Verification of Channel Passing in Choreography with Model Checking*

Liyang Peng†, Chao Cai†, Geguang Pu‡ and Zongyan Qiu†

†LMAM and Department of Informatics, School of Math.,
Peking University, Beijing, China
‡Software Engineering Inst., East China Normal University, Shanghai, China
Email: {ply, calc, qzy} @math.pku.edu.cn, ggpu@sei.ecnu.edu.cn

Abstract. A web service choreography describes a global protocol of interactions among a set of cooperating services. For the dynamic composition, changing interconnections by channel passing between services is necessary. In this paper we use model checking technique for the verifying problems related to channel passing in choreography. We develop a framework: for each kind of property to be verified, we define an abstraction function based on it, which map each basic interaction into a pair of pre- and post-conditions, then propose a compositional approach to translate choreographies into models for model checkers. A number of examples are presented to show how the verification is carried out. Additionally, we have an example showing that some constructive work can also be done with model checking technique.

Keyword: Web service composition, Choreography, Channel passing, Abstraction method, Model checking

1 Introduction

Web services are platform-independent software components available in the Internet. A web service is an autonomous, standards-based component whose public interfaces are defined and published. Several web services can be assembled to provide a value-added service. This design paradigm, called web services composition, has enormous potential in the intra- and crossing-organization application integration and attracts ever increasing attentions from researchers and developers.

Two different viewpoints about web services composition exist, i.e. from one party’s view or from a global view, called the web service orchestration or choreography respectively, while the latter is the focus here. In choreography, observable activities, including the interactions, are described from the viewpoint of an ideal observer who oversees all interactions between the participating services, Web Service Choreography Description Language (WS-CDL) [1], developed by a W3C working group, is a typical language for describing the global models of this kind. Such a choreography, will be implemented by a group of individual participating services without a center control.

In the practical web environment, a service may acquire or lose connections to other services on the fly. Think about a simple scenario involving a buyer, a seller and a shipper. The shipper cannot have the buyer’s connection unless another service tells it after

* Supported by National Natural Science Foundation of China (60773161,60603033).
a deal between the buyer and the seller has been confirmed. Thus, a language for choreography should support the exchanging of connections between services. WS-CDL achieves this goal by allowing the channels to be passed in the interactions. Clearly, channel passing plays a crucial role to support dynamic service composition.

People have different viewpoints about the concept of channel. What we assume here is one kind of them, which is widely accepted in the web field. We stipulate that a channel is bound to a special service, its messages’ receiver. In some circumstance, it might be called port or address. By channel passing, we mean that a service knowing a channel can pass the channel information to another service, then the latter can send message via the channel to the channel’s receiver. Many web standards take this viewpoint, e.g., WS-Addressing, WS-BPEL and WS-CDL. All the discussions presented here follow this definition.

In design of web applications from a global viewpoint involving many participants, complex channel-related constraints need to be enforced. For examples:

1) Channel-absence: A channel is not available when it is used in an interaction, where either the interaction is carried out via the channel, or the channel is passed to another party in the interaction. In this case, the execution gets stuck.

2) Channel-used-once: Some channel is allowed to be used in only one interaction by only one participant in the whole process, although it can be passed among participants several times. This feature is included in WS-CDL.

3) Channel-redundancy: During the execution, some channel might be passed to a participant who never uses it. This is a potential safety leak when some important channel is passed carelessly to some unprivileged participants.

In case of channel-absence, a repairing method is useful, i.e. generating necessary channel-passings based on the given choreography. This could be thought as an advice for the developers to fix channel-absence problems.

Other requirements for channels exist which need to be checked. It is not possible to enumerate all of them here. In this paper, we just take the problems above as examples to show a general checking approach. Other relevant problems might be checked following this approach.

Experience shows that the design of channel passing is an error-prone work [4], thus effective formal verification techniques are valuable. Some approaches have been suggested for this verification, e.g., the algorithms proposed in [14]. The disadvantage of the algorithmic approach is that it is not general. For each individual problem, a specific algorithm should be designed and justified. It is not easy to design correct and efficient algorithms to verify the problems related to channel passing. Moreover, understanding and justifying the algorithms is also a difficult work.

Model checking is a more general way to verify systems. A choreography text may be translated into an abstract model, and the properties to be checked may be encoded as logic formulae. Then they are fed into a model checker. The checker may give a positive answer, or a negative answer with some heuristic execution trajectory which can be used as a clue for fixing the problem.

Model checkers conduct an exhaustive exploration for all possible states of the model, so reducing the state space is the key point for efficient use of the technique for real problems. To make it usable, the model should capture all the information nee-
ecessary for verifying the aimed properties. On the other hand, the irrelevant details which do not effect the correctness of the required properties should be ruled out as much as possible from the model. In this paper, we design and use an abstraction method which is close related to the checked properties.

In this paper, we use one of state-of-the-art model checkers, SPIN [12]. In SPIN, a model is described as a couple of asynchronous processes in PROMELA language. Some researchers suggested to translate a WS-CDL choreography into PROMELA processes by using a process for each participant, and using a communication on some channel between a pair of participating processes for each interaction in the choreography [6, 8]. In that way, the PROMELA model for a large set of collaborative services might consist of a large set of communicating processes, that would make heavy burden to the verification, because the number of global system states grows exponentially with the number of processes. On the contrary, recognizing the essence of choreography, we use a pair of pre- and post-conditions to represent the effect of an interactions, therefore, many interactions among participants can be represented within a single PROMELA process, thus reduce remarkably the number of processes in the model. As seen later, in our translation, the control flow in PROMELA processes is determined by the control flow in the choreography, while the state variables in PROMELA processes used by the pre- and post conditions are mainly determined by the properties to be checked.

In general, model checking is incapable of doing constructive works. However, when a model does not satisfy the checked property, the model checker can provide a possible execution path showing how the property is violated. We can exploit this feature to do some constructive works, such as finding a path (a counter-example execution path for some checked property) along which the channel-absence can be repaired by adding channel-passings to some interactions. We will show details of this technique in Section 4.

This paper is structured as follows. In Section 2, a small web service choreography description language Chor is defined, which can be seen as a simplified WS-CDL, then we discuss how the semantics of basic interactions in Chor can be denoted by a pair of pre- and post-conditions. Afterwards, we present a compositional approach which is used to translate a Chor choreography into a model. In Section 3, some examples are given to show how the verification is carried out. In Section 4 we illustrate how a channel-absence problem may be repaired by finding the execution paths along which the absent channel might be passed in succession. In the end, we give a summary of related works, and draw some conclusions.

2 Constructing Model for Choreography

In this section we present a systematic way by which a choreography with channel passing can be modeled and verified. In the first, a choreography language Chor, based on WS-CDL is defined, then an abstraction method which maps each interaction into a pair of pre- and post-condition is given. At last we develop a compositional approach for constructing a verification model from a Chor choreography.
2.1 The \textit{Chor}c Language

Now we define the simple language \textit{Chor}, which includes basic features for describing choreographies, and supports channel passing in particular. We begin with its syntax, and then describe its semantics informally.

In a choreography, a \textit{role} is an abstraction for web services that represents a class of services with common behavioral patterns. So there are a fixed number of roles in a given application scenario. We assume that meta-variables $r_i$, $r_j$ range over role names, $x$ ranges over global variable names, $e$ ranges over general expressions, and in particular, $p$ ranges over boolean expressions. The data types considered here are integer and boolean. The definitions of general expressions and boolean expressions are routine, so we omit their details here. We stipulate that each role has only one channel whose target is that role, so use $ch_i$ to represent the channel whose target is $r_i$. In a sense, the channel we defined here is similar with an URL address. Note that, the verification method introduced in this paper can be easily extended to the situation where multiple channels may have a common target, in that case, we can use, e.g., $ch_i[1], ch_i[2], \ldots$ to represent the channels which all have $r_i$ as their common target.

The basic activities in \textit{Chor}c are as follows:

$$BA ::= \text{skip} \mid x = e \quad \text{(assignment)}$$
$$\mid r_i \rightarrow r_j \quad \text{(simple interaction, via $ch_j$)}$$
$$\mid r_i ch_k \rightarrow r_j \quad \text{(interaction with channel passing, via $ch_j$)}$$

Here we have \textit{skip}, assignment, and two forms of interactions. We omit other local activities of roles because they are not important in this study. The \textit{skip} activity does nothing and always terminates successfully, assignment $x = e$ assigns the value of expression $e$ to the variable $x$. The activity $r_i \rightarrow r_j$ denotes an interaction between the source role $r_i$ and the target role $r_j$, where no channel passing in the interaction is done and the necessary channel which $r_i$ should know is $ch_j$. The activity $r_i ch_k \rightarrow r_j$ is an interaction from role $r_i$ to role $r_j$, where the information about channel $ch_k$ is passed from $r_i$ to $r_j$, and the necessary channels which $r_i$ should know are $ch_j$ and $ch_k$. These interactions would be \textit{blocked} if the channels they need are unavailable. We do not take care of the general data transferred in interactions, because it makes no difference to the focus of this paper.

The syntax of the compound activities is:

$$A, B ::= BA \quad \text{(basic activity)}$$
$$\mid A; B \quad \text{(sequence)}$$
$$\mid A \cap B \quad \text{(non-deterministic)}$$
$$\mid A \parallel B \quad \text{(parallel)}$$
$$\mid p ? A \quad \text{(condition)}$$
$$\mid p * A \quad \text{(iteration)}$$

Sequential composition $A; B$ behaves first like $A$. When $A$ terminates successfully, it continues like $B$. If $A$ never terminates, neither does $A; B$. The non-deterministic
choice \( A \sqcap B \) behaves like either \( A \) or \( B \), where the selection between two branches is internal and non-deterministic. The parallel composition \( A \parallel B \) enables both \( A \) and \( B \) concurrently in an asynchronous manner. It completes only if both the two branches complete successfully.

The behavior of the condition activity \( p ? A \) is the same as \( A \) when the boolean expression \( p \) evaluates to true. Otherwise, it does nothing and terminates immediately. The repeat activity \( p * A \) is executed by first evaluating \( p \). When \( p \) is false, the activity terminates immediately, otherwise, the structure \( A; p * A \) will be executed.

2.2 Abstraction for interaction

For model checking to be carried on, we need to construct a model for the system in the first. As a general principle, the model should preserve the necessary information of the system for the verification, and all irrelevant details can be abstracted.

In this subsection, we describe our abstraction method for the model construction of choreography, in which basic interactions are not translated into communications between roles, as suggested by other works, for example [6], because that leads to a model with big state space (the number of global system states grows exponentially with the number of processes), thus is not scalable. Instead, we define an abstraction function which maps each basic interaction into a pair of pre- and post-conditions referring to some state variables relevant to the properties to be checked and reflecting the effect of the interaction. This abstraction method preserves the properties of the system, and generates a model with much smaller state space (the number of global system states is directly proportional to the number of the interactions, largely irrelevant to the number of processes). The abstraction technique used here is called cone-of-influence reduction [5].

Now we use some examples to illustrate how the abstraction function can be defined w.r.t. concrete properties.

Example 1 (Channel-absence). Now we consider the channel-absence problem, i.e., verifying that a choreography would not run into stuck due to unavailable channels required in some interactions. We use boolean variable \( K^t_j \) to represent that role \( r_i \) knows channel \( ch_j \). The initial values of these boolean variables depend on the initial channel configuration of the choreography. It is easy to understand that \( K^t_i \) is true for each \( i \) all the time. To enable the interactions in the initial configuration, some role should know additional channel(s), otherwise, the choreography will get stuck immediately.

With the boolean variables introduced above, we define a pair of pre- and post-conditions for each interaction in the choreography. For checking this property, the pre-condition for \( r_i \rightarrow r_j \) is \( K^t_j \), that means role \( r_i \) must know channel \( ch_j \) before the interaction, because otherwise \( r_i \) cannot launch this interaction. On the other hand, the post-condition of \( r_i \rightarrow r_j \) is unchanged, which mean that none of the boolean variables are affected by the simple interaction. Similarly, the pre-condition for \( r_i \xrightarrow{ch_k} r_j \) is \( K^t_i \land K^t_j \), and the post-condition is \( K^t_j \), that means role \( r_j \) would know channel \( ch_k \) after this interaction. When the precondition for some interaction is not satisfied, a channel-absence event occurs, and the execution will deadlock, which can be detected by SPIN as an invalid end.
Example 2 (Channel-used-once). For the channel-used-once problem, suppose we want to verify that a channel, say $ch_a$, should be used in at most one interaction, and never more. Here “some interaction uses channel $ch_a$” means that the interaction is conducted via $ch_a$, i.e., with role $r_a$ as the destination. To verify this property, we introduce a boolean variable $ChUsed_a$ to denote whether $ch_a$ has been used in some interaction. This method can be directly extended to the case where several used-once channels exist. This kind of variables are set to false initially. When channel $ch_a$ is used, the value of $ChUsed_a$ becomes true.

The pre-condition for both $r_i \rightarrow r_a$ and $r_i \xrightarrow{ch_a} r_a$ is $\neg ChUsed_a$, and the post-condition is $ChUsed_a$. For all other interactions, both conditions are true. When the choreography attempts to use $ch_a$ again, the corresponding precondition cannot be satisfied and the execution deadlocks, which can be caught by SPIN as an invalid end.

Example 3 (Channel-redundancy). Now we consider the channel-redundancy problem, i.e., verifying that there is no role in a choreography which knows some channel but never uses it in execution. Here “a role uses a channel” means that the role launches an interaction via the channel or passes it to another role in an interaction. This meaning of “use” here is slightly different from the meaning in above example.

To represent the effect of an interaction here, we introduce a pair of boolean variables for each interaction, where $K_{ji}^i$ means the same as in Example 1, and $U_{ji}^i$ means that $r_i$ has used $ch_j$ in some interaction in execution. Both variables ($K_{ji}^i$ and $U_{ji}^i$) have persistence, that is, if they become true, they would preserve true forever. The initial values of variables $K_{ji}^i$ are set according to the design, and the initial values of $U_{ji}^i$ are all set to false. We take the pre-condition for $r_i \rightarrow r_j$ as $K_{ji}^i$, and the corresponding post-condition as $U_{ji}^i$. The pre-condition for $r_i \xrightarrow{ch_j} r_j$ is $K_{ji}^i \land K_{kj}^i$, and the post-condition is $K_{kj}^i \land U_{ji}^i \land U_{ki}^i$.

In fact, we can distinguish two different versions of channel-redundancy problems.

In term of the strong explanation, a choreography has no channel-redundancy if for every possible execution path, any role $r_i$ knowing channel $ch_j$, either initially or receiving in some interaction, uses it in some interaction afterward. In term of the weak explanation, a choreography has no channel-redundancy if, for any role $r_i$ knowing a channel $ch_j$, there exists at least an execution path started from that “knowing point”, along that path, there is an interaction in which $r_i$ uses channel $ch_j$.

For the strong explanation, the non-redundancy property can be expressed as $\Box(K_{ji}^i \rightarrow \Diamond U_{ji}^i)$ in LTL. For the weak explanation, we meet a problem because “the existence of some path” cannot be expressed in LTL. In CTL, it can be expressed as $AG(K_{ji}^i \rightarrow EFU_{ji}^i)$, here $AG$ and $EF$ are the CTL temporal connectives. This is a typical example which CTL can express and LTL cannot, and because SPIN does not support CTL claims, so we omit it in this paper.

In these examples, we illustrate how the effect of interactions can be abstracted to pairs of pre- and post-conditions, and show how the conditions can be constructed using some state variables. People may doubt why channel passings can be coded in this way. In fact, this is because on the choreography level, we stand outside the roles to arrange their behavior. On this level, an interaction behaves as a cross-roles assignment,
thus, the channel passing seems similar to address-passing, which makes verification of channel-relative properties much easier.

Once we have coded the interactions, it is not difficult to translate them to PROMELA statements. Because guard statements are supported in PROMELA, we can use the pre-condition directly as the guard, followed with one or more assignments which would make the post-condition true. Take the channel-redundancy problem as an example, for an interaction, say $r_i \xrightarrow{ch} r_j$, whose pre- and post-conditions are $K^1_i \wedge K^k_i$ and $K^k_j \wedge U^1_i \wedge U^k_i$ respectively, the corresponding PROMELA statements are as follows:

$$K^1_i \wedge K^k_i \rightarrow K^k_j = \text{true}; U^1_i = \text{true}; U^k_i = \text{true};$$

### 2.3 Translation from Chor\(_c\) to PROMELA

Having expressed semantics of the basic interactions in Chor\(_c\) as value-changes of state variables selected according to checked properties, in this subsection, we show the translation of compound structures in Chor\(_c\) into PROMELA in a compositional way, then the model checking can be carried on. For an activity in a choreography, say $A$, we use $P_A$ to represent its corresponding PROMELA constructs:

1. For the interaction in Chor\(_c\), the translation has been discussed in the last subsection.
2. For **skip** or $x = e$ activity, the translation is direct.
3. For sequential composition $A; B$, the corresponding PROMELA statement is $P_A; P_B$.
4. For choice $A \sqcup B$, the corresponding PROMELA structures is in the form:

   
   if
   :: \text{true} \rightarrow P_A;
   :: \text{true} \rightarrow P_B;
   fi;

   If $P_A$ or $P_B$ are too long, we can define two inline processes for $P_A$ and $P_B$ respectively. Because inline processes in SPIN are not really asynchronous processes, but only abbreviations for statements, there is not extra burden with this change of description forms for the execution of the model checking.
5. For $A \parallel B$, firstly, we need to construct two processes from $P_A$ and $P_B$ respectively, where each with a name, namely $PNameA$ and $PNameB$. In this construction, we need to add for each process a new state variable to indicate the termination of the process. For $P_A$, the corresponding process takes the form as:

   proctype $PNameA()$
   
   $PNameA.fin = \text{false};$
   $P_A;$
   $PNameA.fin = \text{true};$
   
   }
Having the two processes for $A$ and $B$ respectively, we can further construct the corresponding PROMELA structure for $A \parallel B$ as follows.

```plaintext
atomic {
    run PName_A;
    run PName_B;
}

PName_A.fin && PName_B.fin;
```

In SPIN, to “run” a process means activating an instance of the process type, and all active processes, including the main process, are executed asynchronously. The last statement above asks that the main process could not continue until both the concurrent sub-processes have terminated normally.

6. The translation for $p ? A$ is direct and simple, that is:

```plaintext
if

:: p \rightarrow P_A;
:: \text{else} \rightarrow \text{skip};
fi
```

7. The translation for $p * A$ is also direct:

```plaintext
do

:: p \rightarrow P_A;
:: \text{else} \rightarrow \text{break};
od
```

These translation rules can be thought as a definition of an abstract semantics for $\text{Chor}_c$ by the resulting PROMELA processes. On the other hand, based on these rules, we have implemented a translator from WS-CDL programs to PROMELA processes, and conducted some experiments on a number of sample WS-CDL programs. That proves that our approach is effective and efficient.

## 3 Model checking about Channel Passing

Now we give some examples to demonstrate applications of model checking in verifying properties about channel passing in $\text{Chor}_c$ choreographies. To keep the uniformity of the notation and improve the readability, we still use the superscript and subscript form to name the introduced state variables. In real implementation, we may replace $K_a^b$ with $K_{a,b}$ in the translation result.

### 3.1 Checking the Channel-absence

We use the following $\text{Chor}_c$ choreography to show how the channel-absence problem in choreography is checked:

```plaintext
(r_a \xrightarrow{ch} r_b \parallel r_a \rightarrow r_c); (r_b \rightarrow r_a \land r_c \rightarrow r_a)
```

(1)
We assume initially, $r_a$ knows both $ch_b$ and $ch_c$, and on the other hand, $r_b$ or $r_c$ knows only the channel of itself.

The corresponding SPIN model is shown in List 1, using the translation presented above. Here proctype par1 and par2 are types of the PROMELA processes corresponding to the two parallel branches in (1) respectively. Two boolean variables par1_fin and par2_fin identify the end of each process respectively. Only if both of them terminate, the main process would continue to choose non-deterministically one from two possible interactions. If all above terminate, the main process end normally, otherwise it is blocked. The latter case is called an invalid end, which SPIN can detect in the verification run.

Back to our example, SPIN will discover an invalid end and write a counterexample into a trail file. Based on the file, SPIN can conduct a guided simulation run to show how the problem is caused. The output produced by SPIN is referred to List 2, where some abbreviation is made for layout purpose. Here, after both par1 and par2 complete, the counterexample selects the statement corresponding to interaction $r_c \rightarrow r_a$ from the choice structure (step 12, proc 0, line 8), where precondition $K_a^c$ is not satisfied and therefore the execution is deadlocked.
Starting main with pid 0
0: proc - (:root:) creates proc 0 (main)
Starting par1 with pid 1
1: proc 0 (main) creates proc 1 (par1)
1: proc 0 (main) line 4 . . . [(run par1())]
Starting par2 with pid 2
2: proc 0 (main) creates proc 2 (par2)
2: proc 0 (main) line 4 . . . [(run par2())]
3: proc 2 (par2) line 17 . . . [par2_fin = 0]
4: proc 1 (par1) line 12 . . . [par1_fin = 0]
5: proc 1 (par1) line 13 . . . [(K_a^b)]
6: proc 2 (par2) line 18 . . . [(K_a^b)]
7: proc 1 (par1) line 13 . . . [K_a^b = 1]
8: proc 2 (par2) line 18 . . . [(1)]
9: proc 1 (par1) line 14 . . . [par1_fin = 1]
10: proc 2 (par2) line 19 . . . [par2_fin = 1]
11: proc 0 (main) line 5 . . . [(par1_fin&&par2_fin)]
11: proc 2 (par2), proc 1 (par1) terminate
12: proc 0 (main) line 6 . . . [(1)]
    timeout
    #processes: 1
    . . . , K_a^c = 0, . . .
12: proc 0 (main) line 8 . . .

List 2: Output for List 1: Channel-absence

3.2 Checking the Channel-used-once

The example choreography is as follows, where ch_a is assumed a used-once channel:

\[
(r_a \xrightarrow{ch_a} r_b); i = 0;
(i < 2) \ast ((r_b \rightarrow r_a \land r_b \rightarrow r_c); i = i + 1)
\]

The SPIN model produced is shown in List 3.

After a verification run, SPIN detects an invalid end and write the counterexample. Using that counterexample, SPIN can do a guided simulation, where we can see, after \(r_b \rightarrow r_a\) in the loop body is chosen to execute at the first time, ChUsed_a becomes true, and when the same interaction is chosen at the second time, precondition ChUsed_a == false is not satisfied, so the execution is blocked.

3.3 Checking the Channel-redundancy

The example choreography is as follows, where we suppose role r_a knows channel ch_b initially:

\[
\text{flag} = \text{false};
(\text{flag} \land r_a \rightarrow r_b) \parallel (\text{flag} = \text{true})
\]

The SPIN model produced is shown in List 4.
int i;
active proctype main()
{
bool ChUsed_a = false;
true → true;
i = 0;
do:: i < 2 →
  if:: true →
    { ChUsed_a == false →
      ChUsed_a = true};
  :: true → true:
fi;
i = i + 1;
:: else → break;
od;
}

List 3: Example for Channel-used-once

Here we want to verify that this choreography has no channel-redundancy in term of strong explanation (ref. Subsection 2.2), that is expressed as LTL formula □(K^b_a → ♦U^b_a), which, in turn, will be translated to a never claim by SPIN. The never claim and the model are combined together, and run by SPIN in the verification mode. The execution will give a counterexample path and write it into a trail file. By performing a simulation run based on that file, we can reproduce the error trail which shows why the channel-redundancy occurs. That is, at first flag is assigned false, then the processes par1 is executed where the interaction can not be done because of the unsatisfied if condition, finally the process par2 is executed. Here, K^b_a is true according to the initial condition, but U^b_a would never become true, so the checked LTL formula is violated.

4 Generating Channel-passings

Now we want to extend the application of model checking to the problems which are usually solved by algorithms in constructive ways. The example presented here is to generate channel-passings, i.e, to find a path (i.e., a chain of interactions) in a channel-absence choreography so that if channel passings are added for each of the interactions along the path, the channel-absence problem is fixed.

A model checker always tries to find a counterexample from the model and the property, which is a feasible execution of the model and shows the violation of the property. If is can not find such counterexample by an exhaustive search in the whole state space, the property is proved. Although what we want to do is something constructive, the work must be done based on the mechanism of the model checker.

It is clear that if a path found by the model checker can be used to repair the channel-absence, the path must be a counterexample for some property. Then, the prop-
bool $k_a^b = true$, $u_a^b$, flag = false;
bool par1_fin, par2_fin;
active proctype main() {
  atomic { run par1(); run par2() }
  par1_fin & par2_fin;
}
proctype par1() {
  par1_fin = false;
  if
    :: flag $\rightarrow \{ k_a^b \rightarrow u_a^b = true\}$
  :: else $\rightarrow$ skip
  fi;
  par1_fin = true;
}
proctype par2() {
  par2_fin = false;
  flag = true;
  par2_fin = true;
}

List 4: Example for Channel-redundancy

...
bool \( P^n_a, P^n_b, K^n_a, K^n_b \);
active proctype main() {
  if :: true →
    atomic \{ \( K^n_a = \text{true}; P^n_a = \text{true} \) \}
  \( P^n_c = (P^n_a || P^n_b) \)
  :: true → \( P^n_c = \text{true} \)
fi;
  assert(!K^n_a \rightarrow !P^n_c : \text{true})
}

List 5: Example for Generating Channel-passings

bool \( P^n_b, P^n_c \);
active proctype main() {
  if :: true → \( P^n_b = \text{true} \);
  :: true → \( P^n_c = \text{true} \);
fi;
  assert(P^n_c);
}

List 6: Example

We still use a pair of pre- and post-conditions to represent the interactions. For \( r_i \rightarrow r_j \), the pre-condition is \( \text{true} \), the post-condition is \( P^n_i \rightarrow P^n_j \). These conditions can be represented by PROMELA statement \( P^n_i = (P^n_j || P^n_i) \), where || is logical or operator. For \( r_i \xrightarrow{ch_a} r_j \), the pre-condition is \( \text{true} \), the post-condition is \( (K^n_i \rightarrow K^n_j) \land (P^n_i \rightarrow P^n_j) \). They are represented as PROMELA statement \( \text{atomic} \{ K^n_i = (K^n_j || K^n_i); P^n_j = (P^n_j || P^n_i) \} \). For \( r_i \xrightarrow{ch_k} r_j \), where \( ch_k \neq ch_a \), its pre- and post-conditions are all \( \text{true} \), and the corresponding statement is \text{skip}.

To show the approach, we use an example as follows.

\[
((r_a \xrightarrow{ch_a} r_b; r_b \rightarrow r_c) \sqcap r_a \rightarrow r_c); r_c \rightarrow r_a
\]

Suppose initially, \( r_a \) knows \( ch_b \) and \( ch_c \), and \( r_b \) knows \( ch_c \), but \( r_c \) does not know \( ch_a \). It is easy to see that channel-absence occurs at \( r_c \rightarrow r_a \). The corresponding PROMELA process is given in List 5.

Having a run in verification mode, SPIN gives a counterexample path, along which we can add the passing of the absent channel. A modified choreography may be:

\[
((r_a \xrightarrow{ch_a} r_b; r_b \xrightarrow{ch_b} r_c) \sqcap r_a \rightarrow r_c); r_c \rightarrow r_a
\]

In one verification run, we can fix only one execution path for the channel-absence, and can not assure that there is no other absence in the choreography. In order to remove
all the possible problems, we should repeat this procedure until no more counterexample is found. After that, we try the channel-absence-checking again to see if there is still absence of the channel in the choreography.

In fact, for some choreography, it is impossible to completely solve the channel-absence problem by just only generating channel-passings without changing the existing control structure. For example, consider choreography \((r_a \rightarrow r_b \sqcap r_a \rightarrow r_c); r_c \rightarrow r_a\), where \(r_c\) does not know \(ch_a\) initially. It is easily seen that a channel-absence would occur in \(r_c \rightarrow r_a\), and this cannot be solved by only adding channel-passings to the choreography. We call this choreography *intrinsically ill-organized*, which can be identified by a method similar with the above. There are two modifications for the method above: first, the boolean variable \(K^i_a\) are omitted; second, the assertion is simply written as \(P^a\), that is, for every possible execution path, \(r_c\) might know \(ch_a\) eventually by adding channel-passings along it. If a counterexample is found, it means that there is some path along which we could not repair the channel-absence—that just proves that the checked choreography is intrinsically ill-organized. The translation of above example is shown in **List 6**. **SP**\(\text{IN}\) will give us a counterexample showing there is an execution path which cannot be repaired by adding channel-passings.

5 \textbf{Related work}


There are some literatures about verification on choreography. Corradini \textit{et al.} [6] proposed a technique which maps WS-CDL choreography to **PROMELA** processes and model check them. Their method is to translate each participant in a choreography into a separate **PROMELA** process, apparently different from ours. We develop a different method based on the following two reasons. First, the increased number of the **PROMELA** processes would noticeably increase the state space of the model and then make heavy burden on the verification. Second, perhaps more important, their method actually ignore the global view of WS-CDL and instead build a trial implementation by simply projecting the global behavior description into all participant roles. The conformance between the global description and their implementation is in doubt. For example, a nondeterministic choice structure in a choreography, say \(A \sqcap B\), would lead all participants to make the choice uniformly, no matter all go into \(A\) or into \(B\); whereas their projection method would no longer assure this kind of uniformity, because each participants would make the choice itself, thus someone may take \(A\) and the others may take \(B\). Díaz \textit{et al.} [8] demonstrated how to translate the descriptions written in WSCI/WS-CDL into timed automata, and then, use the **UPPAAL** tool to simulate and
verify the system’s behavior. The method they used is to translate each participant into a separate process, similar with the previous one. In addition, Fredlund [9] implemented a prototype tool which can simulate and debug simple WS-CDL descriptions.

6 Conclusion and Future Work

A choreography takes a global viewpoint on a set of collaborative services for a common business goal. For supporting dynamic configuration and composition, channel passing is a basic feature in languages for describing choreographies. A typical language falling into this catalog is WS-CDL developed by a W3C group. Some experience [4] shows that the design of channel passing is an error-prone work, thus formal verification techniques are very valuable.

In this paper we study how to verify properties about channel passing in a choreography with model checking. To develop an efficient method for the verification, suitable abstraction is indispensable. Many properties related to channel passing in choreography need to be verified, and the abstraction functions depend on the properties that we want to prove. In this work, we propose to map each basic interaction into a pair of pre- and post-conditions which use the state variables introduced into the specifications of the properties. Based on this abstraction, we can construct a model for any choreography in a compositional way. Associated with the relevant temporal logical formula, the model can be checked by model checkers, e.g., SPIN.

Many temporal properties about channel passing of choreography can be verified easily in this manner. The examples we present here include the channel-absence, channel-redundancy and channel-used-once problems. Furthermore, we present a specific example, searching a feasible path to fix the channel-absence automatically. This shows that model checking technique can be used for some constructive tasks as well as for verification.

References


