Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 255 (2009) 159–176
www.elsevier.com/locate/entcs

Analyzing Chor Specifications by Translation into FSP

Nima Roohi[1]

Department of Computer Science
Sharif University of Technology
Tehran, Iran

Gwen Salaün[2]

Department of Computer Science
University of Málaga
Málaga, Spain

Seyyed Hassan Mirian[3]

Department of Computer Science
Sharif University of Technology
Tehran, Iran

Abstract

A *choreography* specifies activities and interactions among a set of services from a global point of view. 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*. This paper presents an encoding of the Chor choreography calculus into the FSP process algebra. This encoding allows to: (i) validate and verify Chor specifications using the FSP toolbox (LTSA), (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. Our proposal is supported and completely automated by a prototype tool we have implemented.

Keywords: Choreography, WS-CDL, Chor, FSP

1 Introduction

Choreography description languages aim at specifying from a global point of view interactions among a set of services involved in a new application. Several formalisms

[1] Email: roohi@ce.sharif.edu
[2] Email: salaun@lcc.uma.es
[3] Email: hmirian@sina.sharif.edu

1571-0661© 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.10.030

have already been proposed to specify choreographies: WS-CDL, collaboration diagrams, BPMN, SRML, etc. From such specifications, local implementations, namely *peers*, can be automatically generated. However, generation of peers which exactly implement the choreography specification is not always possible: this problem is known as *realizability*.

Recent works on this topic [7,12,4,2] advocate techniques to check realizability of a choreography, or define well-formedness rules to be applied while writing the choreography specification in order to ensure its realizability. However, most of these approaches focus on theoretical aspects and lack of tool support. In addition, these works mainly focus on the realizability issue but do not allow to check choreographies (in order to verify that the overall goal of the composition is achieved) or generate code from such specifications.

In this paper, we use the Chor calculus [12] as choreography language because it is an abstract model of WS-CDL coming with a formal syntax and semantics (not the case of WS-CDL). We propose an encoding from Chor into the FSP process algebra [9]. 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 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. Our proposal is supported and completely automated by a prototype tool we have implemented.

The rest of this paper is organized as follows: Section 2 introduces the Chor and FSP calculi. Section 3 presents the encoding of Chor into FSP. Section 4 extends the FSP encoding to take the peer generation into account, and focuses on the realizability issue. Section 5 sketches the prototype tool that supports our approach and the Java code generation. Section 6 compares our proposal to related works, and Section 7 ends the paper with some concluding remarks.

2 Preliminaries: Chor and FSP

2.1 Chor

Chor [12] 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 [12]. First of all, let us define the trace operators which are used throughout this paper.

Definition 2.1 Trace operators which are used in this paper are defined as follows (t, t_1, and t_2 are arbitrary traces, T, T_1, and T_2 are arbitrary sets of traces, S is a set of actions, R is a partial function [4] of type $Action \rightarrow Action$, and $\mathsf{dom}R$ stands

[4] In general, R can be a partial relation, but we only use it when it is a partial one-to-one function.

for the domain of R):

$\#t$: length of t
t^0 : head of t (first action of t)
t' : tail of t (trace obtained from t by removing its head)
$t[i]$: i-th element of t ($0 \leq i < \#t$)
$\mathsf{s}(t)$: set of actions in t ($\{t[i] | \forall 0 \leq i < \#t\}$)

t_1 concatenated with t_2 (trace with length $\#t_1 + \#t_2$):

$$(t_1 \frown t_2)[i] \mathrel{\widehat{=}} \begin{cases} t_1[i] & \text{if } i < \#t_1 \\ t_2[i - \#t_1] & \text{else} \end{cases}$$

t renamed by R:

$$t/R \quad \mathrel{\widehat{=}} \begin{cases} t & \text{if } t = \langle\rangle \\ \langle t^0 \rangle \frown t'/R & \text{if } t \neq \langle\rangle \wedge t^0 \\ & \quad \notin \mathsf{dom} R \\ \langle R(t^0) \rangle \frown t'/R & \text{else} \end{cases}$$

t_1 interleaved with t_2:

$$t_1 \bowtie t_2 \mathrel{\widehat{=}} \begin{cases} \{t_1\} & \text{if } t_2 = \langle\rangle \\ \{t_2\} & \text{if } t_1 = \langle\rangle \\ t_1^0 \frown (t_1' \bowtie t_2) \cup & \text{else} \\ t_2^0 \frown (t_1 \bowtie t_2') \end{cases}$$

t_1 synchronized with t_2 on S:

$$t_1 \overset{S}{\bowtie} t_2 \mathrel{\widehat{=}} \begin{cases} \{t_1\} & \text{if } t_2 = \langle\rangle \wedge \\ & \mathsf{s}(t_1) \cap S = \emptyset \\ \{t_2\} & \text{if } t_1 = \langle\rangle \wedge \\ & \mathsf{s}(t_2) \cap S = \emptyset \\ t_1^0 \frown (t_1' \overset{S}{\bowtie} t_2') & \text{if } t_1^0 = t_2^0 \\ & \quad \wedge t_1^0 \in S \\ t_1^0 \frown (t_1' \overset{S}{\bowtie} t_2) \cup & \text{if } t_1^0 = t_2^0 \wedge \\ t_2^0 \frown (t_1 \overset{S}{\bowtie} t_2') & t_1^0 \notin S \vee \\ & t_1^0, t_2^0 \notin S \\ \{\} & \text{else} \end{cases}$$

t restricted to S:

$$t \downharpoonright S \quad \mathrel{\widehat{=}} \begin{cases} t & \text{if } t = \langle\rangle \\ t' \downharpoonright S & \text{if } t \neq \langle\rangle \wedge t^0 \notin S \\ \langle t^0 \rangle \frown t' \downharpoonright S & \text{else} \end{cases}$$

t filtered by S:

$$t \backslash S \mathrel{\widehat{=}} \begin{cases} t & \text{if } t = \langle\rangle \\ t' \backslash S & \text{if } t \neq \langle\rangle \wedge t^0 \in S \\ \langle t^0 \rangle \frown t' \backslash S & \text{else} \end{cases}$$

t concatenated i times with itself:

$$t^{(i)} \quad \mathrel{\widehat{=}} \begin{cases} \langle\rangle & \text{if } i = 0 \\ t \frown t^{(i-1)} & \text{else} \end{cases}$$

T concatenated i times with itself:

$$T^{(i)} \quad \mathrel{\widehat{=}} \begin{cases} \{\langle\rangle\} & \text{if } i = 0 \\ T \frown T^{(i-1)} & \text{else} \end{cases}$$

T renamed by R:
$T/R \quad \mathrel{\widehat{=}} \{t/R | t \in T\}$

T restricted to S:
$T \downharpoonright S \mathrel{\widehat{=}} \{t \downharpoonright S | t \in T\}$

T filtered by S:
$T \backslash S \mathrel{\widehat{=}} \{t \backslash S | t \in T\}$

t_1 concatenated with T_2:
$t_1 \frown T_2 \mathrel{\widehat{=}} \{t_1\} \frown T_2$

Closure of t:
$t^* \quad \mathrel{\widehat{=}} \bigcup_{i \geq 0} t^{(i)}$

Closure of T:
$T^* \quad \mathrel{\widehat{=}} \bigcup_{i \geq 0} T^{(i)}$

T_1 concatenated with T_2:
$T_1 \frown T_2 \mathrel{\widehat{=}} \{t | \exists t_1 \in T_1 \wedge \exists t_2 \in T_2 \centerdot t = t_1 \frown t_2\}$

T_1 interleaved with T_2:
$T_1 \bowtie T_2 \mathrel{\widehat{=}} \{t | \exists t_1 \in T_1 \wedge \exists t_2 \in T_2 \centerdot t \in t_1 \bowtie t_2\}$

T_1 synchronized with T_2 on S:
$T_1 \overset{S}{\bowtie} T_2 \mathrel{\widehat{=}} \{t | \exists t_1 \in T_1 \wedge \exists t_2 \in T_2 \centerdot t \in t_1 \overset{S}{\bowtie} t_2\}$

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

Therefore R^{-1} is also a partial function, and can be used in the defined operators.

semantics (where $[\![C]\!]$ stands for the weak trace set of C). The reader interested in more details on the language may refer to [12].

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, $[\![*C]\!] = [\![C]\!]^*$

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 $P \stackrel{t}{\Longrightarrow} \epsilon$ we have $t \in [\![P]\!]$ (we use † to denote deadlock). Table 2 gives the syntax and semantics of the Peer language.

In Table 2, P, P_1 and P_2 are 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 input activity, 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):

$$\mathsf{fst}(\epsilon) = \mathsf{fst}(skip) = \mathsf{fst}(P_1 \sqcap P_2) = \mathsf{fst}(*P) \widehat{=} \emptyset \qquad \mathsf{fst}(\alpha) \widehat{=} \{\alpha\}$$
$$\mathsf{fst}(P_1; P_2) \widehat{=} \mathsf{fst}(P_1) \qquad \mathsf{fst}(P_1 \| P_2) \widehat{=} \mathsf{fst}(P_1) \cup \mathsf{fst}(P_2)$$

$$skip/\alpha \widehat{=} \perp \qquad \alpha/\alpha' \widehat{=} \begin{cases} \epsilon & \text{if } \alpha = \alpha' \\ \perp & \text{if } \alpha \neq \alpha' \end{cases}$$
$$(P_1; P_2)/\alpha \widehat{=} P_1/\alpha; P_2 \qquad (P_1 \sqcap P_2)/\alpha = (*P)/\alpha \widehat{=} \perp$$
$$(P_1 \| P_2)/\alpha \widehat{=} \begin{cases} P_1/\alpha \| P_2 & \text{if } \alpha \in \mathsf{fst}(P_1) \\ P_1 \| P_2/\alpha & \text{if } \alpha \in \mathsf{fst}(P_2) \\ \perp & \text{else} \end{cases}$$

Table 2
Syntax & Semantics of Peer

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

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

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

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

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

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

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

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

Example 2.2 We will use throughout this paper a metal stock market as a 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:

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

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 [9]. 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 paper (x and y are actions, and P and Q are FSP processes).

In the following lemma we show how trace sets of the FSP operators presented in Table 3 can be computed (this lemma will be used later on in this paper). The

Table 3
FSP Operators and Informal Semantics

$(x \to P)$	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.
$(x \to P \mid y \to Q)$	describes a process that either executes action x and then P, or action y and then Q.
$(P \mid\mid Q)$	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.
$P/\{new_1/old_1, \cdots, new_n/old_n\}$	renames action labels. Each old label in P is replaced by the new one.
$P\backslash\{x_1, \cdots, x_n\}$	removes action names $x_1, \cdots, x_n$ 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.
$P@\{x_1, \cdots, x_n\}$	hides all actions in the alphabet of P which do not belong to the set $\{x_1, \cdots, x_n\}$.

formal semantics of these operators are defined using LTS and can be found in [9].

Lemma 2.3 (Trace Set of FSP) *Traces for each FSP operator are obtained as follows, where $P, P_1, \cdots, P_n$ are FSP process identifiers, αP stands for the alphabet of P, $\alpha P_{i\cdots j}$ stands for $\alpha P_i \cup \cdots \cup \alpha P_j$, and END is the simplest FSP process with no transition and one start and final state:*

$$
\begin{aligned}
&[\![END]\!] &&= \{\langle\rangle\}\\
&[\![P_1;\cdots;P_n]\!] &&= [\![P_1]\!] \frown \cdots \frown [\![P_n]\!]\\
&[\![ba_1 \to P_1 \mid \cdots \mid ba_n \to P_n]\!] &&= (ba_1 \frown [\![P_1]\!]) \cup \cdots \cup (ba_n \frown [\![P_n]\!])\\
&[\![p1 : P_1 \| \cdots \| pn : P_n]\!] &&=\\
&\quad [\![P_1]\!]/\{(x, p1.x) | x \in \alpha P_1\} \bowtie \cdots \bowtie [\![P_n]\!]/\{(x, pn.x) | x \in \alpha P_n\}\\
&[\![P_1 \| \cdots \| P_n]\!] &&= [\![P_1]\!] \overset{\alpha P_1 \cap \alpha P_{2\cdots n}}{\bowtie} \left([\![P_2]\!] \overset{\alpha P_2 \cap \alpha P_{3\cdots n}}{\bowtie} \left(\cdots \overset{\alpha P_{n-1} \cap \alpha P_n}{\bowtie} [\![P_n]\!] \right) \right)\\
&[\![(P;END)/R]\!] &&= [\![P]\!]/R\\
&[\![(P;END)\backslash S]\!] &&= [\![P]\!]\backslash S\\
&[\![(P;END)@S]\!] &&= [\![P]\!] \downarrow S
\end{aligned}
$$

3 Translating Chor into FSP

In this section we introduce our encoding of Chor into FSP, which allows to use existing tool support for FSP. Thus, the resulting FSP specification can be compiled into LTS (Labeled Transition System), and checked using LTSA (animation, LTL model-checking). This encoding, will also be used to generate peers and check for realizability (see Section 4).

First of all, activities *skip* are hidden in Chor traces, so we use τ for encoding *skip* activities into FSP [5]. Each local or communication activity of Chor is encoded

[5] It is possible to first remove *skip* activities from a Chor specification and then perform the encoding. It is also possible to prove that for every Chor specification C there exists Chor specifications C' and C'' in

using an FSP process with a single action corresponding to this activity.

The Chor sequential operator is encoded using the FSP sequential operator. As regards the choice operator, we prefix each operand by a τ action, therefore similarly to Chor and Peer languages, selecting a choice operand is performed non-deterministically. As an example, if the first action in one operand of a choice operator is $c^{[i,j]}$, it is also possible $c^{[i,j]}$ be in the alphabet of another FSP process. We will see in Section 4 that these FSP processes may be composed using the FSP parallel operator (without any renaming). Selection of a choice operand has to be performed in an independent way. Hence, deadlock can occur if one FSP process chooses to perform the operand with $c^{[i,j]}$ as action and the other does not.

As far as the parallel operator is concerned, the translation is more complicated. In FSP, actions which are in alphabets of both operands can only evolve through synchronizations, but the parallel operator of Chor does not synchronize activities of its operands (interleaving). Consequently, we first prefix operands of each parallel operator with a unique value (*e.g.*, $p1{:}P_1 \| p2{:}P_2$ instead of $P_1 \| P_2$), thus no synchronization occurs. Then, we use the renaming operator of FSP to replace these new action names with their original values. To perform this final renaming while encoding the parallel operator, we need to figure out all the non-skip basic activities for each operand. These activities are computed using Definition 3.1.

Definition 3.1 [Included Basic Activities] We define ac, a function of type $Chor \rightarrow \mathbb{P}(Activity)$, as follows:

$$\mathsf{ac}(skip) \mathrel{\widehat{=}} \{\} \quad \mathsf{ac}(a^i) \mathrel{\widehat{=}} \{a_i\} \quad \mathsf{ac}(c^{[i,j]}) \mathrel{\widehat{=}} \{c^{[i,j]}\}$$
$$\mathsf{ac}(C_1; \cdots ; C_n) = \mathsf{ac}(C_1 \sqcap \cdots \sqcap C_n) =$$
$$\mathsf{ac}(C_1 \| \cdots \| C_n) \mathrel{\widehat{=}} \mathsf{ac}(C_1) \cup \cdots \cup \mathsf{ac}(C_n)$$

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 3.2 [Chor into FSP] Encoding a Chor specification C into FSP is achieved using function $\mathsf{c2f} : Chor \rightarrow FSPdescription$, as follows:

$\mathsf{c2f}(skip) \mathrel{\widehat{=}} SKIP = (skip \text{->} END) \backslash \{skip\}.$

$\mathsf{c2f}(a^i) \mathrel{\widehat{=}} \mathsf{c2f}_{pi}(a^i) = (a\_i \text{->} END).$

$\mathsf{c2f}(c^{[i,j]}) \mathrel{\widehat{=}} \mathsf{c2f}_{pi}(c^{[i,j]}) = (c\_i\_j \text{->} END).$

$\mathsf{c2f}(C_1; C_2) \mathrel{\widehat{=}} \mathsf{c2f}_{pi}(C_1; C_2) = \mathsf{c2f}_{pi}(C_1); SKIP; \mathsf{c2f}_{pi}(C_2); END.$

$\mathsf{c2f}(C_1 \sqcap C_2) \mathrel{\widehat{=}} \mathsf{c2f}_{pi}(C_1 \sqcap C_2) = (z \text{->} \mathsf{c2f}_{pi}(C_1); END | z \text{->} \mathsf{c2f}_{pi}(C_2); END) \backslash \{z\}.$
assuming z is neither in $\alpha\mathsf{c2f}_{pi}(C_1)$ nor in $\alpha\mathsf{c2f}_{pi}(C_2)$.

$\mathsf{c2f}(C_1 \| C_2) \mathrel{\widehat{=}}$
$\| T.\mathsf{c2f}_{pi}(C_1 \| C_2) = (p1 : \mathsf{c2f}_{pi}(C_1) \| p2 : \mathsf{c2f}_{pi}(C_2)).$
$\mathsf{c2f}_{pi}(C_1 \| C_2) = T.\mathsf{c2f}_{pi}(C_1 \| C_2); SKIP; END /$
$\{ba_1/p1.ba_1 | ba_1 \in \mathsf{ac}(C_1)\} \cup \{ba_2/p2.ba_2 | ba_2 \in \mathsf{ac}(C_2)\}.$

$\mathsf{c2f}(*C) \mathrel{\widehat{=}}$
$\mathsf{c2f}_{pi}(*C) = (z \text{->} SKIP; END | z \text{->} \mathsf{c2f}_{pi}(C); SKIP; \mathsf{c2f}_{pi}(*C)) \backslash \{z\}.$
assuming z is not in $\alpha\mathsf{c2f}_{pi}(C)$.

We added $SKIP$ between operands of each sequential composition because of

which $[\![C]\!] = [\![C']\!]$ and $C' = skip$, $C' = skip \sqcap C''$, or $C' = C''$ (C'' has no *skip* in its specification).

the rule $\epsilon; P \xrightarrow{\langle\rangle} P$ in Table 2. We also put $z{\rightarrow}SKIP; END$ instead of $z{\rightarrow}END$ for encoding the loop operator because of the rules $*P \xrightarrow{\langle\rangle} skip$ and $skip \xrightarrow{\langle\rangle} \epsilon$ in that table. Each of these rules has a $\xrightarrow{\langle\rangle}$ transition, and we encode it using the additional $SKIP$ process. These additional SKIP processes are useful to preserve the semantics of peers while encoding them into FSP (see Theorem 4.5).

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 $Chor \nrightarrow ProcessIdentifier$ generating fresh identifiers (the same ones for identical Chor specifications) as output, which obey naming rules[6] 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.

In the following theorem we prove that the Chor semantics is preserved by the defined translation. Since the Chor semantics is defined using weak trace sets, we show that our encoding preserves the semantics of this language using Theorem 3.3.

Theorem 3.3 (Weak Trace Equivalence between Chor and FSP) *Given a Chor specification C, and the corresponding FSP process $\mathsf{c2f}_{pi}(C)$, two specifications are weak trace equivalent:*

$$[\![C]\!] = [\![\mathsf{c2f}_{pi}(C)]\!]$$

Proof. We prove this theorem using Lemma 2.3 and induction on the structure of operators in C. Basis of the induction ($C = skip$, $C = a^i$, or $C = c^{[i,j]}$) holds vacuously. Also for the cases C is equal to $C_1; C_2$ and $C_1 \sqcap C_2$ one can check that $[\![\mathsf{c2f}_{pi}(C)]\!]$ is equal to $[\![C_1]\!] \frown [\![C_2]\!]$ and $[\![C_1]\!] \cup [\![C_2]\!]$, respectively. When $C = C_1 \| C_2$ then $[\![T.\mathsf{c2f}_{pi}(C)]\!] = ([\![\mathsf{c2f}_{pi}(C_1)]\!]/R_1) \bowtie ([\![\mathsf{c2f}_{pi}(C_2)]\!]/R_2)$ and $[\![\mathsf{c2f}_{pi}(C)]\!] = [\![T.\mathsf{c2f}_{pi}(C)]\!]/R$, where R_1, R_2, and R are defined as follows:

$R_1 = \{(x, p1.x) | x \in \alpha\mathsf{c2f}_{pi}(C_1)\}$
$R_2 = \{(x, p2.x) | x \in \alpha\mathsf{c2f}_{pi}(C_2)\}$
$R = \{(p1.ba_1, ba_1) | ba_1 \in \mathsf{ac}(C_1)\} \cup \{(p2.ba_2, ba_2) | ba_2 \in \mathsf{ac}(C_2)\}$

R renames each activity to its original value (when R_1 or R_2 has not been applied). Since operator $\bowtie$ does not impose any synchronization on traces of its operands $[\![\mathsf{c2f}_{pi}(C)]\!] = [\![\mathsf{c2f}_{pi}(C_1)]\!] \bowtie [\![\mathsf{c2f}_{pi}(C_2)]\!] = [\![C_1]\!] \bowtie [\![C_2]\!] = [\![C]\!]$.

When $C = *C'$, we have $[\![\mathsf{c2f}_{pi}(C)]\!] = [\![SKIP]\!] \cup [\![\mathsf{c2f}_{pi}(C')]\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!] = \{\langle\rangle\} \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!]$. One could see, $\forall t \in [\![\mathsf{c2f}_{pi}(C)]\!] \centerdot t \in [\![C']\!] \frown \cdots \frown [\![C']\!] = [\![C']\!]^{(i)}$, for some $i \geq 0$. Therefore, we have $[\![\mathsf{c2f}_{pi}(C)]\!] \subseteq [\![C']\!]^*$. For the reverse direction, by induction on n, we prove that for all $n \geq 0$, $[\![\mathsf{c2f}_{pi}(C)]\!] = \bigcup_{0 \leq i \leq n} [\![C']\!]^{(i)} \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!]$. Basis of induction is $[\![\mathsf{c2f}_{pi}(C)]\!] = [\![C']\!]^{(0)} \cup$

[6] These rules are defined in Section 2 of Appendix B in [9].

$[\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!] = \{\langle\rangle\} \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!]$, which is trivially true. Thus, assuming $[\![\mathsf{c2f}_{pi}(C)]\!] = \bigcup_{0\leq i<n} [\![C']\!]^{(i)} \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!]$, for all $n > 0$, we prove $[\![\mathsf{c2f}_{pi}(C)]\!] = \bigcup_{0\leq i\leq n} [\![C']\!]^{(i)} \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!]$. By substituting the right hand side of the antecedent in the definition of $[\![\mathsf{c2f}_{pi}(C)]\!]$, the consequent is proved. Hence, we have: $[\![\mathsf{c2f}_{pi}(C)]\!] = [\![C']\!]^* \cup [\![C']\!] \frown [\![\mathsf{c2f}_{pi}(C)]\!] \Rightarrow [\![C']\!]^* \subseteq [\![\mathsf{c2f}_{pi}(C)]\!]$. Consequently, both $[\![C']\!]^*$ and $[\![\mathsf{c2f}_{pi}(C)]\!]$ are subsets of each other, and hence equal. □

Example 3.4 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 1 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 paper for the sake of comprehension.

Table 4
Some FSP Processes Generated for the Running Example

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

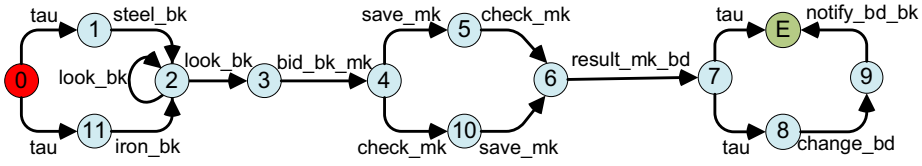

Fig. 1. Minimized LTS of the *Stock* Market Case Study

4 Peer Generation and Realizability

4.1 Peer Generation

Given a Chor specification, one can generate the specification of each Peer using *natural projection*. Natural projection[7] of a Chor specification to Peer P replaces each observable action with τ iff P does not perform that action.

Chor and Peer share parallel, sequential, choice, and loop operators. For these operators the natural projection first replaces each Chor operator by its equivalent in Peer, and then applies recursively to their operands. Projection of basic activities from a Chor specification C to a Peer specification P is achieved as follows:

(i) each activity not performed by P is replaced by *skip*,

(ii) a local activity performed by P remains unchanged,

(iii) 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 in Section 3. 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 4.1).

Definition 4.1 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 $\mathsf{c2f}_{pi}$):

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

As specified in [12] for projecting Chor to peers the name of each Peer 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 $\mathsf{p2f}_{pi}(C,1) \| \cdots \| \mathsf{p2f}_{pi}(C,n)$, 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.

Similarly to Section 3, to prove that our encoding method preserves the semantics of the Peer language, we need some preliminary definitions and a lemma. Instead of proving that our encoding method preserves trace semantics, we prove that the parallel composition of the encoded peers is strongly bisimilar to the parallel composition of the input peers, because it is a stronger result, yet easier to prove. The semantics of an FSP process F is defined using LTS ($\mathsf{lts}(F)$[8]), therefore in the following definitions, lemma, and theorem we use LTS instead of FSP (for FSP specifications we do not have a definition for the transition relation ("$\xrightarrow{a}$"), but

[7] The reader may refer to [12] for the formal definition of natural projection.

[8] The LTS semantics of FSP is defined in [9]. In this paper, we use LTSA to compute LTSs of different FSP specifications.

we need this relation in the bisimulation relation definition). A strong bisimulation relation between Peer and LTS is defined as follows:

Definition 4.2 [Strong bisimulation relation ($\sim$) between Peer and LTS] We set $\epsilon \sim \mathsf{lts}(END)$, and $\dagger \sim \mathsf{lts}(STOP)$ [9]. For any other Peer specification P and LTS L, we have $P \sim L$ if and only if the following two conditions hold [10]:

- $\forall b \,.\, P \xrightarrow{b} P' \Rightarrow \exists L' \,.\, L \xrightarrow{b} L' \wedge P' \sim L'$
- $\forall b \,.\, L \xrightarrow{b} L' \Rightarrow \exists P' \,.\, P \xrightarrow{b} P' \wedge P' \sim L'$

Communication activities can occur in peers only when they are composed with their complementary activities in parallel. For example, there is no transition relation defined for $c!$, but if it is composed with $c?$ in parallel, we will have $c!\|c? \xrightarrow{c} \epsilon\|\epsilon$. Hence, in order to compare behaviors of an individual Peer specification with its corresponding FSP process, we need another definition.

Definition 4.3 [Similar Behavior ($\sim'$) between Peer and LTS] We define the similar behavior relation between Peer P and LTS L as $\epsilon \sim' \mathsf{lts}(END)$, $\dagger \sim' \mathsf{lts}(STOP)$, and for all other peers which is neither ϵ nor $\dagger$ this relation is defined as follows:

$$
\begin{aligned}
&\forall P, L \,.\, P \sim' L \Leftrightarrow \\
&\quad (\forall c \,.\, c! \in \mathsf{fst}(P) \Rightarrow L \xrightarrow{c} L' \wedge P/c! \sim' L') \wedge \\
&\quad (\forall c \,.\, c? \in \mathsf{fst}(P) \Rightarrow L \xrightarrow{c} L' \wedge P/c? \sim' L') \wedge \\
&\quad (\forall c \,.\, L \xrightarrow{c} L' \Rightarrow (c! \in \mathsf{fst}(P) \wedge P/c! \sim L') \veebar (c? \in \mathsf{fst}(P) \wedge P/c? \sim' L')) \wedge \\
&\quad (\forall P' \,.\, P \xrightarrow{a} P' \Rightarrow \exists L' \,.\, L \xrightarrow{a} L' \wedge P' \sim' L') \wedge \\
&\quad (\forall L' \,.\, L \xrightarrow{a} L' \Rightarrow \exists P' \,.\, P \xrightarrow{a} P' \wedge P' \sim' L') \wedge \\
&\quad (\forall P' \,.\, P \xrightarrow{\tau} P' \Rightarrow \exists L' \,.\, L \xrightarrow{\tau} L' \wedge P' \sim' L') \wedge \\
&\quad (\forall L' \,.\, L \xrightarrow{\tau} L' \Rightarrow \exists P' \,.\, P \xrightarrow{\tau} P' \wedge P' \sim' L')
\end{aligned}
$$

Lemma 4.4 *For each Chor specification C and Peer p, we have* $\mathsf{nproj}(C,p) \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C,p))$.

Proof. For the sake of comprehension, we only show that the behavior of Peer p can be represented by the corresponding LTS (proof for the reverse direction is similar and hence straightforward). We use induction on the operators of $\mathsf{nproj}(C,p)$. Basis of induction (cases where $\mathsf{nproj}(C,p)$ is equal to ϵ, $\dagger$, a^p, $c^{[p,p']}$, or $c^{[p',p]}$) holds vacuously. Next, we prove the following inductive step:

$$
\begin{aligned}
&P = \mathsf{nproj}(C,p) = P_1; P_2 \Rightarrow \\
&\quad \exists C_1, C_2 \,.\, C = C_1; C_2 \wedge \mathsf{nproj}(C_1,p) = P_1 \wedge \mathsf{nproj}(C_2,p) = P_2 \Rightarrow \\
&\qquad \mathsf{nproj}(C_1,p) \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C_1,p)) \wedge \mathsf{nproj}(C_2,p) \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \Rightarrow \\
&\qquad\quad \mathsf{lts}(\mathsf{p2f}_{pi}(C,p)) = \mathsf{lts}(\mathsf{p2f}_{pi}(C_1,p)); \mathsf{lts}(SKIP); \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \\
&\text{If } P_1 = \epsilon \Rightarrow P \xrightarrow{\tau} P_2 \wedge \mathsf{lts}(\mathsf{p2f}_{pi}(C_1,p)) \sim \mathsf{lts}(END) \Rightarrow \\
&\quad \mathsf{lts}(\mathsf{p2f}_{pi}(C,p)) \sim \mathsf{lts}(SKIP); \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \wedge \\
&\quad \exists L' \,.\, \mathsf{lts}(\mathsf{p2f}_{pi}(C,p)) \xrightarrow{\tau} L' \wedge L' \sim \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \Rightarrow P' \sim' L'.
\end{aligned}
$$

[9] $\mathsf{lts}(STOP)$ has one state which is initial but not final, and no transition.

[10] Bisimulation relations between two peers or between two LTSs [9] are similar and straightforward, therefore they are not presented here.

If $P_1 \neq \epsilon \Rightarrow$
$P \xrightarrow{a} P' \Rightarrow P_1 \xrightarrow{a} P_1' \Rightarrow \exists L_1' \,.\, \mathsf{lts}(\mathsf{p2f}_{pi}(C_1,p)) \xrightarrow{a} L_1' \wedge L_1' \sim' P_1' \Rightarrow$
$\exists L' \,.\, \mathsf{lts}(\mathsf{p2f}_{pi}(C_1,p)); \mathsf{lts}(SKIP); \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \xrightarrow{a} L' \wedge$
$L' \sim L_1'; \mathsf{lts}(SKIP); \mathsf{lts}(\mathsf{p2f}_{pi}(C_2,p)) \Rightarrow P' \sim' L'$

Proofs for the cases $c! \in \mathsf{fst}(P)$, $c? \in \mathsf{fst}(P)$, or $P \xrightarrow{\tau} P'$, and cases $P = P_1 \| P_2$, $P = P_1 \sqcap P_2$, or $P = *P_1$ are similar and omitted here. □

Theorem 4.5 *For each Chor C with n peers, $PS = \mathsf{nproj}(C,1) \| \cdots \| \mathsf{nproj}(C,n)$ is strongly bisimilar ($\sim$) with $L = \mathsf{lts}(Peers2)$, where $Peers2 = Peers; SKIP; END$. and $\| Peers = (\mathsf{p2f}_{pi}(C,1) \| \cdots \| \mathsf{p2f}_{pi}(C,n))$.*

Proof. If $PS = \epsilon \| \cdots \| \epsilon$ then $\forall 1 \leq i \leq n \,.\, \mathsf{lts}(\mathsf{p2f}_{pi}(C,i)) \sim \mathsf{lts}(END)$, thus one can see $\mathsf{lts}(\mathsf{p2f}_{pi}(C,1) \| \cdots \| \mathsf{p2f}_{pi}(C,n)) \sim \mathsf{lts}(END)$. Hence $PS \xrightarrow{\tau} \epsilon$ and $L \xrightarrow{\tau} L' \wedge L' \sim \mathsf{lts}(END)$. For all the other cases in which $PS \neq \epsilon \| \cdots \| \epsilon$ proof is as follows:
For all PS' such that $PS \xrightarrow{b} PS'$ (b stands for a local activity or τ)
$\exists 1 \leq i \leq n \,.\, \mathsf{nproj}(C,i) \xrightarrow{b} \mathsf{nproj}(C,i)' \Rightarrow$
$\mathsf{lts}(\mathsf{p2f}_{pi}(C,i)) \xrightarrow{b} \mathsf{lts}(\mathsf{p2f}_{pi}(C,i))' \wedge \mathsf{nproj}(C,i)' \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C,i))' \wedge$
$b = \tau \vee \nexists j \neq i \,.\, b \in \alpha \mathsf{p2f}_{pi}(C,j) \Rightarrow \exists L' \,.\, L \xrightarrow{b} L' \wedge PS' \sim L'$ (induction)
For all PS' such that $PS \xrightarrow{c} PS'$ (c stands for a communication activity)
$\exists 1 \leq i,j \leq n . i \neq j \wedge \mathsf{nproj}(C,i)/c! = \mathsf{nproj}(C,i)' \wedge \mathsf{nproj}(C,j)/c? = \mathsf{nproj}(C,j)' \Rightarrow$
$\mathsf{lts}(\mathsf{p2f}_{pi}(C,i)) \xrightarrow{c} \mathsf{lts}(\mathsf{p2f}_{pi}(C,i))' \wedge \mathsf{nproj}(C,i)' \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C,i))' \wedge$
$\mathsf{lts}(\mathsf{p2f}_{pi}(C,j)) \xrightarrow{c} \mathsf{lts}(\mathsf{p2f}_{pi}(C,j))' \wedge \mathsf{nproj}(C,j)' \sim' \mathsf{lts}(\mathsf{p2f}_{pi}(C,j))' \wedge$
$\nexists k \neq i,j \,.\, c \in \alpha \mathsf{p2f}_{pi}(C,k) \Rightarrow \exists L' \,.\, L \xrightarrow{c} L' \wedge PS' \sim L'$ (induction)
Proof of the reverse direction is similar to the forward direction, and is omitted here. □

Corollary 4.6 *Bisimulation is a stronger relation than (weak) trace equivalence, thus for each Chor specification C with n peers the following relation holds between Peer specifications and their corresponding FSP specifications:*

$$[\![\mathsf{nproj}(C,1) \| \cdots \| \mathsf{nproj}(C,n)]\!] = [\![\mathsf{p2f}_{pi}(C,1) \| \cdots \| \mathsf{p2f}_{pi}(C,n)]\!]$$

4.2 Realizability

Definition 4.7 formalizes the notion of choreography realizability we use in this paper. We chose a strong realizability [2,7] for experimentation purposes, but weak notions could be used instead [7].

Definition 4.7 [Realizability of Chor under Natural Projection] 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:

(i) $[\![C]\!] = [\![\mathsf{nproj}(C,1) \| \cdots \| \mathsf{nproj}(C,n)]\!]$

(ii) $\nexists t \,.\, \mathsf{nproj}(C,1) \| \cdots \| \mathsf{nproj}(C,n) \stackrel{t}{\Longrightarrow} \dagger$

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 Theorem 3.3 that the trace set of a Chor specification is equal to the trace set of its FSP encoding. We showed in Corollary 4.6 that our encoding also 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 specification of Chor 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.

Magee and Kramer explained in [9] how some Java code can be generated from FSP. Therefore, in our approach, these guidelines can be followed for rapid prototyping purposes to implement real peers from their FSP specifications, and deploy them in the context of a concrete system. In Section 5, we will show on our running example how Java code is automatically generated. Implementation of executable services from abstract descriptions could also be achieved in BPEL using Pi4SOA technologies [1] or following guidelines presented in [10].

Example 4.8 For each Peer P, all actions in $\mathsf{c2f}_{pi}(\mathsf{Stock})$ in which P is not involved, are hidden. The three peers of our example are encoded by the following FSP specifications:

1. $\mathsf{Broker} = \mathsf{c2f_{pi}(Stock)}; \mathsf{END@\{iron\_bk, steel\_bk, bid\_bk\_mk, look\_bk, notify\_bd\_bk\}}.$
2. $\mathsf{Market} = \mathsf{c2f_{pi}(Stock)}; \mathsf{END@\{save\_mk, check\_mk, bid\_bk\_mk, result\_mk\_bd\}}.$
3. $\mathsf{Board} = \mathsf{c2f_{pi}(Stock)}; \mathsf{END@\{result\_mk\_bd, notify\_bd\_bk, change\_bd\}}.$

Figure 2 shows the minimized LTSs of these peers generated from the FSP processes presented above.

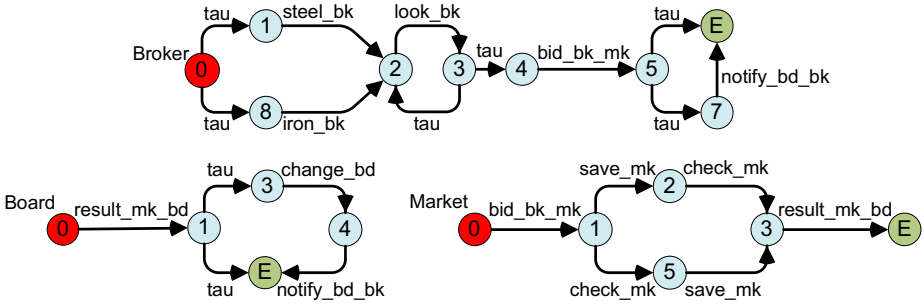

Fig. 2. *Stock* Market: Minimized LTSs of Peers

Finally, we run all these peers concurrently. The FSP process for the whole system is: $\|\mathsf{Peers} = (\mathsf{Broker}\|\mathsf{Market}\|\mathsf{Board})$. As for the realizability test, first, we compute LTSs from FSP processes Stock and Peers, using LTSA. Next, we compare trace sets of these processes using *ltscompare*, one of the tools belonging to

the mCRL2 toolset [6], and find out they produce the same set of traces (first realizability condition, Definition 4.7). For a Chor specification to be realizable, it is also required to satisfy the second condition of Definition 4.7. 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:

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

This new specification satisfies both realizability conditions.

5 Prototype Tool and Code Generation

All the steps of the approach we have presented in Sections 3 and 4 are automatically computed by a prototype tool we implemented (see Figure 3 for an overview) and tested on more than 300 examples of different sizes. Checking deadlock-freeness costs less than checking weak trace equivalence. Therefore, in our prototype we first check deadlock-freeness and then trace equivalence (if no deadlock was found).

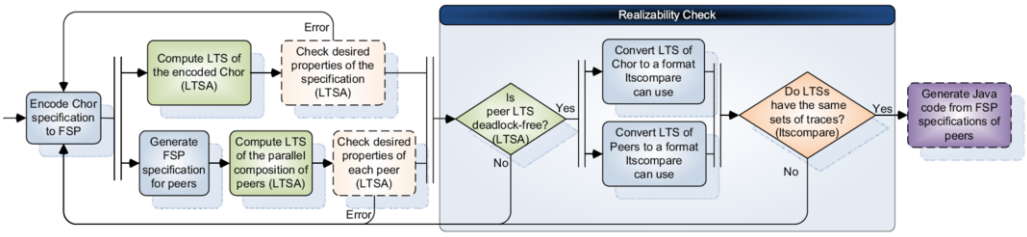

Fig. 3. Prototype Tool Overview

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 Chor specification, and respectively presents the number of: peers, activities used in the Chor specification, FSP processes resulting while encoding the Chor specification into FSP, states and transitions in the LTS corresponding to the Chor specification, states and transitions in the LTS

computed from the parallel composition of the Peer specifications, result of the deadlock-freeness check, result of the weak trace equivalence check, time and memory used to verify realizability. Numbers of states and transitions have two parts, before and after minimization using LTSA. To compute the LTS of the parallel composition of peers we first compute the minimal LTS for each peer and then compose them using the parallel operator. We perform this minimization because there are lots of τ transitions in peer LTSs and it may cause state explosion if we compose them in parallel without applying minimization before. Each minimized LTS is weakly bisimilar to its original LTS [9].

Whenever a deadlock-freeness check fails, there is no need to check the weak trace equivalence (marked by "–" in Table 5). Note that we use LTSA to obtain the parallel composition of Peer LTSs, and this computation has to be completed before performing the deadlock-freeness check. In some cases, this computation can be very costly in time and memory as it can be observed in the last row of Table 5 (almost 10 minutes to obtain the Peer LTS).

Table 5
Experimental Results

P	Act.	FSP Proc.	Chor to FSP States	Chor to FSP Trans.	All Peers in Parallel States	All Peers in Parallel Trans.	D F	W T	Time	Mem. (MB)
3	10	14	20/10	19/9	22/22	33/33	√	×	0.32s	2
6	37	51	86/46	101/57	50/46	61/57	√	√	1.30s	3
3	10	15	21/10	23/12	76/62	178/206	×	–	0.65s	3
7	133	155	302/163	353/207	176/163	220/207	√	√	5.95s	8
7	229	224	518/280	605/357	302/280	379/357	√	√	22.17s	14
9	420	330	972/509	1147/640	553/509	684/640	√	√	11m9.56s	51
4	15	22	30/14	38/18	2688/1271	11520/17112	×	–	3.86s	14
5	20	32	40/18	46/24	34958/6021	195586/131921	×	–	9m40.52s	308

5.2 Code Generation

As mentioned earlier, the final step is to produce some Java code following guidelines presented in [9]. Like the other steps, this is completely automated by our tool. Figure 4 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 behavior and should not be changed. The user only needs to implement interfaces of peers and distributes classes to different locations, as needed.

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.

```
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);}}
```

Fig. 4. *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 [14], 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 [11], 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 some recent

papers [12,8], 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 [4,12] 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.

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 was presented in [13] and showed on an example how realizability can be checked using a LOTOS encoding. However, in [13] the choreography language, namely collaboration diagrams, was less expressive than Chor (no choice and a loop operator restricted to a single message), and the proposal focused only on abstract languages (no relationships with implementations or real code).

To sum up, first, most of these approaches focus on theoretical aspects. Our contribution is a tool-supported yet formal approach tackling the realizability issue for choreography, but considers a different choreography specification language compared to [5,13], and therefore deals with its own specificities. Second, the works presented in this section focus on the realizability issue but do not allow to check choreography specifications or generate code from such specifications, whereas our FSP encoding makes it possible.

7 Concluding Remarks

In this paper, 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. 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 paper, we have focused on synchronous communication, and it makes the realizability computation and model-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 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.

Acknowledgement

This work has been partially supported by project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), and project P06-TIC-02250 funded by the *Junta de Andalucía*.

References

[1] Pi4SOA Project. `www.pi4soa.org`.

[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] 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.

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

[10] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.

[11] 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.

[12] 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.

[13] G. Salaün and T. Bultan. Realizability of Choreographies using Process Algebra Encodings. In *Proc. of IFM'09*, LNCS, pages 167–182. Springer, 2009.

[14] 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.