existing services, and have to be solved without modifying the service's code. Mismatch situations may be caused when message names do not correspond, the order of messages is not respected, a message in one service has no counterpart or matches with several messages, exchanged parameters arrive in different order or along different messages, etc.

The remainder of the paper is structured as follows. Section 2 presents our model of service interfaces. Section 3

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.42

IEEE computer society

describes the choreography specification language. In Sections 4 and 5, we formalise the LOTOS encoding which allows the generation of central orchestrator and wrapper protocols. Section 6 introduces our prototype tool which completely automates the orchestrator and wrapper generation. Finally, Section 7 compares our approach to related work, and Section 8 ends the paper with some conclusions.

2 Model of Services

In this section, we present our model of service interfaces. We assume that service interfaces are given using both a signature and a behavioural interface (or protocol).

Definition 1 (Signature) *A signature Σ is a set of provided and required operation profiles. An operation profile is the name of an operation, along with its argument types (possibly empty), and its return types (possibly empty):*

$$op : ti_1 * \ldots * ti_n \rightarrow to_1 * \ldots * to_m$$

Signatures usually correspond in component-based frameworks (*e.g.*, .NET or J2EE) to operation profiles described using an Interface Definition Language (IDL). WSDL is the accepted standard in the Web services area.

Furthermore, we propose that behavioural interfaces are represented by means of *Symbolic Transition Systems* (STSs). In this paper, STSs are Labelled Transition Systems (LTSs) extended with value passing (data parameters coming with messages). Communication between services is represented using events relative to the emission (!) and reception (?) of messages corresponding to operation calls. Events may come with a set of data terms whose types respect the operation signatures.

Definition 2 (Label) *In our model, a label is either the internal action τ or a tuple (M, D, PL) where M is the message name, D stands for the direction (!,?), and PL is either a list of data terms, if the message corresponds to an emission, or a list of variables, if the message is a reception.*

Definition 3 (STS) *A Symbolic Transition System (STS) is a tuple (A, S, I, F, T) where: A is an alphabet which corresponds to the set of labels associated to transitions, S is a set of states, $I \in S$ is the initial state, $F \in S$ are final states, and $T \in S \times A \times S$ is the transition function (see [14] for semantic aspects).*

This formal model has been chosen because it is simple, graphical, and can be easily derived from existing implementation platforms' languages, see for instance [11, 23, 10, 9] where such abstractions for Web services were used for verification, composition or adaptation purposes. In some cases, for conciseness reasons for example, a textual notation is better than a graphical one. Thus, a process algebra with value passing could be used as a higher level language to specify behavioural interfaces. STSs can be automatically obtained from these processes using the operational rules of the process algebra semantics.

Example. Throughout the paper, we will use as running example a multi-function device service to illustrate the different steps of our approach. In Figure 1, we give the signatures and behavioural interfaces of the different services used in our example. Let us present first the device that may perform several functions (print, scan, copy). This service can receive a file (file?id,f with a person identifier and the file as parameters) and an action (action?a) as many times as required, and use them as input to the multi-function device. At some point, the device can receive a termination message (halt?), send an invoice (invoice!id,inv with the person identifier and the corresponding invoice as parameters), and the amount to pay (price!am). The bank service can receive an invoice (invoice?y) and a payment (payment?id coming with the person identifier) in any order. Last, we introduce a client to give a full description of one transaction for the different involved services. In this specific case, our client can connect (connect!id), and submit file (file!f) and action (action!x) several times until (s)he decides to stop (halt!). Then, (s)he receives the price (price?p) and pays for it (pay!id,p with its identifier and the price to pay) in sequence. In the client protocol, tau transitions stand for an internal choice made by the user.

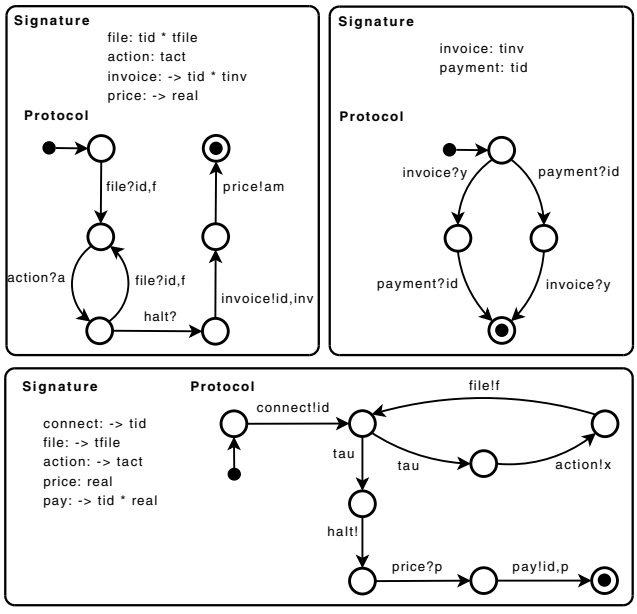

Figure 1. Service interfaces: (left) multi-function device, (right) bank, (bottom) client

3 Choreography Specification Language

In this section, we present our abstract choreography language that allows to specify interactions (composition) and how to work out mismatch situations (adaptation). We rely on *synchronization vectors* [1] (or vector for short). They express correspondences between messages, like bindings between ports or connectors in architectural descriptions. Each event appearing in a vector is executed by one service and the overall result corresponds to an interaction between all the involved services. A vector may involve any number of services and does not require interactions occurring on the same names of events. Furthermore, variables are used in events as placeholders for message parameters. The same variable name appearing in different events (possibly in different vectors) enables one to relate sent and received message parameters. Vectors can be either written by hand or obtained from a graphical description of the architecture built by the designer.

Definition 4 (Vector) *A vector for a set of service interfaces* $(\Sigma_i, (A_i, S_I, I_i, F_i, T_i))$, $i \in \{1, .., n\}$, *is a tuple* $\langle e_1; \ldots; e_n \rangle$ *where* e_i *is a label term for* A_i *or* $\{\varepsilon\}$, ε *meaning that the service does not participate in this synchronization. A label term* t *contains the name of the operation, a direction, and as many untyped fresh names as parameters in the argument type list. The prefixing of messages by service identifiers can be used in vectors in complement to* ε *omission in order to yield a digest notation.*

In this work, we assume a *synchronous* communication model: two (or more in case of broadcast) entities synchronize on one event (rendez-vous) and continue their own evolution. Asynchronous communication could be modelled describing message queues using additional services which interact synchronously with the services they represent.

Vectors are not sufficient to describe more advanced composition scenarios such as contextual rules, choice between vectors or ordering between them. The order in which vectors have to be applied can be specified using different notations such as regular expressions, Labeled Transition Systems (LTSs), or (Hierarchical) Message Sequence Charts. Due to their readability and user-friendliness, we chose to specify these additional constraints using *vector LTSs*, that is, LTSs whose labels are vectors (Fig. 2).

Definition 5 (Choreography specification) *A choreography specification for a set of service interfaces* $(\Sigma_i, (A_i, S_i, I_i, F_i, T_i))$, $i \in \{1, .., n\}$, *is a couple* (V, L) *where* V *is a set of vectors, and* L *is a vector LTS for* V.

If only message name correspondences are necessary to solve service mismatch, the vector LTS may leave the vector application order unconstrained using a single state and all vector transitions looping on it. In particular, this pattern can be used on specific parts of the vector LTS for which the designer does not want to impose any ordering.

Example. Let us go back to our multi-function device example. First, we specify the architecture of our system by defining the following vectors which connect messages and parameters included in the different service signatures:

$$
\begin{array}{lcl}
v_{conn} & = & \langle \mathsf{c:connect!ID} \rangle \\
v_{file} & = & \langle \mathsf{md:file?ID, F; c:file!F} \rangle \\
v_{act} & = & \langle \mathsf{md:action?A; c:action!A} \rangle \\
v_{halt} & = & \langle \mathsf{md:halt?; c:halt!} \rangle \\
v_{inv} & = & \langle \mathsf{md:invoice!ID, INV; b:invoice?INV} \rangle \\
v_{price} & = & \langle \mathsf{md:price!P; c:price?P} \rangle \\
v_{pay} & = & \langle \mathsf{b:payment?ID; c:pay!ID, P} \rangle
\end{array}
$$

where for instance vector v_{file} means that message file? that appears in the multi-function device interface will match with the file! message in the client interface. In addition, placeholders relate parameters in both interfaces using new variables; in this vector, the client identifier and the file parameter are respectively denoted using ID and F. In some cases, data connections are transversal to vector definitions (ID as an example is used in various vectors).

Vectors are not enough to describe advanced composition constraints. Imagine for instance that during busy hours the device restricts its access and limits the number of requests sent by clients to two. Figure 2 gives the vector LTS defining these constraints. The limit in the number of files sent by the client is specified by a sequence of two applications of vector v_{file}. There are two final states which mean that the client can submit files once or twice. All the other vectors appear as looping transitions on states because they can be applied at any moment without restriction.

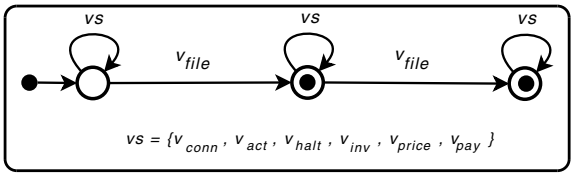

Figure 2. Choreography specif.: vector LTS

Our choreography language is also able to solve mismatch situations that occur between interfaces. This is the case for instance with vector v_{conn} where $connect!$ in the client does not have any correspondence in the other services, or with vector v_{pay} where a signature mismatch (different message names) has to be worked out. Vectors also solve the various mismatches at the parameter level.

4 Orchestrator Generation

An orchestrator for a set of services is an STS running in parallel with the service STSs and guiding their execution (all exchanged messages pass through the orchestrator). In this way, if adaptations are needed, mismatches are compensated by the orchestrator and the ordering of messages imposed by the choreography specification is guaranteed. Generating orchestrator protocols is a complicated task since the orchestrator has to respect the choreography specification taking into consideration behavioural constraints of services formalised into their interfaces (STSs). In addition, protocols may generate many interleaved interactions that we want to preserve so as to accept all the possible execution orders of messages.

In this work, we chose the LOTOS process algebra [15] to encode the systems' composition constraints (interfaces and choreography). Indeed, LOTOS is expressive enough *wrt.* the problem at hand, and is equipped with CADP [13] a toolbox which implements optimised state space exploration techniques as well as numerous verification tools. The LOTOS encoding enables the automatic generation of orchestrator protocols whose traces represent all possible (correct) interactions between services. To do so, our approach employs *on-the-fly* algorithms to increase, *w.r.t.* existing approaches, the efficiency of the orchestrator generation and reduction process by avoiding the generation of the full state space. The LOTOS encoding also enables the verification of the orchestrator protocol by using model checking tools available in CADP.

In this section, we rely on the approach presented in [17] to encode LTSs into LOTOS, and extend it to take value passing (in STSs and vectors) into account.

Service STS encoding. Each state $s \in S$ of a service STS $sv = (A, S, I, F, T)$ is encoded as a process sv_s with as many branches as there are transitions outgoing from s. An additional branch, using a specific `FINAL` action, models termination when s is final ($s \in F$). As far as the encoding of STS labels into LOTOS is concerned, we want to distinguish sent and received messages with a "`_EM`" and "`_REC`" suffix. In addition, LOTOS symbols ! and ? are used to support data transfer (resp. emission and reception). In our context, the correct distribution will be ensured by the encoding of the choreography constraints (see the next step in this section), therefore all service STS labels that involve value passing (emission or reception of parameters) are translated into LOTOS with a question mark followed by as many fresh variables as there are parameters coming with the message. Since these variables are placeholders, their LOTOS type can simply be an arbitrary one that we call `PH`. This type is defined beforehand using the LOTOS abstract datatype facilities with all the placeholder names appearing in vectors defined as type constructors. Every label encoding is followed by a call to the LOTOS process encoding the target state of the transition being translated.

```
process sv_s [gates(sv, A), FINAL] : func(sv, s) :=
  enc(sv, l_1); sv_s_1 [gates(sv, A), FINAL]
  [] ... []
  enc(sv, l_m); sv_s_m [gates(sv, A), FINAL]
  [ [] FINAL; exit ]
endproc
```

where $A = \{l_1, \ldots, l_m, \ldots, l_n\}$, and $\{(s, l_1, s_1), \ldots, (s, l_m, s_m)\} = \{t \in T \mid \textit{source}(t) = s\}$, $source$ returns the source state of a transition, and $enc(sv, m!) = sv_m_\texttt{EM}$, $enc(sv, m?) = sv_m_\texttt{REC}$, $enc(sv, m!e_1, \ldots, e_n) = sv_m_\texttt{EM}?x_1 : \texttt{PH}, \ldots, ?x_n : \texttt{PH}$, and $enc(sv, m?y_1, \ldots, y_n) = sv_m_\texttt{REC}?y_1 : \texttt{PH}, \ldots, ?y_n : \texttt{PH}$. Function $gates$ returns the alphabet for the LOTOS process by extracting gates as follows: $gates(sv, A) = \{gates_l(sv, l) \mid l \in A\}$, $gates_l(sv, m!) = gates_l(sv, m!e_1, \ldots, e_n) = sv_m_\texttt{EM}$, and $gates_l(sv, m?) = gates_l(sv, m?y_1, \ldots, y_n) = sv_m_\texttt{REC}$. Function $func$ states whether the process terminates or not (see [17] for its computation).

Choreography specification encoding. A choreography specification (V, L), with $L = (A_C, S_C, I_C, F_C, T_C)$, is encoded by generating (i) a process for each state in the vector LTS L, (ii) a process for each vector in V, and (iii) the interleaving of all these vector processes. The correct ordering of vectors is ensured by the vector LTS thanks to two actions for each vector v. A first one (`run_`v) activates the corresponding vector process. A second one (`rel_`v) releases the vector LTS and enables it to apply several vectors at the same time. The vector LTS (i) is encoded using the same pattern as service STSs, that is every state is encoded as a LOTOS process. For each transition in the vector LTS, the new "`run_`" and "`rel_`" actions are generated in sequence.

```
process vLTS_s [A_L] : func(L, s) :=
  run_v_1 ; rel_v_1 ; vLTS_s_1 [A_L]
  [] ... []
  run_v_m ; rel_v_m ; vLTS_s_m [A_L]
  [ [] FINAL; exit ]
endproc
```

where $A_C = \{v_1, \ldots, v_n\}$, $A_L = (\bigcup_{v \in A_C} \{\texttt{run_}v, \texttt{rel_}v\}) \cup \{\texttt{FINAL}\}$, and $\{(s, v_1, s_1), \ldots, (s, v_m, s_m)\} = \{t \in T_C \mid \textit{source}(t) = s\}$.

Vector processes (ii) are first launched through a "`run_`" interaction with the vector LTS. Next, they communicate with services on all actions appearing in their vector definition. They have to receive the sent messages before beginning to emit some. There is no specific ordering between emissions in a vector process. When a vector process executes a vector, it must be ready to interact with the service STSs on their emissions. Then, several strategies are possible to release the vector (`rel_`v), and therefore to execute the services' receptions. A first option is to wait for the

complete processing of a vector before firing a new one. Another strategy is to execute the release action once all the emissions executed: the receptions are run after this release, and meanwhile the vector LTS can launch another vector. This behavior makes the reordering of messages possible, a typical case of mismatch between services.

As regards value passing, an auxiliary LOTOS process `Store` is generated to store information about the availability of received values. Every time some values are sent by a service, they are received by one of the vector processes and stored by using the (global) process `Store`, which makes them available at the level of the orchestrator. This availability is essential, because when service receptions in a vector are being run (emissions at the level of the orchestrator), this firing is conditioned by the availability of the values to be emitted. Thus, every service emission in a vector is followed by an interaction with the process `Store` to set to *true* the availability of the received values, and every service reception in a vector is preceded by some interactions with the `Store` process to check that the required values have been received. In the latter case, the vector process may have to wait the availability of the needed resources. Such an active waiting is encoded using a looping process (v`_wait`) that terminates once the data are available. If they are never available, this will generate a deadlock in the underlying state space that will be cut away in a second step by our reduction techniques (see further in this section).

```
process Store [A_St] (vs : VarStore) : exit :=
  store_v1?x : VarStore; Store [A_St] (append(x, vs))
  []
  read_v1?x : VarStore; reply_v1!allin(x, vs);
      Store [A_L] (vs)
  [] ... []
  store_vn?x : VarStore; Store [A_St] (append(x, vs))
  []
  read_vn?x : VarStore; reply_vn!allin(x, vs);
      Store [A_L] (vs)
  [] FINAL; exit
endproc
```

where A_{St} = {`store_`v_1, `read_`v_1, `reply_`v_1, ..., `store_`v_n, `read_`v_n, `reply_`v_n}, `VarStore` is a datatype defined using abstract datatype LOTOS facilities, operation `append` adds a set of variables as available in the store, and operation `allin` tests availability of a set of variables.

Each vector v is encoded as follows:

```
process vector_v [A_v] : exit :=
 run_v; ( (e1!x1!..!xi; store_v!cs(x1, .., xi); exit) |||
  .. ||| (ek!xj!..!xp; store_v!cs(xj, .., xp); exit) ) >>
   rel_v; ( (read_v!cs(y1, .., yi); reply_v?b : Bool;
    v_wait [A_v] (b, cs(y1, .., yi)) >> r1!y1!..!yi; exit) |||
     .. ||| (read_v!cs(yj, .., yq) .. >> rm!yj!..!yq; exit) )
        >> vector_v [A_v]
  [] FINAL; exit
endproc
```

where A_v = {`run_`v, `rel_`v, `FINAL`} $\cup$ $\{e_1, \ldots, e_k, r_1, \ldots, r_m\}$, $em(v) = \{e_1, \ldots, e_k\}$, $rec(v) = \{r_1, \ldots, r_m\}$, functions em and rec are defined as $em(\langle l_1, \ldots, l_n \rangle) = \{enc(m!) \mid l_i = m!v_1, \ldots, v_k \vee l_i = m!\}$ and $rec(\langle l_1, \ldots, l_n \rangle) = \{enc(m?) \mid l_i = m?x_1, \ldots, x_k \vee l_i = m?\}$. Expression `cs`$(x_1, \ldots, x_p)$ is a simplified notation for `cons`$(x_1, $`cons`$(x_2, \ldots, $`cons`$(x_p, $`nil`$)\ldots))$.

Process v`_wait` tests if variables are available, and if not, it starts an active waiting (successive "`read_`" and "`reply_`" interactions) with the store until variables are available.

```
process v_wait [A_v] (b : Bool, x : VarStore) : exit :=
  [b] -> exit
  []
  [not(b)] ->
      read_v!x; reply_v?b : Bool; v_wait [A_v] (b, x)
endproc
```

Finally, vector processes (iii) are interleaved since they do not communicate together. All the vector processes may synchronize with the `Store` process to store new available variables, or check the availability of some variables to be sent. The store process starts without any variable (`nil`).

```
process vectors [A_V, A_St] : exit :=
  ( vector_v1 [A_v1] ||| ... ||| vector_vn [A_vn] )
  | [A_St] |
  Store [A_St] (nil)
endproc
```

where $A_C = \{v_1, \ldots, v_n\}$, $A_V = \bigcup_{v \in A_C} A_v$, and $A_{St} =$ {`store_`v_1, `read_`v_1, `reply_`v_1, ..., `reply_`v_n}.

System encoding. In this step, we generate a LOTOS process corresponding to the whole system's constraints (LOTOS processes encoding the service STSs and the choreography specification), and respecting the desired system architecture (orchestrator in-the-middle, intercepting all messages). This means that the service STSs only interact together on `FINAL` (correct termination is when all services terminate) while they interact with vectors on actions used in their alphabets. The synchronization between vector processes and vector LTS has been described earlier on (using "`run_`" and "`rel_`" actions). In addition, all actions that are not messages appearing in the involved services (*e.g.*, "`run_`" and "`rel_`" actions, or all interactions with the `Store` process) are hidden as they represent internal actions of the orchestrator. They will be removed by reduction steps when generating the orchestrator from the LOTOS code.

```
process central_orchestrator [A_CX, A_L] : exit :=
  hide A*_L in
  ( (sv1_I_sv1 [gates(sv1, A_sv1), FINAL]
    | [FINAL] | ... | [FINAL] |
    svn_I_svn [gates(svn, A_svn), FINAL] )
  | [A_CX, FINAL] |
  (vLTS_I_C [A_L] | [A_L] | vectors [A_V]) )
```

endproc

where $A_L^* = A_L \backslash \{\texttt{FINAL}\}$ and $A_{CX} = \bigcup_{i \in \{1,\ldots,n\}} enc(A_{sv_i})$.

The LOTOS encoding is automated by Compositor, a tool we implemented originally for LTS interfaces [17] and extended (*wrt.* the encoding presented above) to take value passing into account. From this encoding, the orchestrator STS is generated using Scrutator [17], a tool that removes remaining erroneous paths, τ transitions and path similarities by applying state-of-the-art exploration and reduction techniques while avoiding the complete state space generation. Finally, we generate an SVL [12] script to reverse message directions and obtain the resulting orchestrator.

Example. We present in Figure 3 the beginning of the centralized orchestrator protocol in which only vectors are considered as connectors between services (no vector LTS). Its full description was generated using Compositor and Scrutator, and contains 27 states and 34 transitions. All the messages involved in its description are reversed to make synchronizations with the services possible. In this piece of orchestrator, reordering of messages is required. Thus, the orchestrator starts by receiving a sequence of messages from the client (c:connect?ID, c:action?A, and c:file?F). Next, several evolutions are possible. Let us focus on the scenario in which the orchestrator delivers the file (md:file!ID,F), as specified in vector v_{file}, and the action (md:action!A) to the multi-function device. Then, the orchestrator may receive a termination message (c:halt?) or another action (c:action?A) from the client. If the client decides to stop (c:halt?), the orchestrator forwards this message to the multi-function device (md:halt!), and so on. The

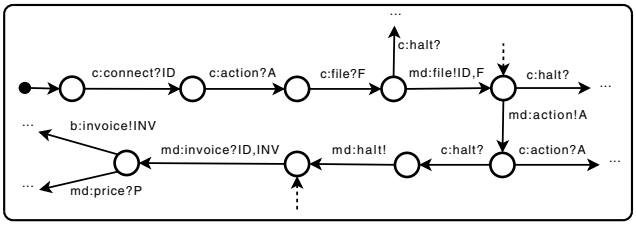

Figure 3. Orchestrator protocol (piece of) for the multi-function device example (vectors)

orchestrator protocol built with respect to the vector LTS introduced in Figure 2 contains 46 states and 57 transitions.

5 Wrapper Generation

In the context of distributed systems, the generation of wrappers can be necessary in order to implement the interaction constraints described in the choreography specification. Moreover, it allows to preserve a full parallelism of the system's execution. In the following, we distinguish two kinds of entities: services and wrappers. Services implement the system's functionality, and are the primary computational constituents of a system. Wrappers are local orchestrators, and there is one for each service involved in the system. They receive all messages emitted by their service, and route them with respect to constraints specified in the choreography. Then, a system is defined as a set of services each of them directly connected to its local wrapper. Each wrapper is connected to its service and to the services involved in the system and with which it will interact (Fig. 4).

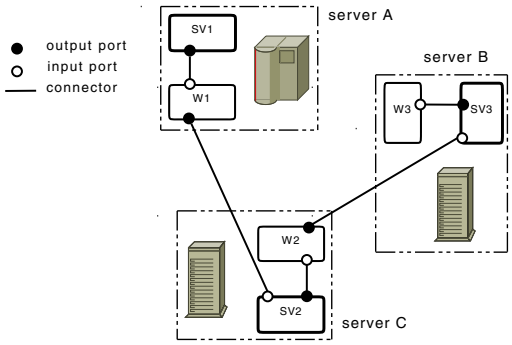

Figure 4. Simple choreography architecture

In addition, in case of complex composition constraints (for instance if some interactions must be applied in a precise order), wrappers have to synchronize on specific points. More precisely, each wrapper is enhanced with additional synchronizations involving all the wrappers that make them respect the overall choreography specification (Fig. 5).

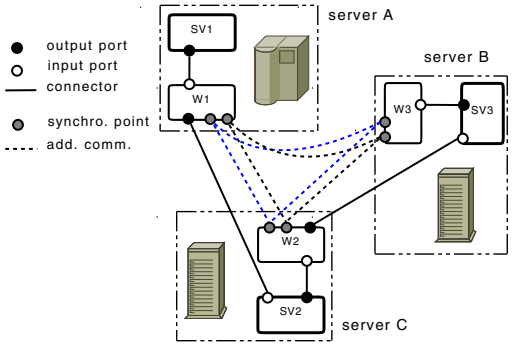

Figure 5. Complex choreography architecture

In order to automatically generate wrapper protocols we extend the LOTOS encoding presented in Section 2. There are two cases. First, if the choreography specification only involves correspondences between messages (vectors), wrappers are obtained by focusing on one service after the other, and by restricting the wrapper behaviour to mes-

sages going out of its services and messages matching to these emissions in the corresponding vectors. For example, in vector $v_{inv} = \langle \mathsf{md:invoice!ID, INV; b:invoice?INV} \rangle$, the emission is issued by the multi-function device service, thus its wrapper will contain all messages appearing in v_{inv} (with reversed directions). Our approach assumes that all vectors contain one emission.

Second, in case the choreography specification involves ordering (not only vectors but also a vector LTS), wrapper protocols must also take into account additional messages, those that are used to run and release vectors (*i.e.*, `run_`v and `rel_`v). However, we do not keep all these messages, but only those that impose an order in the vector application. Hence, these additional communications correspond to transitions of the vector LTS that connect two different states, all the transitions looping on a same state are discarded. For these remaining vectors, wrappers synchronize on both "`run_`" and "`rel_`" messages so as to preserve the application order of vectors as specified in the vector LTS.

Last but not least, some variable scope issues may occur when a service receives a specific data value directly from another wrapper and further sends it to another service. In this situation, the wrapper is asked to emit a value that it has never received. To detect that each variable is received before being sent, we automatically generate mu-calculus temporal formulas for each variable V involved in a wrapper (using the pattern $[(\texttt{not}\ '.*\texttt{_EM}.*!V.*')*.('.*\texttt{_REC}.*!V.*')]\ \texttt{false}$) and check them using the Evaluator [18] model checker. If such an issue is detected, we launch again the wrapper generation by imposing that every message received by a service is also received by the wrapper (see emission from W1 to both W2 and SV2 in Fig. 6). This solution induces more communications, therefore it is adopted only when scope problems are detected.

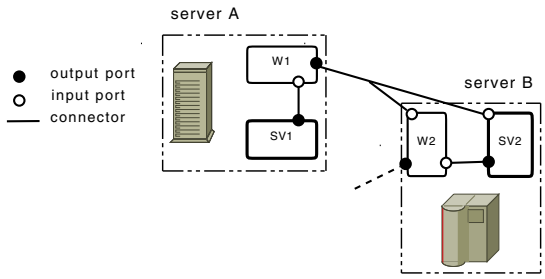

Figure 6. Architecture: variable scope issues

In the following of this section, we formalise the generation of wrapper protocols, and we illustrate it on our running example. However, for space reasons, we will not introduce our solutions to variable scope issues, although they are implemented in the prototype tool we present in Section 6.

5.1 Generation of Service Wrappers

This section deals with the case in which wrappers evolve independently of each other (only vectors in the choreography specification) without adding synchronization points. For each service STS $sv = (A, S, I, F, T)$, a wrapper is generated from the description of the LOTOS central orchestrator by hiding all messages which should not be handled by the wrapper. Accordingly, the wrapper only preserves messages appearing in vectors in which the emission belongs to the service at hand.

```
process wrapper_sv [A_CX, A_L] : func(sv, I) :=
  hide A_w in central_orchestrator [A_CX, A_L]
endproc
```

where $A_w = (A_{CX} \backslash A_{obs})$, $A_{CX} = \bigcup_{i \in \{1,\ldots,n\}} gates(sv_i, A_{sv_i})$, $A_{obs} = \{gates_l(sv, l_{i,i\in\{1,\ldots,n\}}) \mid \langle l_1, \ldots, l_n \rangle \in V, \exists i \in \{1,\ldots,n\}, l_i \in A, em(l_i)\}$, $A_L = (\bigcup_{v \in A_C} \{\texttt{run_}v, \texttt{rel_}v\}) \cup \{\texttt{FINAL}\}$, $A_C = \{v_1, \ldots, v_n\}$.

Beyond this LOTOS code, we also generate an SVL script which calls Scrutator to generate each wrapper protocol from its LOTOS process, and reverses message directions in these protocols.

Example. We show in Figure 7 the wrapper generated using our approach (generated files contain 493 lines of LOTOS and 114 lines of SVL respectively) for the multi-function device service of our running example with only vectors used as choreography specification. The wrapper is able to catch all emissions coming from its service (invoice!id,inv and price!am), and routes these messages with respect to their matching as specified in vectors v_{inv} and v_{price}. Therefore, the invoice message is forwarded to the bank, and the price to the client to inform him of the amount to pay. As far as the receptions in the device interface are concerned, they match with emissions coming from bank and client wrappers. For instance, the client wrapper catches file and action coming from its service, and sends these information to the multi-function device service.

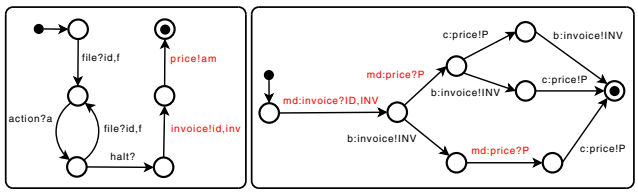

Figure 7. Multi-function device service: interface (left) and wrapper (right) protocols

5.2 Wrapper Synchronization

This section deals with the case in which wrappers must interact together in order to ensure the choreography specification. To do so, wrapper generation is extended to take into account additional communications. We use the "`run_`" and "`rel_`" messages introduced in Section 2 to control the firing of vectors with respect to their order in the vector LTS. All these messages are not preserved, only those for vectors involved in a sequence in the vector LTS are kept.

`process wrapper_`sv `[`A_{CX}, A_L`]` : $func(sv, I)$ `:=`
  `hide` A'_w `in central_orchestrator [`A_{CX}, A_L`]`
`endproc`

where $A'_w = (A_{CX} \backslash A_{obs}) \cup A_{loop}$, A_{CX}, A_{obs} and A_L are computed as presented in Section 5.1, and $A_{loop} = \{\{\texttt{run_}v, \texttt{rel_}v\} \mid v \in V_{loop} \backslash V_{trans}\}$ with $V_{loop} = \{v \mid (s, v, s) \in T_C\}$ and $V_{trans} = \{v \mid (s, v, s') \in T_C, s \neq s'\}$.

Similarly to Section 5.1, an SVL script allows to automate the generation of protocol wrappers corresponding to the LOTOS encoding.

Example. Figure 8 presents a piece of the client wrapper and the bank wrapper (both obtained from generated files containing 552 lines of LOTOS, and 121 lines of SVL). The full description of the client wrapper contains 32 states and 35 transitions. Compared to the former section in which no additional synchronizations were used, here the wrapper does more than just taking care of its service since it may also synchronize on `run_`v and `rel_`v messages with the other wrappers. In our example, these additional communications only happen for vector v_{file}, which expresses the exchange of files between the client and device services. This can be visualized in the vector LTS (Fig. 2) in which only two transitions with label v_{file} appear in sequence whereas all the other ones loop on a same state meaning that their application order does not matter.

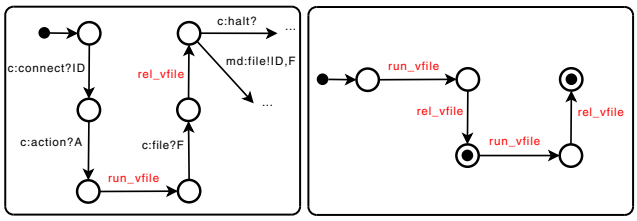

Figure 8. Client wrapper protocol (piece of, left), and bank wrapper protocol (right)

6 Prototype Tool

The different steps of our approach have been implemented in a prototype tool called DCompositor which generates LOTOS code for service STSs, the choreography specification, and wrappers. SVL scripts are also generated to automate the computation of wrappers from the LOTOS code by calling Scrutator. We present in Figure 9 an overview of the tools, and of their input and output formats (aut and bcg respectively stand for a textual and a computer representation for state/transition models).

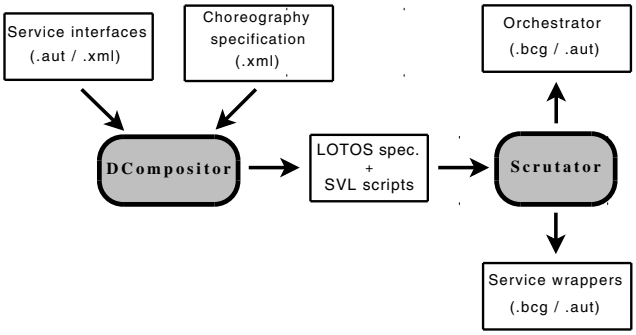

Figure 9. Tool support overview

We applied and validated our proposal on about 200 examples, some of which were reused from previous works on adaptor generation [7, 17]. Table 1 shows experimental results on some of the examples belonging to our database. For each experiment, the table gives the size of the service STSs, the use ($\surd$) or not ($\times$) of a vector LTS in the choreography specification, the size of the centralized orchestrator protocol and of the wrapper protocols. As far as services and wrappers are concerned, we use in the table "/" to separate the state and transition numbers, for instance broadcast-007 involves 5 services, and 5 wrappers are generated.

First of all, in the case where only vectors are given as choreography specification, the distribution does not generate an overhead in the number of states and transitions compared to the centralized orchestrator (see for instance emuseum-007 or mf-device-006). Furthermore, with vectors only, this distribution sometimes diminishes drastically the size of the wrappers (see broadcast-007 and mail-system-002); indeed, wrappers may avoid message interleavings made explicit in centralized orchestrators. If more advanced composition constraints are required (vector LTS), we may observe a slight increase in the number of states and transitions in the distributed approach compared to the centralized one (see for example mf-device-007 or rate-service-001). This increase is reasonable since we have only kept in the design the additional communications that impose an order between messages in the vector LTS.

As far as correctness is concerned, we take advantage of the equivalence checking techniques available into CADP to check the correctness of our proposal. SVL code is automatically generated to verify that the centralized system (centralized orchestrator protocol + service interfaces) is

Case study	Services		vLTS	Orchestrator		Wrappers	
	states	trans.		states	trans.	states	trans.
broadcast-007	5/5/5/5/5	4/4/4/4/4	×	91	201	4/4/3/4/6	3/3/2/3/6
emuseum-004	10/10/7	12/13/9	√	121	202	24/76/15	29/111/14
emuseum-007	10/10/7	12/13/8	×	56	82	4/47/4	4/69/4
flight-booking-001	9/5/5	10/7/6	×	33	34	20/8/8	21/7/7
mail-system-002	7/9	9/16	×	150	381	78/8	192/21
mf-device-006	6/8/4	6/8/4	×	27	34	9/18/2	10/21/1
mf-device-007	6/8/4	6/8/4	√	46	57	21/32/7	24/35/6
pc-store-002	9/10	10/11	×	16	16	11/8	11/8
rate-service-001	13/11/6	16/12/7	√	28	32	31/28/18	37/37/18
sql-server-005	6/3	8/6	√	38	51	34/16	45/19
vod-006	3/5	6/10	√	17	22	7/11	12/18

Table 1. Case studies: orchestration versus wrapper generation

equivalent to the distributed one (wrapper protocols + service interfaces). To do so, the SVL script first generates the centralized system using the following expression:

("sv_1.bcg" | [FINAL] | ... | [FINAL] | "sv_n.bcg")
| [A_{CX}] |
"central_coordinator.bcg"

where A_{CX} is computed as presented in Section 5.1. Next, the distributed system is generated as follows:

("sv_1.bcg" | [FINAL] | ... | [FINAL] | "sv_n.bcg")
| [A_{CX}] |
("wrapper_sv_1.bcg"
| [As] | ... | [As] | "wrapper_sv_n.bcg")

where $As = \{\texttt{FINAL}\} \cup \{\{\texttt{run_}v, \texttt{rel_}v\} \mid v \in V_{trans}\}$ with $V_{trans} = \{v \mid (s, v, s') \in T_C, s \neq s'\}$ if a vector LTS is used, otherwise $As = \{\texttt{FINAL}\}$.

In a last step, the Bisimulator tool [4] of the CADP toolbox is called, and it enabled us to check that strong equivalence [21] is preserved between centralized and distributed systems for all the examples of our database.

7 Related Work

The contribution we have presented in this paper is a solution to the *realizability* challenge that the authors state in [24], namely how to turn a choreography into service implementations automatically. In [6], the authors define models for choreography and orchestration, and formalise a conformance relation to connect both models. These models are assumed given as input whereas we focus on the generation of both (orchestrator and implementations) from a global specification while ensuring conformance. In [20], the authors show through a simple example how BPEL stubs can be derived from WS-CDL choreographies. However, due to the lack of semantics of both languages, correctness of the generation cannot be ensured. In some recent papers [22, 16], formal languages to describe choreographies have been proposed. Conformance with respect to an orchestration specification and implementability issues have been studied from a formal point of view. In [8], the authors propose a language based on π-calculus and session types to formally describe choreographies. Then, they identify three principles for global description under which they define a sound and complete end-point projection, that is the generation of end-point processes from the choreography description. Compared to these approaches, we have presented a tool-supported approach (lack of all the papers mentioned above) to automatically generate wrappers that ensure conformance with respect to their global description. The main difference with these works is that they focus on the peer generation assuming no services exist whereas we suppose some service implementations are reused, and wrappers aim at constraining the functionality of these existing services to make them respect the choreography.

In the area of Software Adaptation, [2, 19] aim at distributing behavioural adaptors (orchestrators able to correct incompatibilities between components). In [2], the authors start with a central description of an adaptor, and split this protocol into pieces to distribute it on the involved components. This approach mainly suffers an overhead in the number of messages. Indeed, many additional communications are added to keep components aware of the evolution of the others, and to fire their own evolution. Moreover, remaining erroneous paths are kept in the description of adaptors, and they are avoided dynamically by using controllers which introduce other additional messages. In comparison, our approach minimises the number of additional messages which remains reasonable as we have showed by experimental results in Section 6. In [19], the authors propose an approach to generate automatically the composition of semantic services, therefore they do not only focus on signature and behavioural descriptions of entities, but also take semantic information into account. This work addresses

fully-automatic adaptation as no mapping is required since it uses semantic annotations. Nevertheless, a central protocol is fully computed before distributing it. The specification of system properties such as what we do using our vector LTS is presented as an extension of their algorithm.

8 Conclusion

In this paper, we have presented an approach to generate wrapper protocols from behavioural descriptions of services and an abstract specification of the choreography. To do so, we have encoded into LOTOS the different inputs of our system. In a second step, exploration and reduction techniques have been used to generate wrapper protocols from the LOTOS code. Our proposal is completely automated by tools we implemented and applied to many examples.

As regards future works, we would like to consider other choreography languages (*e.g.*, collaboration diagrams [5] or the chor calculus [22]), and study how our wrapper generation approach can be extended to deal with them. Similarly to [20] and as sketched in [9], we also plan to generate from wrapper protocols some BPEL wrappers that would be in charge of the involved services and make services interact as specified in the choreography.

Acknowledgements. The author thanks Javier Cámara, José Martin, and Pascal Poizat for interesting discussions and comments on a former version of this paper. This work has been supported by project TIN2007-67134 funded by the Spanish Ministry of Innovation and Science, and project P06-TIC2250 funded by the Andalusian local Government.

References

[1] A. Arnold. *Finite Transition Systems*. International Series in Computer Science. 1994.

[2] 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*, volume 4344 of *LNCS*, pages 17–32. Springer-Verlag, 2006.

[3] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, chapter Towards an Engineering Approach to Component Adaptation. Springer-Verlag, 2006.

[4] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585. Springer-Verlag, 2005.

[5] T. Bultan and X. Fu. Specification of Realizable Service Conversations Using Collaboration Diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.

[6] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration Conformance for System Design. In *Proc. of Coordination'06*, volume 4038 of *LNCS*, pages 63–81. Springer-Verlag, 2006.

[7] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*, pages 63–77. Springer-Verlag, 2006.

[8] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer-Verlag, 2007.

[9] 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*, volume 215 of *ENTCS*, pages 39–55. Elsevier, 2007.

[10] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.

[11] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.

[12] H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. FORTE'01*, pages 377–392. IFIP, Kluwer Academic Publishers, 2001.

[13] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–163. Springer-Verlag, 2007.

[14] A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, pages 427–478. Handbook of Process Algebra. Elsevier, 2001.

[15] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.

[16] J. Li, H. Zhu, and G. Pu. Conformance Validation between Choreography and Orchestration. In *Proc. of TASE'07*, pages 473–482. IEEE Computer Society, 2007.

[17] R. Mateescu, P. Poizat, and G. Salaün. Behavioral Adaptation of Component Compositions based on Process Algebra Encodings. In *Proc. of ASE'07*, pages 385–388. IEEE Computer Society, 2007.

[18] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Sci. Comput. Programming*, 46(3):255–281, 2003.

[19] T. Melliti, P. Poizat, and S. B. Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In *Proc. of FASE'08*, volume 4961 of *LNCS*, pages 146–162. Springer-Verlag, 2008.

[20] J. Mendling and M. Hafner. From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *Proc. of OTM'05 Workshops*, volume 3762 of *LNCS*, pages 506–515. Springer-Verlag, 2005.

[21] R. Milner. *Communication and Concurrency*. PH, 1989.

[22] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*, pages 973–982. ACM Press, 2007.

[23] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *IJBPIM*, 1(2):116–128, 2006.

[24] J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a Theory of Web Service Choreographies. In *Proc. of WS-FM'07*, volume 4937 of *LNCS*, pages 1–16. Springer-Verlag.