Realizability and Dynamic Reconfiguration of Chor Specifications

Nima Roohi
Sharif University of Technology, Tehran, Iran
E-mail: roohi@ce.sharif.edu

Gwen Salaün
INRIA Grenoble - Rhône-Alpes / VASY France
E-mail: gwen.salaun@inria.fr

Keywords: choreography, realizability, dynamic reconfiguration, process calculus, labeled transition systems

Received: January 6, 2010

Choreography description languages aim at specifying from a global point of view interactions among a set of services involved in a new system. From this specification, local implementations or peers can be automatically generated. Generation of peers that precisely implement the choreography specification is not always possible: this problem is known as realizability. When peers corresponding to this specification are being executed we may want to modify the choreography specification and reconfigure dynamically the system. This is the case for instance if we add or remove interactions due to the addition of functionalities to the system at hand or the loss of a service. In this article, we present our solutions to check if a choreography is realizable and if a specific reconfiguration can be applied or not.

Povzetek: Opisana je metoda preverjanja možnosti implementacije sistema na osnovi opisa.

1 Introduction

A choreography describes how a set of services interact together from a global point of view. Several formalisms have already been proposed to specify choreographies: WS-CDL, collaboration diagrams, process calculi (such as Chor), BPMN, SRML, etc. Choreography specification, correctness, realizability and implementation are crucial issues in Service Oriented Computing. Several works aimed at studying and proposing solutions to the realizability problem [7, 18, 4, 2, 20] that consists in checking if a set of existing peers implements a choreography. In this article, we first present some techniques to check realizability of choreographies. Next, we focus on the dynamic reconfiguration of a choreography which has been distributed and deployed. Such reconfigurations correspond to the addition or removal of some interactions (loss of a service, extension of the functionalities, substitution of a service, etc.).

We use the Chor calculus [18] as choreography specification language, because it is an abstract model of WS-CDL coming with a formal syntax and semantics (not the case of WS-CDL). Our goal here is first to check the realizability of a choreography. To do so, we propose an encoding of Chor into the FSP process algebra and reuse equivalence checking tools to verify that the behaviors of both systems (centralized and distributed) are the same. Next, we formalize a reconfigurability test that checks if a set of peers that have been obtained from a choreography, can be reconfigured with respect to a second choreography specification which consists in an extension (addition of some interactions) or a simplification (removal of some interactions) of the original choreography. If these reconfigurations are possible, new peers are generated and replace the former ones. In addition, we also propose some analysis techniques to check some properties on the reconfiguration, *e.g.,* if modifications coming from the new choreography specification impact current peer behaviors only after their current execution state. Finally, if a choreography is realizable or can be reconfigured, we can automatically generate Java code for the corresponding peers for rapid prototyping purposes.

The rest of this article is organized as follows: Section 2 introduces Chor, Peer, and FSP, respectively as our choreography, peer, and intermediate languages. Section 3 presents some automatic techniques to first convert choreographies to an intermediate language, and then to check whether this choreography is realizable or not. In Section 4, we present our approach to check if some reconfigurations specified as a new choreography can be applied or not. We also present some techniques to analyze the impact of reconfigurations. In section 5 we describe our prototype tool, and comment on some experimental results. Also, we briefly overview code generation for peers. Section 6 compares our approach to related works, and Section 6 ends the article with some concluding remarks.

2 Preliminaries: Chor, Peer, and FSP

2.1 Chor and Peer

Chor [18] is a simple process language, and a simplified model of WS-CDL, for describing peers from a global point of view. From this global specification, behavioral specifications of peers can be generated by projection. In this section, we will overview both the Chor language (global view) and the Peer language (local view) introduced in [18].

Table 1 shows the syntax and semantics of Chor (C, C_1 and C_2 are arbitrary Chor specifications). It uses weak traces (τ actions are hidden) for specifying its semantics (where $[\![C]\!]$ stands for the weak trace set of C). The reader interested in more details on the language may refer to [18]. Also, operators on sets of traces which are used in Table 1 have been formally defined in [19].

The loop operator "$*$" has the highest priority among the others. After that, priority of the sequential composition operator ";" is higher than the other operators, as an example, $*C_1 \sqcap C_2; C_3$ is not ambiguous. Priority of parallel "||" and choice "$\sqcap$" operators is equal, as an example, $C_1 \| *C_2 \sqcap C_3 = (C_1 \| (*C_2)) \sqcap C_3$ (left associativity).

Chor is implemented by the coordination of a set of independent processes. The Peer language is a simple calculus for describing these processes. In this language, ε is an empty process which means do nothing, and for an arbitrary trace t if $P \stackrel{t}{\Longrightarrow} \varepsilon$ we have $t \in [\![P]\!]$ (we use $\dagger$ to denote deadlock). Table 2 gives the syntax and semantics of the Peer language (P, P_1 and P_2 are arbitrary Peer specifications).

The Peer language mainly differs from Chor by the description of interactions. Peer specifies them from a local point of view. Therefore, at the Peer level, an interaction activity is either an emission or a reception, and peers interact together by handshake communication (same channels, opposite directions).

Using rules defined in Table 2, trace sets of Peer processes are obtained as follows:

$$\frac{P \xrightarrow{\sigma} P'}{P \stackrel{\sigma}{\Longrightarrow} P'} \qquad \frac{P \xrightarrow{\sigma} P' \quad P' \stackrel{\sigma'}{\Longrightarrow} P''}{P \stackrel{\sigma^\frown\sigma'}{\Longrightarrow} P''}$$

Last, operator $/ : Peer \times Activity \to Peer$ returns the process obtained after executing the activity which is specified as the second input parameter of the "$/$" operator, and function fst (abbreviation for *first*) : $Peer \to \mathbb{P}(Activity)$, in which $\mathbb{P}(Activity)$ is the power set of all possible activities, computes activities of a Peer process which can be executed first. Formal definitions of operator "$/$" and function fst are as follows ($\perp$ denotes an undefined process):

$\mathsf{fst}(\alpha) \widehat{=} \{\alpha\}$

$\mathsf{fst}(\varepsilon) = \mathsf{fst}(skip) = \mathsf{fst}(P_1 \sqcap P_2) = \mathsf{fst}(*P) \widehat{=} \emptyset$

$\mathsf{fst}(P_1; P_2) \widehat{=} \mathsf{fst}(P_1) \quad \mathsf{fst}(P_1 \| P_2) \widehat{=} \mathsf{fst}(P_1) \cup \mathsf{fst}(P_2)$

$$skip/\alpha \widehat{=} \perp \qquad \alpha/\alpha' \widehat{=} \begin{cases} \varepsilon & \text{if} \quad \alpha = \alpha' \\ \perp & \text{if} \quad \alpha \neq \alpha' \end{cases}$$

$$(P_1; P_2)/\alpha \widehat{=} P_1/\alpha; P_2 \qquad (P_1 \sqcap P_2)/\alpha = (*P)/\alpha \widehat{=} \perp$$

$$(P_1 \| P_2)/\alpha \widehat{=} \begin{cases} P_1/\alpha \| P_2 & \text{if} \quad \alpha \in \mathsf{fst}(P_1) \\ P_1 \| P_2/\alpha & \text{if} \quad \alpha \in \mathsf{fst}(P_2) \\ \perp & \text{else} \end{cases}$$

Example. We will use throughout this article a metal stock market as running example. There are three peers in our example. First, peer Broker selects one of two metals, namely iron and steel, then look at the market as many times as needed until a sale on the selected metal becomes available. Broker sends his/her bid on the selected metal to the second peer (Market) of our example. After receiving a bid, Market performs the following two tasks concurrently: saving the bid in its own database, and checking to see if this bid is better than the best current one or not. Then, Market sends the result of this check and the name of the broker to the announcement Board (third peer of our example). If this bid is the best so far, Board will change the current winner and notifies the broker. Otherwise, Board does nothing (skip). In the Chor specification below, bk, mk, bd respectively stand for Broker, Market, and Board:

$$\begin{aligned} &\mathsf{Stock} = \\ &\quad (\mathsf{iron}^{\mathsf{bk}} \sqcap \mathsf{steel}^{\mathsf{bk}}); \mathsf{look}^{\mathsf{bk}}; *\mathsf{look}^{\mathsf{bk}}; \mathsf{bid}^{[\mathsf{bk},\mathsf{mk}]}; \\ &\quad (\mathsf{save}^{\mathsf{mk}} || \mathsf{check}^{\mathsf{mk}}); \mathsf{result}^{[\mathsf{mk},\mathsf{bd}]}; \\ &\quad (\mathsf{change}^{\mathsf{bd}}; \mathsf{notify}^{[\mathsf{bd},\mathsf{bk}]} \sqcap \mathsf{skip}) \end{aligned}$$

2.2 FSP

FSP is a process calculus that takes inspiration in Milner's Calculus of Communicating Systems (1980) and in Hoare's Communicating Sequential Processes (1985), as explained by Magee and Kramer in [12]. FSP was originally designed for distributed software architecture specification, and distinguishes sequential and composite processes. Table 3 introduces FSP operators which are used in the rest of this article (x, y, *new*, and *old* are actions, P and Q are FSP processes).

3 Realizability of Chor specifications

3.1 Translating Chor into FSP

There are two main solutions in order to perform the realizability check automatically: (i) generate and compare sets of traces for Chor and Peer in an *ad-hoc* manner, or (ii) translate Chor and Peer to some intermediate language and use existing tools to compare their behaviours. We prefer the second solution because it enables the designers to take advantage of existing tools such as equivalence checking to verify realizability, or model-checking tools for validation and verification purposes. We chose FSP because it relies on a simple language yet expressive enough to encode Chor operators. Moreover, FSP is equipped with the LTSA toolbox which provides efficient tools for state space exploration and verification. This encoding allows

Table 1: Syntax & Semantics of Chor

$skip$	means do nothing, its trace set is equal to $\{\langle\rangle\}$
a^i	is an arbitrary local activity performed by peer i, and its trace set is $\{\langle a^i\rangle\}$
$c^{[i,j]}$	is a communication between two peers i (sender) and j (receiver) through channel c, its trace set is $\{\langle c^{[i,j]}\rangle\}$
$C_1;C_2$	means first C_1 and then C_2, $[\![C_1;C_2]\!] = [\![C_1]\!] \frown [\![C_2]\!]$
$C_1 \sqcap C_2$	means either C_1 or C_2, $[\![C_1 \sqcap C_2]\!] = [\![C_1]\!] \cup [\![C_2]\!]$
$C_1 \| C_2$	means C_1 and C_2 run concurrently, $[\![C_1 \| C_2]\!] = [\![C_1]\!] \bowtie [\![C_2]\!]$
$*C$	means execute C an arbitrary number of times, $[\![*C]\!] = [\![C]\!]^*$

Table 2: Syntax & Semantics of Peer

P	::=	BP	(basics)	BP	::=	$skip$	(no action)
	|	$P;P$	(sequential)		|	a	(local)
	|	$P \sqcap P$	(choice)		|	$c!$	(send)
	|	$P \| P$	(parallel)		|	$c?$	(receive)
	|	$*P$	(loop)				

Skip: $skip \xrightarrow{\langle\rangle} \varepsilon$ **Local:** $a \xrightarrow{\langle a\rangle} \varepsilon$

Sequential: $\dfrac{P_1 \xrightarrow{\sigma} P_1'}{P_1;P_2 \xrightarrow{\sigma} P_1';P_2}$ $\quad \varepsilon;P \xrightarrow{\langle\rangle} P$

Choice: $P_1 \sqcap P_2 \xrightarrow{\langle\rangle} P_1$ $\quad P_1 \sqcap P_2 \xrightarrow{\langle\rangle} P_2$

Parallel: $\varepsilon \| \varepsilon \xrightarrow{\langle\rangle} \varepsilon$

$$\frac{P_1 \xrightarrow{\sigma} P_1'}{P_1 \| P_2 \xrightarrow{\sigma} P_1' \| P_2} \qquad \frac{c! \in \mathsf{fst}(P_1) \quad c? \in \mathsf{fst}(P_2)}{P_1 \| P_2 \xrightarrow{\langle c\rangle} P_1/c! \| P_2/c?}$$

$$\frac{P_2 \xrightarrow{\sigma} P_2'}{P_1 \| P_2 \xrightarrow{\sigma} P_1 \| P_2'} \qquad \frac{c? \in \mathsf{fst}(P_1) \quad c! \in \mathsf{fst}(P_2)}{P_1 \| P_2 \xrightarrow{\langle c\rangle} P_1/c? \| P_2/c!}$$

Loop: $*P \xrightarrow{\langle\rangle} skip$ $\quad *P \xrightarrow{\langle\rangle} P;*P$

to: (i) validate and verify Chor specifications using the LTSA toolbox, (ii) generate peer protocols from its choreography specified in Chor, (iii) test for realizability of the Chor specification, and (iv) generate Java code from FSP for rapid prototyping purposes. One could decide to specify choreographies and peers directly using FSP. However, domain-specific languages such as Chor and Peer are more adequate to write such specifications, since they provide the exact level of expressiveness to do so.

Basic activities are translated into simple FSP processes with one transition from the source to the final state (we use τ for the *skip* action). The Chor sequential operator is encoded using the FSP sequential operator. As regards the choice operator, we prefix each operand by a τ transition, therefore similarly to the Chor language, selecting a choice operand is performed non-deterministically. In the FSP parallel operator, actions which are in alphabets of both operands can only evolve through synchronization, but the Chor parallel operator does not synchronize activities of its operands (interleaving). Consequently, we first prefix operands of each parallel operator with a unique value, thus no synchronization occurs. Then, we use the renaming operator of FSP to replace these new action names with their original values. The loop operator $*C$ is specified in FSP using a non-deterministic choice between performing *skip*, or performing C and then a recursive call to the FSP process that encodes the loop operator.

Definition 1 (Chor into FSP)**.** *Encoding a Chor specification C into FSP is achieved using function* $\mathsf{c2f} : Chor \rightarrow FSPdescription$*, as presented in Figure 1 ("$\backslash$" operator hides actions in the FSP process, "/" operator renames actions in the FSP process, and* $\mathsf{ac}(C)$ *returns non-skip basic activities of its Chor operand).*

FSP does not allow actions to have subscript or superscript. Therefore, we respectively translate a^i and $c^{[i,j]}$ into a_i and c_i_j. $\mathsf{c2f}_{pi}$ is a one-to-one function of type $Chor \rightarrow ProcessIdentifier$ generating fresh identifiers (the same ones for identical Chor specifications) as output, which obey naming rules[1] of FSP process identifiers. $T.\mathsf{c2f}_{pi}$ returns a process identifier which is obtained by prefixing the result of $\mathsf{c2f}_{pi}$ by T. For all C and C' such that $\mathsf{c2f}(C)$ has a process identifier $\mathsf{c2f}_{pi}(C')$ in its specification, the result of $\mathsf{c2f}(C')$ must be included in the result of $\mathsf{c2f}(C)$, because whenever we use one FSP identifier in our specification, we must include the specification of that process in our final specification. We proved that this translation preserves the semantics of the Chor language [19].

[1]These rules are defined in Section 2 of Appendix B in [12].

Table 3: FSP Operators and Informal Semantics

$(x\text{->}P)$	describes a process that initially executes action x and then behaves as P.
$(P;Q)$	describes a process that first behaves as P, and then (after completion of P) behaves as Q.
$(x\text{->}P\|y\text{->}Q)$	describes a process that either executes action x and then P, or action y and then Q.
$(P\|\|Q)$	represents the concurrent execution of P and Q. This operator synchronizes shared actions of P and Q.
$x:P$	prefixes each label in the alphabet of P with x.
$P/\{new_1/old_1, \ldots, new_n/old_n\}$	renames action labels. Each old label in P is replaced by the new one.
$P\backslash\{x_1,\ldots,x_n\}$	removes action names $x_1,\ldots,x_n$ from the alphabet of P and makes these actions "silent". These silent actions are labeled by τ. Silent actions in different processes are not shared.
$P@\{x_1,\ldots,x_n\}$	hides all actions in the alphabet of P which do not belong to the set $\{x_1,\ldots,x_n\}$.

Figure 1: Encoding Chor into FSP

$\mathsf{c2f}(skip)$	$\widehat{=}\ SKIP = (skip\text{->}END)\backslash\{skip\}.$
$\mathsf{c2f}(a^i)$	$\widehat{=}\ \mathsf{c2f}_{pi}(a^i) = (a\_i\text{->}END).$
$\mathsf{c2f}(c^{[i,j]})$	$\widehat{=}\ \mathsf{c2f}_{pi}(c^{[i,j]}) = (c\_i\_j\text{->}END).$
$\mathsf{c2f}(C_1;C_2)$	$\widehat{=}\ \mathsf{c2f}_{pi}(C_1;C_2) = \mathsf{c2f}_{pi}(C_1);SKIP;\mathsf{c2f}_{pi}(C_2);END.$
$\mathsf{c2f}(C_1 \sqcap C_2)$	$\widehat{=}\ \mathsf{c2f}_{pi}(C_1 \sqcap C_2) = (z\text{->}\mathsf{c2f}_{pi}(C_1);END\|z\text{->}\mathsf{c2f}_{pi}(C_2);END)\backslash\{z\}.$ assuming z is neither in the alphabet of $\mathsf{c2f}_{pi}(C_1)$ nor $\mathsf{c2f}_{pi}(C_2)$.
$\mathsf{c2f}(C_1\|C_2)$	$\widehat{=}\ \|T.\mathsf{c2f}_{pi}(C_1\|C_2)=(p1:\mathsf{c2f}_{pi}(C_1)\|p2:\mathsf{c2f}_{pi}(C_2)).$ $\mathsf{c2f}_{pi}(C_1\|C_2)=T.\mathsf{c2f}_{pi}(C_1\|C_2);SKIP;END/$ $\{ba_1/p1.ba_1\|ba_1\in\mathsf{ac}(C_1)\}\cup\{ba_2/p2.ba_2\|ba_2\in\mathsf{ac}(C_2)\}.$
$\mathsf{c2f}(*C)$	$\widehat{=}\ \mathsf{c2f}_{pi}(*C) = (z\text{->}SKIP;END\|z\text{->}\mathsf{c2f}_{pi}(C);SKIP;\mathsf{c2f}_{pi}(*C))\backslash\{z\}.$ assuming z is not in the alphabet of $\mathsf{c2f}_{pi}(C)$.

Example. Let us illustrate our encoding with some of the FSP processes generated for our example. In Table 4 we can see for instance how the choice operator is performed non-deterministically by prefixing the choice's operands by z and then hiding this action. Figure 2 shows the minimized LTS, obtained by compilation with LTSA, of the generated FSP code ($\mathsf{c2f}_{pi}(\mathsf{Stock})$). First, Broker decides what metal (s)he wants, iron or steel. Then, (s)he looks at the market as many times as needed until a sale on the selected metal becomes available (there is a loop on state 2 in the LTS). After that, (s)he sends his/her bid to the market. Next, Market saves the price and checks it, concurrently (there are two different paths from state 4 to state 6 in the LTS). Then, Market sends the result of the performed check to the board. Finally, Board either does nothing (if the result says the bid was not good enough), or changes itself and notifies the broker (if the result says the bid was the best one so far). This LTS was run several times using LTSA animation techniques, and the system behaved as expected. Model-checking was not required here because we chose a simple example in this article for the sake of comprehension.

3.2 Peer Generation

Given a Chor specification, one can generate the specification of each Peer using *natural projection*. Natural projection[2] of a Chor specification to Peer P first replaces each observable action with *skip* iff P does not perform that action. Chor and Peer share parallel, sequential, choice, and loop operators. For these operators the natural projection replaces each Chor operator by its equivalent in Peer, and applies recursively to their operands. Projection of basic activities from a Chor specification C to a Peer specification P is achieved as follows:

1. each activity not performed by P is replaced by *skip*,
2. a local activity performed by P remains unchanged,
3. a communication activity involving P is replaced by a channel input activity (if P is the receiver) or a channel output activity (if P is the sender).

Generation of FSP processes for an arbitrary Chor specification is performed using function c2f, defined previously in this section. The behavior of each Peer P in the choreography C is generated by hiding in the corresponding FSP ($\mathsf{c2f}_{pi}(C)$) all actions to which P does not participate (Definition 2).

Definition 2. *Given a Chor specification C and a Peer identifier p, the FSP process corresponding to* $\mathsf{nproj}(C,p)$*, the natural projection of the Chor specification C to the Peer p, is generated as follows (*$\mathsf{p2f}_{pi}$ *is defined similarly to* $\mathsf{c2f}_{pi}$*):*

[2]The reader may refer to [18] for the formal definition of natural projection.

Table 4: Some FSP Processes Generated for the Running Example

Chor Specification	FSP Process Specification
skip	SKIP = (skip->END)\{skip}.
iron^{bk}	Iron_bk = (iron_bk->END).
look^{bk}	Look_bk = (look_bk->END).
$\text{bid}^{[\text{bk,mk}]}$	Bid_bk_mk = (bid_bk_mk->END).
$\text{iron}^{\text{bk}} \sqcap \text{steel}^{\text{bk}}$	Ch = (z->Iron_bk;END|z->Steel_bk;END)\{z}.
$*\text{look}^{\text{bk}}$	L = (z->SKIP;END|z->Look_bk;SKIP;L)\{z}.
$(\text{iron}^{\text{bk}} \sqcap \text{steel}^{\text{bk}}); \text{look}^{\text{bk}}$	S = Ch;SKIP;Look_bk;END.
$\text{save}^{\text{mk}} \Vert \text{check}^{\text{mk}}$	||TP = (p1 : Check_mk||p2 : Save_mk).
	P = TP;SKIP;END/{check_mk/p1.check_mk, save_mk/p2.save_mk}.

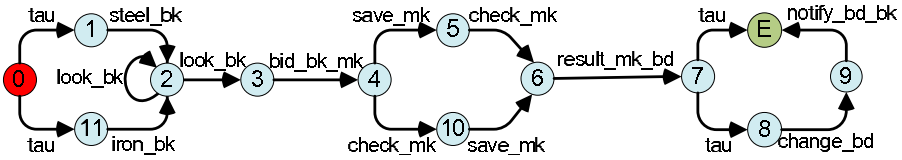

Figure 2: Minimized LTS of the *Stock* Market Case Study

$$\text{p2f}(C,p) \widehat{=} \text{p2f}_{pi}(C,p) = \text{c2f}_{pi}(C)@\{b | b \text{ is an activity of } p\}.$$

As specified in [18] for projecting Chor to peers, the name of each Peer process is taken as a part of each activity name (for instance, here we add it as suffix). Therefore, local activities of different peers are pair-wise different, and peers use exclusive channels for communicating with each other. Thus, each channel synchronizes activities of exactly two peers. Hence, in $\text{p2f}_{pi}(C,1) \Vert \cdots \Vert \text{p2f}_{pi}(C,n)$, only actions which represent communication activities are synchronized with each other, and each of these actions belongs to alphabets of exactly two FSP processes of the parallel operator's operands. We also proved that this translation preserves the semantics of the Peer language [19].

Example. For each Peer P, all actions in $\text{c2f}_{pi}(\text{Stock})$ in which P is not involved, are hidden. The three peers of our example are encoded by the following FSP specifications:

$\text{Broker} = \text{c2f}_{\text{pi}}(\text{Stock}); \text{END}@\{\text{iron\_bk}, \text{steel\_bk}, \text{bid\_bk\_mk}, \text{look\_bk}, \text{notify\_bd\_bk}\}.$

$\text{Market} = \text{c2f}_{\text{pi}}(\text{Stock}); \text{END}@\{\text{save\_mk}, \text{check\_mk}, \text{bid\_bk\_mk}, \text{result\_mk\_bd}\}.$

$\text{Board} = \text{c2f}_{\text{pi}}(\text{Stock}); \text{END}@\{\text{result\_mk\_bd}, \text{notify\_bd\_bk}, \text{change\_bd}\}.$

Figure 3 shows the minimized LTSs of these peers generated from the FSP processes presented above.

3.3 Realizability

Definition 3 formalizes the notion of choreography realizability we use in this article. We chose a strong realizability [2, 7] for experimentation purposes, but weak notions could be used instead [7].

Definition 3 (Realizability of Chor)**.** *For a Chor specification C with n peers, we say C is realizable under natural projection, if and only if the following two conditions hold:*

1. $[\![C]\!] = [\![\text{nproj}(C,1) \Vert \cdots \Vert \text{nproj}(C,n)]\!]$
2. $\nexists t \,.\, \text{nproj}(C,1) \Vert \cdots \Vert \text{nproj}(C,n) \stackrel{t}{\Longrightarrow} \dagger$

Both Chor and Peer languages use trace semantics. Therefore, for checking the realizability of a Chor specification we need to compare the trace set of a Chor specification with the trace set of the parallel composition of all peers. We proved in [19] that the trace set of a Chor specification is equal to the trace set of its FSP encoding, we also proved our encoding preserves the semantics of the Peer language. Thus, we have to check that FSP specifications for Chor and peers produce the same set of traces (in which τ actions are hidden) and terminate. Although the Chor specification is deadlock-free, the specification of the final system made of interacting peers (generated using natural projection) may cause deadlock. In addition to check that both specifications have the same set of traces, the parallel composition of the different peers has also to be deadlock-free. This check is easily computed using the LTSA toolbox. Also, one can perform any kind of test that is provided by LTSA, such as checking temporal properties between different activities in the Chor and Peer specifications.

Example. As for the realizability test, we first compute LTSs from FSP processes Stock and Peers, using LTSA. The FSP process for the whole system is: $\Vert \text{Peers} = (\text{Broker} \Vert \text{Market} \Vert \text{Board})$. Then, we compare trace sets of these processes using *ltscompare*, one of the tools belonging to the mCRL2 toolset[3] [6], and find out they produce

[3]LTSA does not allow to compute trace equivalence of two LTSs.

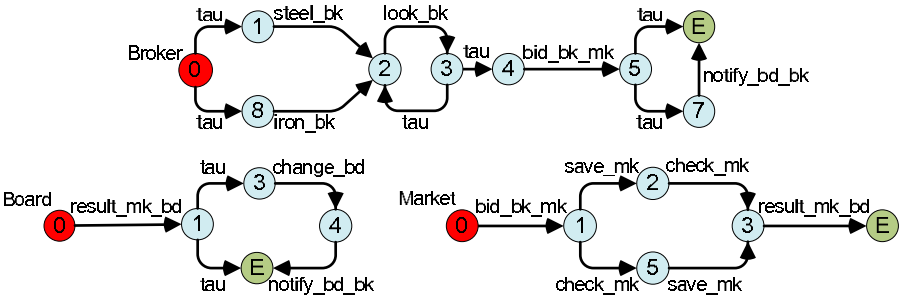

Figure 3: *Stock* Market: Minimized LTSs of Peers

the same set of traces (first realizability condition, Definition 3). For a Chor specification to be realizable, it is also required to satisfy the second condition of Definition 3. LTSA helps us on validating this condition, and using the *check safety* test, we find that the following trace causes deadlock:

$$\langle \mathsf{iron\_bk}, \mathsf{look\_bk}, \mathsf{bid\_bk\_mk}, \mathsf{check\_mk}, \mathsf{save\_mk}, \mathsf{result\_mk\_bd} \rangle$$

Indeed, after Broker sends his/her bid to the market, (s)he should decides if (s)he will be notified by the board or not. On the other hand, Board also makes this decision according to the result which is received from the market. So if peers Broker and Board make different decisions, a deadlock occurs. To make our specification realizable we slightly change it as follows: Whatever value is received from the market, Board always notifies the broker about the result. Thus, the specification of the system becomes as follows:

$\mathsf{Stock} =$
$(\mathsf{iron}^{\mathsf{bk}} \sqcap \mathsf{steel}^{\mathsf{bk}}); \mathsf{look}^{\mathsf{bk}}; *\mathsf{look}^{\mathsf{bk}}; \mathsf{bid}^{[\mathsf{bk},\mathsf{mk}]};$
$(\mathsf{save}^{\mathsf{mk}} || \mathsf{check}^{\mathsf{mk}}); \mathsf{result}^{[\mathsf{mk},\mathsf{bd}]};$
$(\mathsf{change}^{\mathsf{bd}} \sqcap \mathsf{skip}); \mathsf{notify}^{[\mathsf{bd},\mathsf{bk}]}$

This new specification satisfies both realizability conditions.

4 Dynamic reconfiguration of Chor specifications

4.1 Reconfigurability Definition

In this section, we show how we check whether a reconfiguration can be applied or not. Note that here our goal is not to verify the reconfiguration specification, it can be checked beforehand on the choreography specification using validation and verification techniques (see Section 3). Instead, we propose some techniques to check if, from a protocol point of view, a reconfiguration preserves the global flow of control executed so far.

This process accepts as input two choreographies (an initial one, say C_I, and a reconfigured one, say C_R) and a trace t which corresponds to the history of the current execution (sequence of local or communication activities, that interacting peers have performed). Traces only contain observable activities (τ corresponding to internal actions and used to encode non-deterministic choices in peers are not stored in these traces). From the choreography specification C_R, peer LTSs are obtained using techniques presented in Section 3. If the trace t executed by peers obtained out of C_I can also be executed in reconfigured peers generated from C_R, then the reconfiguration can take place.

Definition 4 (Reconfigurability)**.** *Given two choreographies C_I and C_R, two sets of peers P_I and P_R respectively obtained from those choreographies, and a trace t, the current system consisting of peers P_I is reconfigurable to peers P_R if there exists P'_R such that $P_R \stackrel{t}{\Longrightarrow} P'_R$, where $\stackrel{t}{\Longrightarrow}$ stands for the execution of local or communication activities as specified in trace t.*

In practice, a reconfiguration is applied as follows: First, actual peers matching with abstract descriptions (LTSs) derived from the choreography C_R are seeked into databases of peers (*e.g.,* UDDI) or directly reused from the former system for peers which have not been modified. Next, these peers are instantiated and executed (using the history stored in trace t) up to the point where the reconfiguration has been applied (this last part can be enforced by an external controller or a monitoring engine for instance). To sum up, our reconfigurability check aims to be transparent from an external point of view.

Example. Now imagine that after peer Broker selects iron ($\mathsf{t} = \langle \mathsf{iron}^{\mathsf{bk}} \rangle$), we want to reconfigure the current choreography for Stock market, in a way that i) in addition to iron and steel, Broker can select gold, and ii) Broker can send his/her bid to the Market without looking at the market. The new specification of the system is as follows:

$\mathsf{Stock} =$
$(\mathsf{iron}^{\mathsf{bk}} \sqcap \mathsf{steel}^{\mathsf{bk}} \sqcap \mathsf{gold}^{\mathsf{bk}}); *\mathsf{look}^{\mathsf{bk}}; \mathsf{bid}^{[\mathsf{bk},\mathsf{mk}]};$
$(\mathsf{save}^{\mathsf{mk}} || \mathsf{check}^{\mathsf{mk}}); \mathsf{result}^{[\mathsf{mk},\mathsf{bd}]};$
$(\mathsf{change}^{\mathsf{bd}} \sqcap \mathsf{skip}); \mathsf{notify}^{[\mathsf{bd},\mathsf{bk}]}$

Figure 4 shows the minimized LTS of the new peer (Broker). We first compute the parallel composition of peers P_R using LTSA, then we check in this system if it is possible to perform activities which are specified in t. The

Therefore, we first save them in a format *ltscompare* accepts, and then use it to check if LTSs have the same set of traces or not.

answer is yes, and P_I is reconfigurable to P_R. This is automatically checked using a prototype tool we implemented (see Section 5).

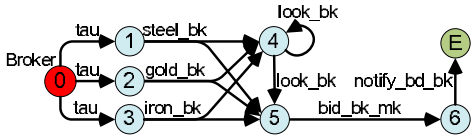

Figure 4: Minimized LTS of new *Broker*

4.2 Reconfigurability Analysis

Our reconfigurability definition, only says if the activities that have occurred so far can be reproduced in the new system, then peers P_I are reconfigurable to peers P_R. We want to go further than this check since in some situations, one may want these reconfigurations to have an immediate impact on the running system, or to preserve the forthcoming behaviour as specified in the former choreography (the system can do at least what was possible before, but it can do more as well).

Therefore, Definition 4 is completed with a couple of analysis of where the modifications take place, that is we check if modifications appear in peers after the current global state, and if the evolutions possible from the current global state are preserved with respect to the former choreography. These analyses may help the designer to decide whether (s)he wants to reconfigure the system or not. Indeed, we can imagine situations in which for instance the designer may want these modifications to immediately impact the whole behaviour.

The first case, referred as *preservative* in the following, is computed by first performing reconfigurability check and finding P'_R. Then if P'_R is found, it is checked that all traces which can be executed from P'_I (assuming $P_I \stackrel{t}{\Longrightarrow} P'_I$) can also be executed from P'_R.

Definition 5 (Preservative Reconfiguration)**.** *Given two choreographies C_I and C_R, two sets of peers P_I and P_R respectively obtained from previous choreographies, and a trace t, new peers P_R are preservative with respect to former peers P_I, if P_I is reconfigurable to P_R and $[\![P'_I]\!] \subseteq [\![P'_R]\!]$, assuming $P_I \stackrel{t}{\Longrightarrow} P'_I$ and $P_R \stackrel{t}{\Longrightarrow} P'_R$.*

Note that P'_I and P'_R obtained by application of trace t are unique, because τ transitions have been removed from peers and they have been determinized (no two transitions holding the same label going out from the same state) after performing the realizability check presented in Section 3.2.

The second case, referred as *modificative* in the following, is computed by first extracting the current global state from the trace t, and checking for each reconfigured peer if all new interactions are reachable from its current execution state.

Definition 6 (Modificative Reconfiguration)**.** *Given two choreographies C_I and C_R, two sets of peers P_I and P_R respectively obtained from previous choreographies, and a trace t, new peers P_R are modificative with respect to former peers P_I if in addition to be reconfigurable, for each peer $p_i \in P'_R$, $s_i \in (s_1,\ldots,s_n)$, we have* $\mathsf{reachable}(s_i,p_i) \cap M_i = M_i$, *where $P_R \stackrel{t}{\Longrightarrow} P'_R$, $(s_1,\ldots,s_n)$ is the current global state of peers P'_R, and M_i stands for all the modifications (added or removed interactions) applied between C_I and C_R for peer i.*

Given a peer i, modifications for this peer between choreographies C_I and C_R are obtained by computing the difference of both alphabets $A_{Ii} \backslash A_{Ri}$ ($A_{Ri} \backslash A_{Ii}$, resp.) if some interactions are removed (added, resp.). Function reachable from a state s and a peer LTS p is defined as follows:

$\forall s,p \,.\, \mathsf{reachable}(s,p) = \emptyset \Leftrightarrow \nexists s',l \,.\, (s,l,s') \in T$

$\forall s,p,l \,.\, l \in \mathsf{reachable}(s,p) \Leftrightarrow \exists s' \,.\, (s,l,s') \in T \vee$

$\quad (\exists l' \,.\, (s,l',s') \in T \wedge l \in \mathsf{reachable}(s',p))$

where T is the transition relation belonging to the peer LTS $p = (A,S,I,F,T)$.

Last, realizability of choreography C_R can be checked using techniques presented in Section 3, and this realizability result is another analysis on which the user can rely on to decide whether or not applying the reconfiguration.

Example. Suppose that in addition to be reconfigurable, we want our system to verify both properties. Since peers P_R are reconfigurable with respect to former peers P_I, we know P'_R exists such that $P_R \stackrel{t}{\Longrightarrow} P'_R$. Therefore, assuming $P_I \stackrel{t}{\Longrightarrow} P'_I$, for reconfiguration to be preservative we need $[\![P'_I]\!] \subseteq [\![P'_R]\!]$. This check can be performed using the *ltscompare* tool, and by performing that check we find that our reconfiguration example is preservative.

As regards the modificative reconfiguration property, $\mathsf{M_{bk}} = \{\mathsf{gold_{bk}}\}$, $\mathsf{M_{mk}} = \mathsf{M_{bd}} = \emptyset$. After selecting iron, there is no way to perform $gold_{bk}$. Consequently, our reconfiguration is not modificative. We have to wait for the current execution to get finished first, and then reconfigure the peers if we want this property to be satisfied.

5 Prototype tool

All the steps of the approach we have presented in Sections 3 and 4 are automatically computed by a prototype tool we implemented (see an overview in Figure 5). Boxes and diamonds with dashed borders are optional. We explicitly wrote names of tools that we did not implement at the bottom of each box or diamond. In Section 3 we have mentioned that one reason to choose an intermediate language, is that we can reuse tools which have already been created for that language. If we assume each box or diamond as a unit of work, one can see that using our approach, we only implemented 41.4 percent of our prototype tool, and 58.6 remaining percent are already implemented in existing tools.

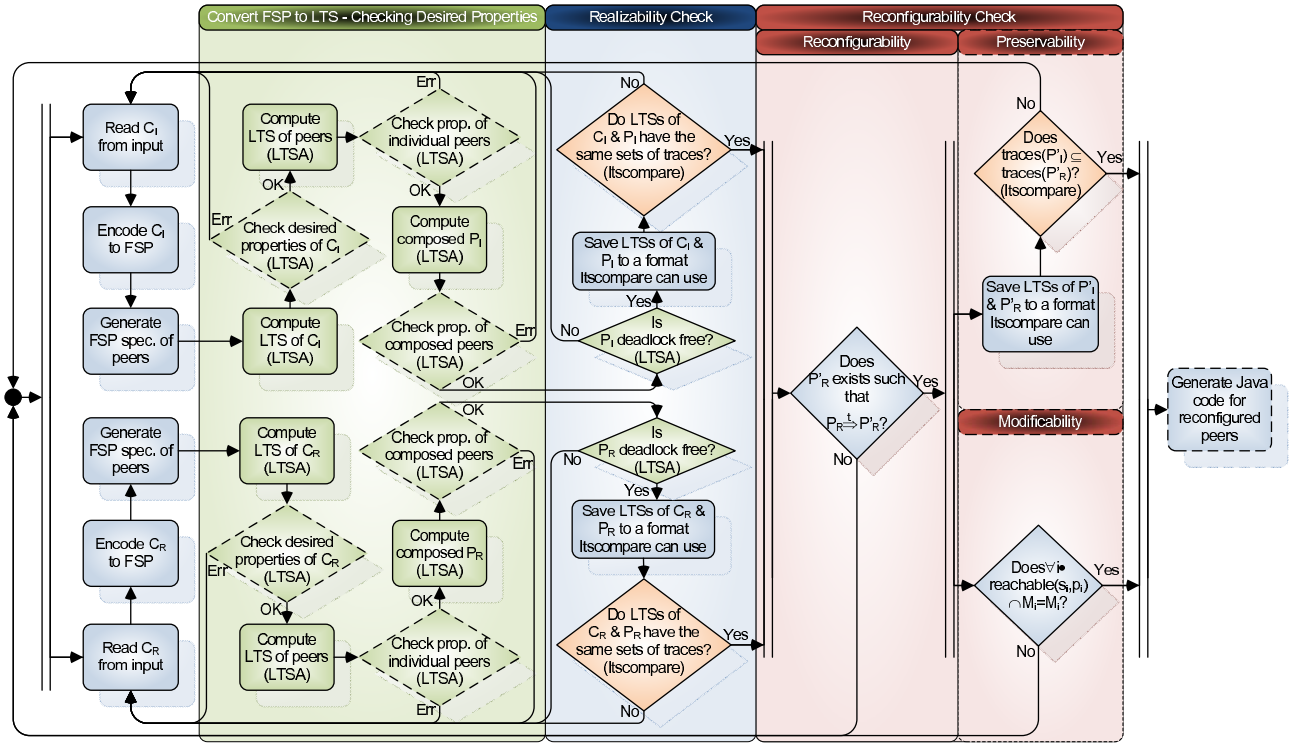

Figure 5: Overview of our prototype tool

5.1 Experimental Results

Table 5 shows experimental results on some of the examples of our database. Each row of this table shows results for one reconfiguration request (C_I, C_R, and t), and respectively presents the number of: peers, distinct basic activities used in the Chor specification, basic activities used in the Chor specification, basic activities in t (length of t), FSP processes resulting while encoding the Chor specification into FSP, and states and transitions in the *minimized* LTS corresponding to the parallel composition of peers (P_I and P_R). It also presents result of realizability plus different types of reconfigurability checks and amount of consumed time and memory. For keeping the table as simple as possible, we chose examples in which number of peers in C_I and C_R are equal. Also number of (distinct) basic activities, and FSP processes for C_I and C_R are close to each other (the maximum is shown).

Note that measured time and memory for the reconfigurability check include time and memory required for the realizability check. Also, time and memory for the two other types of reconfigurability check include time and memory required for the basic reconfigurability check in addition to the realizability check.

Whenever a reconfigurability check fails, there is no need to check the preservability or modificability properties, since being reconfigurable is a precondition for being preservative or modificative.

5.2 Code Generation

As mentioned earlier, the final step is to produce Java code following guidelines presented in [12]. Like the other steps, this is completely automated by our tool. Figure 6 shows a simplified version of some classes produced for our running example. We define an **interface** Channel and implement it in a **class** ChannelImpl. For each channel in the specification, one instance of ChannelImpl is created in **class** ChannelServer and registered in a server. Also, for each peer we create one interface and one class. The interface contains methods for local and communication activities performed by the peer and must be implemented by the user, because the semantics of basic activities used in the specification is not defined. Code in the class file implements the peer protocol and should not be changed. The user only needs to implement interfaces of peers and distributes classes to different locations, as (s)he needs.

Let us comment in more details, for illustration purposes, method run in **class** mkController. We can notice that for each operand of the parallel operator we created one separate thread, and used **class** CyclicBarrier (the Java utility class) to guarantee that the execution of both threads must be finished before the next activities are performed (cb1.wait() and cb2.wait()). Also, SynchronousQueue used in **class** ChannelImpl is another Java class which synchronizes its read/write operations, therefore our communication mechanism remains synchronous.

Table 5: Experimental Results

P	dBA	BA	t	FSP	States		Trans.		Realizability			Reconf.			Preservability			Modificability		
3	5	5	0	10	3	5	2	6	√	96ms	796K	√	174ms	754K	√	187ms	771K	√	173ms	870K
3	8	9	1	18	12	19	16	28	√	140ms	815K	√	224ms	824K	√	248ms	886K	√	243ms	949K
5	16	17	4	29	31	37	52	66	√	174ms	354K	√	392ms	900K	√	404ms	916K	√	378ms	1024K
5	16	36	6	30	65	194	108	449	×	146ms	978K	×	146ms	978K	×	1s	978K	√	146ms	978K
4	5	8	8	15	67	47	489	255	√	833ms	1187K	√	1.1s	2052K	×	1.1s	2055K	√	1.1s	2056K
5	14	672	9	29	757	755	1428	1416	√	3.2s	2965K	√	12.1s	3116K	×	80s	3130K	×	12s	3014K
6	6	6	0	16	95	141	340	602	√	4.7s	2501K	√	5.2s	5212K	√	5.5s	5516K	√	5.2s	5559K
7	13	13	8	23	250	374	725	1277	√	1.3s	4170K	√	1.7s	6375K	√	1.7s	6942K	√	1.7s	6454K
7	17	834	11	35	932	934	1372	1372	√	8.5s	2739K	√	42s	2732K	√	244s	2947K	×	42s	2982K

```
public class mkController extends Thread {
  private final mk mk;
  private final Channel bk;
  private final Channel bd;
  public mkController(mk mk, String server) throws
    RemoteException,NamingException{
    this.mk = mk;
    final Context namingContext = new InitialContext();
    bk = (Channel) namingContext
      .lookup("rmi://"+server+"/bk_mk");
    bd = (Channel) namingContext
      .lookup("rmi://"+server+"/mk_bd");}
  public void run() {
    final Serializable msg1 = bk.recv();
    mk.recv_from_bk(msg1);
    final CyclicBarrier cb1 = new CyclicBarrier(2);
    new Thread(new Runnable() {
      public void run() {
        mk.check();
        cb1.wait();}}).start();
    new Thread(new Runnable() {
      public void run() {
        mk.save();
        cb1.wait();}}).start();
    final Serializable msg2 = mk.send_bd_value();
    bd.send(msg2);}}
```

```
public interface mk {
  void save();
  void check();
  void recv_from_bk(Serializable value);
  Serializable send_bd_value();}
public class ChannelImpl implements Channel {
  public ChannelImpl() throws RemoteException {
    UnicastRemoteObject.exportObject(this, 0);}
  private final SynchronousQueue syncQueue = new
    SynchronousQueue();
  public void send(Serializable value) throws
    RemoteException,InterruptedException{
    syncQueue.put(value);}
  public Serializable recv() throws
    RemoteException,InterruptedException{
    return syncQueue.take();}}
public class ChannelServer {
  public ChannelServer() throws
    RemoteException,NamingException{
    final Channel bk_mk = new ChannelImpl();
    final Channel mk_bd = new ChannelImpl();
    final Channel bd_bk = new ChannelImpl();
    final Context namingContext = new InitialContext();
    namingContext.bind("rmi:bk_mk", bk_mk);
    namingContext.bind("rmi:mk_bd", mk_bd);
    namingContext.bind("rmi:bd_bk", bd_bk);}}
```

Figure 6: *Stock* Market: Java Code

6 Related works

Several works aimed at studying and defining the conformance and/or realizability problem for choreography. In [3], the authors define models for choreography and orchestration, and formalise a conformance relation between both models. These models are assumed given as input whereas we focus on the generation of one from the other (generation of peers from a global specification). In [22], the authors focus on *Let's dance* models for choreographies, and define for them an algorithm that determines if a global model is locally enforceable, and another algorithm for generating local models from global ones. In [15], 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.

Some works define several realizability notions, and classify them in a hierarchy [7]. Bultan and Fu [2] tackle the realizability issue in the context of asynchronous communication, and recently defined some sufficient conditions to test realizability of choreographies specified with collaboration diagrams. In [18, 11], formal languages to describe choreographies were proposed. Conformance with respect to an orchestration specification and implementability issues were studied from a formal point of view.

Other works [4, 18] propose well-formedness rules to enforce the specification to be realizable. For example, in [4], the authors rely on a π-calculus-like language 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 distributed processes from the choreography.

Dynamic reconfiguration [14] is not a new topic and many solutions have already been proposed in the context of distributed systems and software architectures [9, 10], graph transformation [1, 21], software adaptation [17, 16], or metamodelling [8, 13]. However, to the best of our knowledge, nobody has already worked on the reconfiguration of service interactions initially described using a

choreography specification.

As regards tools automating the realizability test, WSAT [5] takes conversation protocols as input, and checks a set of realizability conditions on them. Another tool-supported approach [20] computes realizability using a LOTOS encoding. However, in [20] the choreography language, namely collaboration diagrams, is less expressive than Chor (no choice and a loop operator restricted to a single message), and the proposal focuses only on abstract languages (no relationships with implementations or real code).

7 Concluding remarks

In this article, we have presented an encoding of the choreography calculus Chor into the process algebra FSP. This encoding allows to generate a set of peers corresponding to the choreography, and in a second step to check that (i) they realize the original choreography, and (ii) they ensure some expected properties (by animation and model-checking with LTSA). If the choreography is not realizable or erroneous, the Chor specification can be corrected and the process started again. If a choreography is as expected by the designer, Java code can be generated for rapid prototyping purposes. We have also proposed some techniques to verify if some reconfigurations can be applied dynamically on some peers that have been generated from a choreography specification. For illustration purposes, we have used the Chor language and transition systems to describe peers. Reconfigurations have been specified as a new version of the choreography where some interactions have been added or removed. Our approach is completely automated by a prototype tool we implemented and applied to a large number of examples.

Our main perspective plans to extend our approach to consider asynchronous communication. In this article, we have focused on synchronous communication, and it makes the realizability and reconfigurability checking easier. Dealing with asynchronous communication is a realistic assumption with respect to implementation platforms, however it complicates the analysis and verification stage. Asynchronous communication can be specified using queues. In this context, realizability and reconfigurability results depend on queue size, and some theoretical issues are still open problems such as the relationships of realizability results for queues of size one, queues of size k, and infinite queues. We also plan to extend our analysis techniques to take other kinds of reconfigurations into account. As an example, in some situations one may wish to reduce the behaviour of the interacting peers while producing only traces that were executable before reconfiguring the system (this is the opposite of the preservative property presented in Section 4).

References

[1] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer, 2003.

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

[3] 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, 2006.

[4] 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, 2007.

[5] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 510–514. Springer, 2004.

[6] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In *Proc. of MMOSS'07, Dagstuhl seminar*, 2007.

[7] R. Kazhamiakin and M. Pistore. Analysis of Realizability Conditions for Web Service Choreographies. In *Proc. of FORTE'06*, volume 4229 of *LNCS*, pages 61–76. Springer, 2006.

[8] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer, 2004.

[9] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[10] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[11] 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.

[12] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs, 2nd edition*. Wiley, 2006.

[13] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.

[14] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.

[15] 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, 2005.

[16] P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.

[17] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Software Adaptations. In *Proc. of WCAT'06*, 2006.

[18] 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.

[19] N. Roohi, G. Salaün, and S. H. Mirian. Analyzing Chor Specifications by Translation into FSP. In *Proc. of FOCLASA'09*, volume 255 of *ENTCS*, pages 159–176, 2009.

[20] G. Salaün and T. Bultan. Realizability of Choreographies using Process Algebra Encodings. In *Proc. of IFM'2009*, volume 5423 of *LNCS*, pages 167–182. Springer, 2009.

[21] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE 2001*, pages 21–32. ACM, 2001.

[22] J. Maria Zaha, M. Dumas, A. H. M. ter Hofstede, A. P. Barros, and G. Decker. Service Interaction Modeling: Bridging Global and Local Views. In *Proc. of EDOC'06*, pages 45–55. IEEE Computer Society, 2006.