Service-Oriented Architectures: from Design to Production exploiting Workflow Patterns

Maurizio Gabbrielli[a], Saverio Giallorenzo[a], Fabrizio Montesi[b]

[a] Dipartimento di Informatica – Università di Bologna / INRIA

[b] Southern Denmark University

KEYWORD

Service-Oriented Architecture
Workflow Patterns
Coloured Petri Nets

ABSTRACT

In Service-Oriented Architectures (SOA) services are composed by coordinating their communications into a flow of interactions. Coloured Petri nets (CPN) offer a formal yet easy tool for modelling abstract SOAs. Still, mapping abstract SOAs into executable ones requires a non-trivial and time-costly analysis. Here, we propose a methodology that maps CPN-modelled SOAs into executable Jolie SOAs (our target language). To this end, we employ a collection of recurring control-flow patterns, called Workflow Patterns, as composable blocks of the translation. Following our methodology, we discuss how the Workflow Patterns we consider are translated in Jolie. Finally, we validate our methodology with a realistic use case. As additional result of our research, we could pragmatically assess the expressiveness of Jolie with relation to the considered Workflow Patterns.

1 Introduction

Service-Oriented Computing (SOC) is a design methodology focused on the realisation of systems by composing autonomous entities called *services*. In a Service-Oriented Architecture [ERL, T, 2005] (SOA), services are composed by coordinating their communications into a flow of interactions. Several tools have been presented [OMG, 2009; OASIS, 2012; MAYER, P *et al.*, 2009] to assist the process of SOA design, each focusing on a particular aspect of the system, e.g., the architectural composition or the interaction among components. Coloured Petri nets [JENSEN, K et al., 2007] (CPNs) are a formal yet intuitive graphical tool, largely employed in business process modelling [VAN DER AALST W *et al.*, 2003] and suitable for SOA specification.

Although in CPN models interactions are easy to understand, it is unclear which components form the system, which implement the described logic or whether it be spread among the components or centralised.

Therefore the aim of this work is to provide a methodology that allows the translation of CPN-modelled SOAs into executable ones.

The *Workflow Patterns Initiative* (WPI) studied and collected a comprehensive set of recurring patterns of process-aware information systems, dubbed *Workflow Patterns* [VAN DER AALST W *et al.*, 2003] (WP).

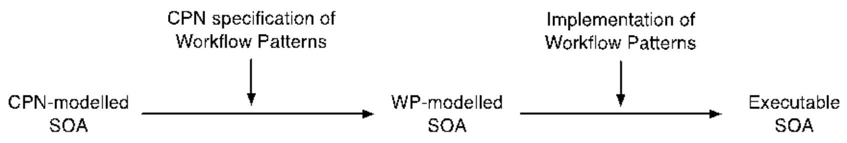

CPN specification of Workflow Patterns
Implementation of Workflow Patterns
CPN-modelled SOA
WP-modelled SOA
Executable SOA

Figure 1: The scheme of translation from abstract to executable SOAs.

In particular we remark the exhaustive set of patterns of interaction, dubbed *Control-Flow Workflow Patterns* [RUSSEL, N *et al*., 2006] (here referred as Workflow Patterns for simplicity), modelled as CPNs. Since CPNs are composable, our idea, depicted in Fig. 1, is that an SOA, modelled as a CPN, can be described in terms of the Workflow Patterns it is made of. Once the SOA is defined by a composition of WPs, the developer only has to refer to the implementation of each WP to build the whole system.

To realise our proposal, we provide the implementation of a substantial set of WPs, discussed in Section 4. Such implementation is not immediate since WPs are abstract specifications and it is unclear how they map into executable code for service coordination. Moreover, although the same WP applies to different subnets of interactions, its implementation may differ sensibly depending on whether its logic is *centralised* in a single component or *distributed* among several ones.

Centralised and distributed approaches suit different contexts. E.g., if a vendor wants to monitor its application he might prefer a single point of control to track the whole system. On the other hand, some scenarios strictly require a distributed approach, e.g., an interaction that comprehends different parties. In Section 5 we consider a realistic use case that combines the two approaches.

We translate both the centralised and distributed versions of WPs as composable and executable SOAs. In order to provide a consistent translation we also define a *procedure* in Section 3.

Such procedure might directly map a CPN-modelled SOA to an executable one, thus skipping the said in-between translation to a WP-modelled SOA. However, the behaviour of some WPs needs ad-hoc solutions (see Table 1), not directly mapped by the presented procedure. Thus, although providing an automatic procedure is an interesting challenge, in this work we focus on the practical implications of enabling developers translate CPN-modelled SOAs into executable ones by referring to our collection of Workflow Patterns. Our procedure applies to any service-oriented language, e.g., BPEL [OASIS, 2006] but we choose to implement WPs in Jolie [JOLIE 2014; MONTESI, F *et al*., 2014] for two main reasons. First Jolie supports several communication and serialisation protocols, thus the same implementation applies to different application domains. Second Jolie is based on a formal process calculus [GUIDI, C et al., 2006], which we plan to use to prove relevant correctness properties of translated SOAs.

2 Background

2.1 Coloured Petri Nets and Workflow Patterns

In this section, we provide a brief introduction to the basic terminology and notation of Coloured Petri Nets (CPN), which are used as specification language for control-flow WPs. CPNs are a modelling language that combines elements inherited from *Petri Nets* [REISIG, W, 1985] (PNs) and capabilities of high-level programming languages, which allow the construction of parameterised models. The main elements of CPNs are the following:

- *places* are locations where tokens reside. A place can have a cardinality associated to it, expressing the maximum amount of tokens that it can contain. Places represent the *state* of the system according to a specific marking, which is a distribution of tokens among the places of a net at a given time. Places are depicted as empty circles;
- *tokens* are used to mark when a certain state, i.e., a place, holds. In CPNs, tokens have a data value attached to them, namely, a token colour. Tokens are represented as filled circles and can only appear inside places;
- *transitions* are used to represent the *dynamic behaviour* of the system and are depicted as boxes;
- *arcs* indicate the relations connecting transitions and places and specify the *flow* of tokens through the net. Graphically, arcs are represented as directed arrows. Each arc has an *expression* associated with it that

defines its binding policies and the quantity of tokens involved. Policies are expressed on values of a specific data type, i.e., a specific token colour.

Defined $\bullet t$ as the set of input places of a transition *t* and $t\bullet$ as the set of its output places, $\bullet t$ may fire if: *i*) all places in $\bullet t$ contain the amount of coloured tokens that satisfy the expression associated with each arcs entering in *t* and *ii*) all places in $t\bullet$ can contain the specific amount of coloured tokens yielded by *t*. When *t* fires, it removes tokens from places in $\bullet t$ and yields tokens in $t\bullet$. The number of tokens is described by the expressions on arcs. The control-flow WPs we refer in this work are taken from [RUSSEL, N *et al*., 2006]. We also adopt the definitions the assumptions made in [RUSSEL, N *et al*., 2006] on CPN models. In particular, tokens that indicate control-flow are typed CID, input places are denoted by `i1,…,in`, output places by `o1,…,on`, internal places by `p1,…,pn`, and transitions by `A,…,Z`. Furthermore, we assume that, unless differently indicated, the CPN that models a pattern is *safe*, i.e., each place in the model can only contain at most one token.

Communication can happen on different media (e.g., TCP/IP, Bluetooth, Java RMI, Unix local sockets, etc.) and with various data protocols (e.g., SODEP, SOAP, HTTP, JSON-RPC, XML-RCP, and their equivalent over SSL).
Here, we focus on the behavioural aspect, i.e., the instructions to be performed and the input/output communications of services.
Jolie combines message passing within an imperative programming style. It provides scopes, indicated by curly brackets `{…}`, to represent procedures. Procedures can be labelled with the keyword `define`. The name of a procedure is unique within a service and is used to execute its code, e.g., the `main` procedure is the entry point of execution of every service. Conditions, loops, and sequence composition operator `;` are standard. The *parallel* composition operator `|` states that both left and right operands execute concurrently. The parallel operator has always priority on the sequence. Scopes ease the definition of precedence between different code blocks (as shown at Lines 2-5 in Listing 1).
Jolie provides also an input-guarded choice with the following syntax: $[\eta_1]\{B_1\}\dots[\eta_n]\{B_n\}$, where $\eta_i, i \in \{1,\dots,n\}$, is an input statement and B_i is the branch-related behaviour.

Listing 1: An example of composition and communication between services

```
1 //service A
2 {
3  op1@B( a )
4  | op2@B( b )
5 };
6 op3@B( e )( h )
```

```
1 //service B
2 {
3  op1( c )
4  | op2( d )
5 };
6 op3( f )( g ){
7  g = "Hello , world"
8 }
```

2.2 Composing services is Jolie

We now present the basic concepts needed for understanding the *behaviour* of services written in Jolie. For a comprehensive presentation of the Jolie language refer to [JOLIE 2014; MONTESI, F *et al*., 2014].

A Jolie service comprehends two parts. One describes the behaviour of the service. The other defines its *deployment*. The independence between behaviour and deployment in Jolie allows to seamlessly integrate heterogeneous networks made by Jolie and non-Jolie services.

When a message on η_i is received, all other branches are deactivated and η_i is executed. Afterwards, B_i is executed. A static check enforces all the input choices to have different operations to avoid ambiguity.

Jolie supports two kinds of message-passing operations, namely *One-Ways* (OWs) and *Request-Responses* (RRs). On the sender's side, OWs send a message and immediately pass the thread of control to the subsequent activity in the process. RRs send a request and keep the thread of control until they receive a response.

On the receiver's side, OWs receive a message and store it into a defined variable. RRs receive a message into a variable, wait for the execution of the code in its body, and finally send the content of the second variable as response.
Listing 1 exemplifies an SOA made of two services `A` and `B`. `A` sends in parallel the content of variables `a` and `b` through OW operations `op1` and `op2`, respectively, at (`@`) `B` (Lines 3-4, Service `A`). When `B` receives a message on one of the corresponding OW operations, it stores the content of the message in the corresponding variable (Lines 3-4, Service `B`). After the completion of scopes at Lines 2-5, `A` proceeds with the subsequent RR operation `op3`. `op3` sends the content of variable `e` and stores its response in `h`. The scope linked to `op3`, in Lines 6-8 of service `B`, is the procedure executed before sending the response to `A`. In the example, the procedure assigns a string to `g`.

In Jolie, variables are dynamically typed while OWs and RRs statically define the type of the message they carry. The language provides the `interface` construct to declare a set of supported operations and the type of their messages. Interfaces are specified in the deployment part of a Jolie service. Whenever a message is sent or received, its type is checked against its specification and a fault is raised in case of mistyping.

The `execution` statement defines how the behaviour of a Jolie service shall run. Allowed values are: `single` (default, if the execution statement is omitted), `concurrent`, and `sequential`. Except for the `single` execution modality, a new instance of the service starts whenever its first input operation is invoked. Concurrent instances run immediately after their invocation. Sequential instances are queued and run only when all previous instances terminated.

3 From Coloured Petri nets to Jolie SOAs

In this section we show how CPN models of Workflow Patterns can be translated into SOAs implemented in Jolie. Our technique for translating CPNs into SOAs is based on five principles:

i. transitions are services;
ii. places are message passing operations (i.e., communications);
iii. communications carry typed messages, as coloured tokens do;
iv. arcs are properties on communications: they express the type of carried messages and the conditions that fire the communication;
v. a CPN models an SOA composed by several services running in parallel.

Following these principles, CPN models of WPs are translated into Jolie SOAs as follows. We map input `i1,…,in` places, internal `p1,…,pn` places, and output places `o1,…,on` into One-Way (OW) operations (principle *ii*). In case other internal operations are needed, we use the notation `pi1,…,pin`, where `i` identifies a set of related operations. When it is compatible with the behaviour of the pattern, we coalesce two round-trip OW operations between two services into one Request-Response (RR) for brevity.

Since in Jolie output operations define the service they communicate to, we map output places into OWs on default output deployment locations `DefaultOutput1,…, DefaultOutputn`. This allows to compose patterns on the basis of the definition of their deployment locations.

As stated in principle (*v*), services in implemented SOAs run in parallel. We set the default execution of each service of the system as `sequential` to comply with the *safety* property defined in section 2.1. We also omit the declaration of scope `main` if the realisation of a pattern is independent from its position in the execution of a service.
In order to model WPs as SOAs, we relax the definition of *instance* given by the *workflow terminology* in [WMC, 1999]. Here, an *SOA instance* is a composition of instances of services that are related by messages carrying specific *session identifiers* or SIDs. Each SID

identifies a unique execution of an SOA and we employ correlation sets to identify and manage different sessions (see [MONTESI, F *et al*. 2011] for more details). However, we omit the definition of correlation sets in our implementations, as they are not necessary for the definition of the behaviour of services.

SOAs can be realised by following a centralised or a distributed approach, usually referred to as, respectively, *orchestration* and *choreography*. In the first case, an *orchestrator*, or *master service*, encodes the whole behaviour of the SOA in terms of interactions among the different services participating in the SOA. BPEL [OASIS, 2006] is the most renowned technology for this approach. In the second case, a *choreography* specifies the global behaviour of an SOA. This description is abstract, has no centralisation point, and defines the interaction of the services participating in the SOA. Choreographic languages such as WS-CDL [WS-CDL, 2004] have been specifically designed for this purpose. Recent works [CARBONE, M *et al*., 2013, LANESE, I *et al*., 2008] introduced automatic projection techniques that allow to obtain executable services of an SOA from a choreographic specification. In our work, we call choreography a set of coordinated services that implement the global behaviour in such a distributed way.

For each WP we provide both a centralised and a distributed implementation. In the centralised implementation, the master service realises the behaviour of a pattern and is the only service that receives and sends messages outside the SOA. In the distributed approach we maintain a direct relation between transitions and services, thus we impose no restriction on the scope of external input and output operations. The implementation of each WP under both methodologies allows us to achieve three results: first, designers can determine the components that enact a specific pattern; second, developers have a standardised reference for the implementation of patterns; third, from the differences in the two approach emerge interesting aspects concerning the expressive power of the implementation language (Jolie in our case), as we discuss in the Conclusions (Section 6).

Example. Let us consider a graphical example of a translation from a CPN model to its centralised and distributed implementations. We label A the CPN in box A of Fig. 2. A reads: when a token reaches place `i1`, transition `A` can fire. `A` yields a token in place `p1` if condition `cond1` holds, else it yields a token in `p2`.

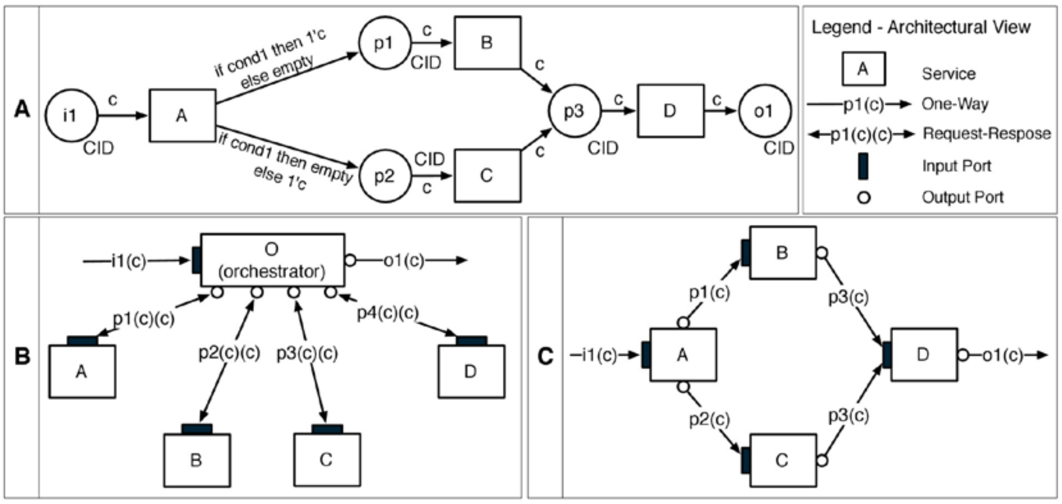


Figure 2 (A) CPN model and its centralised (B) and distributed (C) implementations.

Listing 2: Centralised (right) and distributed (left) implementations of CPN A.

```
//orchestrator
i1( c );
p1@A( c )( cond1 );
if( cond1 ){
 p2@B( c )( c )
} else {
 p3@C( c )( c )
};
p4@D( c )( c );
o1@DefaultOutput1( c )
```

```
//service A
i1( c );
if( cond1 ){ p1@B( c ) }
else { p2@C( c ) }
// service B
p1( c ); p3@D( c )
// service C
p2( c ); p3@D( c )
// service D
p3( c ); o1@DefaultOutput1( c )
```

Transition B or C fires concordantly, yielding a token in place p3. Finally, transition D fires and yields a token in o1. The SOA in box B of Fig. 2 shows the centralised realization of A. The orchestrator implements the behaviour of the pattern by sending round-trip messages by means of RRs that invoke specific operations on services, waiting for their responses. Diagram B reads: the `orchestrator` receives a message on operation `i1`. It evaluates condition `cond1` internally (not shown by the diagram) to decide whether to invoke service B or C on operation `p2` or `p3`, respectively. Then, it invokes operation `p4` on D that returns its output. Finally, it sends the output of the system on `o1`. The distributed approach maintains a direct relation between transitions and services as shown in box C of Figure 2. Services pass the thread of control using OW operations. Service A receives a message on operation `i1`. A evaluates condition `cond1` internally and invokes service B or C, respectively, on operation `p1` or `p2`. The invoked service sends a message to service D that sends its output on `o1`. The operations in boxes B and C show the type of the message they carry between round brackets. The type is the same as the one of c in the CPN. Listing 2 reports the corresponding code of, respectively, the orchestrator of the centralised version (left) and of the services in the distributed one (right).

4 Workflow Patterns in Jolie

In this section, we report the full discussion on the support and the implementations of *basic* and *advanced branching and synchronisation control-flow* Workflow Patterns in Jolie.

In the listings of the considered Workflow Patterns we omit the code of trivial services for a cleaner presentation.

6.1 Basic Control-Flow Patterns

Sequence

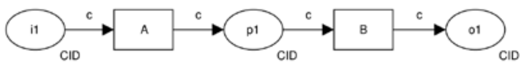


Figure 3: Sequence pattern diagram

Definition
The *Sequence* describes an activity in a workflow process that is enabled after the completion of a preceding activity in the same process.

Implementation
The *Sequence* pattern is directly supported by the sequence operator `;` presented in Section 2.2. The centralised version coalesces couples of round-trip OWs into RRs. In the distributed one each service passes the thread of control to the subsequent service through an OW message.

Listing 3: Sequence – Centralised

```
i1( c );
i1@A( c)(c);
p1@B( c)(c);
o1@DefaultOutput1( c )
```

Listing 4: Sequence – Distributed

```
// service A
{
 i1( c ); p1@B( c )
}
// service B
{
 p1(c); o1@DefaultOutput1( c )
}
```

Parallel Split

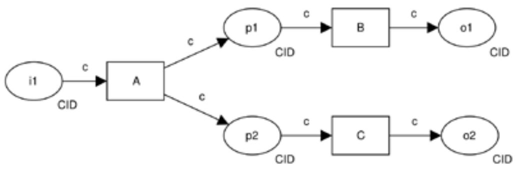


Figure 4: Parallel Split pattern diagram

Definition
The *Parallel Split* represents the divergence of a branch into two or more parallel branches each of which executes concurrently.
Implementation
The parallel operator |, presented in Section 2.2, provides a direct support to the *Parallel Split* pattern as it splits the thread of control between two branches. Noticeably, the centralised version of this pattern makes use of scopes {…} to manage the parallel execution of the two branches emanating from transition A.

Listing 5: Parallel Split – Centralised

```
i1( c );
{
 p1@A( c )( c1 );
 o1@DefaultOutput1( c1 )
}
|
{
 p2@B( c )( c2 );
 o2@DefaultOutput2( c2 )
}
```

Listing 6: Parallel Split – Distributed

```
// service A
{
 i1( c );
 { p1@B( c ) | p2@C( c ) }
}
// service B
{
 p1( c );
 o1@DefaultOutput1( c )
}
// service C
{
 p2( c );
 o2@DefaultOutput2( c )
}
```

Synchronisation

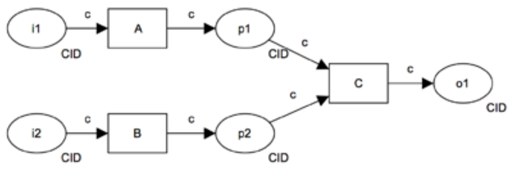


Figure 5: Synchronisation pattern diagram

Definition
The *Synchronisation* represents the convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. As context condition, only one incoming signal can reach each incoming branch. Once the behaviour of the pattern has been reset, no other signal reaches the input branches.

Implementation
The *Synchronisation* complements the *Parallel Split*. The behaviour of the pattern is directly supported by the semantic of scopes {…} presented in Section 2.2. In Jolie, the thread of control of a scope passes to its parent only when its execution terminates. *Synchronisation* derives from a composition of scopes with *Parallel Split*.

Listing 7: Synchronisation – Centralised

```
{
 {
  i1( c1 );
  p1@A( c1 )( c.c1 )
 }
 |
 {
  i2( c2 );
  p2@B( c2 )( c.c2 )
 }
};
p3@C( c )( c );
o1@DefaultOutput( c )
```

Listing 8: Synchronisation – Distributed

```
// service C
{
 {
  p1( c1 )
  |
  p2( c2 )
 };
 o1@DefaultOutput1( c )
}
// service A
{ i1( c ); p1@C( c ) }
// service B
{ i2( c ); p2@C( c ) }
```

In the centralised implementation we used subnodes of variable `c` to store the content of data belonging to different branches. In Jolie variables are organised as data trees. Therefore a variable is a *path* for traversing the data tree. State traversing is obtained through "`.`", the dot operator.

Exclusive Choice

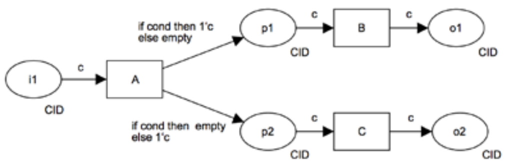


Figure 6: Exclusive Choice pattern diagram

Definition
The *Exclusive Choice* represents the divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch.

Implementation
Jolie directly supports the Exclusive Choice pattern in two ways, whether the desired mechanism of selection is deterministic or non-deterministic. The conditional statement `if…else` performs a deterministic internal choice. The *input choice* rule implements a non-deterministic choice. The condition evaluated by the *input choice* is the invocation of one of the branched operations, which may derive either from an internal choice of the invoker or from a race between several invokers. Both solutions apply to centralised and distributed approaches. For brevity, we show the internal choice in a centralised architecture and the non-deterministic choice in choreography. In Listing 9, the orchestrator evaluates the condition `cond` and chooses whether to proceed on branch `B` or `C`. In Listing 10, we insert an additional service `P` that service `A` invokes on operations `p1` or `p2` after the evaluation of condition `cond`.

Listing 9: Exclusive Choice – internal choice

```
i1( c );
p1@A( c )( cond );
if ( cond ){
 p2@B( c )( c );
 o1@DefaultOutput1( c )
} else {
 p3@C( c )( c );
 o2@DefaultOutput2( c )
}
```

Listing 10: Exclusive Choice – external choice

```
// service A
{
 i1( c );
 if ( cond ){ p1@P( c ) }
 else { p2@P( c ) }
}
// service P
{
 [ p1( c ) ]{ p3@B( c ) }
 [ p2( c ) ]{ p4@C( c ) }
}
```

Simple Merge

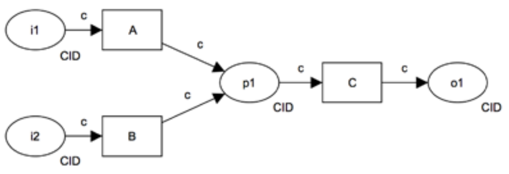


Figure 7: Simple merge pattern diagram

Definition
The *Simple Merge* represents the convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch. There is one context condition associated with the pattern: the place at which the merge occurs,

i.e., place `p1`, is safe thus it cannot contain more than one token.

Implementation
Jolie provides a direct support for this pattern as it can be obtained from a composition of primitive constructs provided by the language and directly supported patterns.

We label *s* the subnet in Fig. 7 composed by the transitions `A`, `B`, and `C` and place `p1`. *s* defines an *OR-join* since it allows the activation of `C` each time `A` or `B` yields a token. Additionally, `p1` is *safe*, which makes *s* become an exclusive *OR-join* (XOR-join). The *OR-join* component derives from a *Sequence* of each incoming branch followed by an OW operation towards the merging service `C`. This holds for both orchestration and choreography.

The *exclusive* property forces each incoming operation to execute sequentially and its implementation differs between the two approaches. The centralised implementation composes the branches corresponding to services `A` and `B` in *Synchronisation*. When each of them returns its response, the orchestrator invokes `p1` on service `C`. The synchronized scope, provided by Jolie, guarantees mutual exclusion among branches that access the same resource (*token*). In the distributed implementation, the `sequential` execution modality queues multiple firings of service `C` and executes them sequentially, guaranteeing mutual exclusion. `C` has no dependency on the number of branches to be merged.

Listing 11: Simple Merge – Centralised

```
{
 i1( c1 );
 i1@A( c1 )( c1 );
 synchronized ( token ){
  p1@C( c1 )( c1 );
  o1@DefaultOutput1( c1 )
 }
}
|
{
 i2( c2 );
 i2@B( c2 )( c2 );
 synchronized ( token ){
  p1@C( c2 )( c2 );
  o1@DefaultOutput1( c2 )
 }
}
```

Listing 12: Simple Merge – Distributed

```
 // service A
 { i1( c ); p1@C( c ) }
 // service B
 { i2( c ); p1@C( c ) }
// service C
execution { sequential }
{ p1( c ); o1@DefaultOutput1( c ) }
```

6.2 Advanced Branching Patterns

Multi-Choice

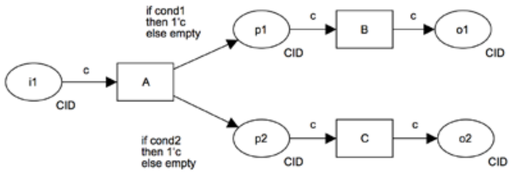


Figure 8: Multi-Choice pattern diagram

Definition
The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

Implementation
Multi-Choice is supported directly and its implementation de-rives from *Exclusive Choices* composed with a *Parallel Split*. This implementation holds for both centralised and distributed approaches.

Listing 13: Mutli-Choice – Centralised

```
 i1( c );
 {
  p@A( c )( cond1 );
  if( cond1 ){
   p1@B( c1 )( c1 );
   o1@DefaultOutput1( c1 )
  }
 }
 |
{
  p@A( c )( cond2 );
  if( cond2 ){
   p2@C( c2 )( c2 );
   o2@DefaultOutput2( c2 )
  }
}
```

Listing 14: Multi-Choice – Distributed

```
// service A
i1( c );
{
 if( cond1 ){
  p1@B( c )
 }
}
|
{
 if( cond2 ){
  p2@C( c )
 }
}
```

Thread Split

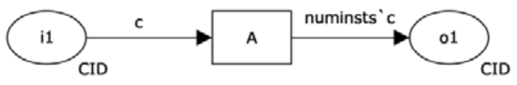


Figure 9: Thread Split pattern diagram

Definition

At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance. There is a context condition for this pattern: the number of splitting threads is known at design-time.

Implementation

Jolie directly supports this pattern. Since the implementations for this pattern are the same for both centralised and distributed approaches, we provide the centralised only. *Thread Split* can be implemented in three ways: iteratively, with parallel recursion, and with the `spawn` construct.

Listing 15: Thread Split – iterative

```
i1( c );
p1@A( c )( c );
for ( i=0, i< numinst , i++ ){
 o1@DefaultPort1( c )
}
```

Listing 15 shows the iterative solution that uses the `for` statement. OWs in Jolie are asynchronous and can start parallel executions of other processes. However, OWs pass the thread of control only after the reception of an acknowledgement. Hence, this solution achieves a “not direct” rating. OWs composed inside an iterative scope prevents a real parallel firing of threads, as the next thread is started only after the acknowledgement of reception of the preceding one.

Listing 16: Thread Split – recursive

```
define thread_split
{
 {
  if ( i < numinst ){
   i++;
   {
    o1@DefaultOutput1( c )
    | thread_split
   }
  }
 }
}

main
{
 i1( c );
 p1@A( c )( c );
 i=0;
 thread_split
}
```

The recursive method, shown in Listing 16, consists of a recursive composition of *Parallel Split*s. This solution offers a direct support for this pattern. At each execution, the branching procedure `thread_split` creates a new invocation and invokes itself in parallel, eventually creating `numinst` parallel branches of the same procedure.

Listing 17: Thread Split – `spawn`

```
i1( c );
p1@A( c )( c );
spawn ( i over numinst )
{
 o1@DefaultOutput1( c )
}
```

The solution that uses the `spawn` [MONTESI, F, 2010] primitive offers a direct support too. Shown in Listing 17, the `spawn` statement creates a parallel composition of a number of processes equal to the integer evaluation of the given expression.

6.3 Advanced Synchronisation Patterns

Generalised AND-Join

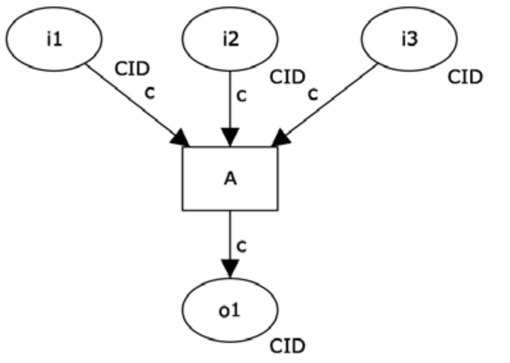


Figure 10: Generalised AND-Join pattern diagram

Definition
The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firing. Unlike the *Synchronisation* pattern, the *Generalised AND-Join* supports non-safe contexts, i.e., one or more incoming branches may receive multiple triggers in the same process instance. When the pattern executes, it takes one token from each input place `i1,…,in`, ignoring additional tokens that are left in place.

Implementation
We identify two implementations for the *Generalised AND-Join*, although they respectively achieve a "not direct" and a "not supported" rating for this pattern. The first solution composes input operations within a *Synchronisation* scope and it is valid only if we assume an *order* among tuples of received messages. We say that, two tuples of incoming messages $s = \{m_1, \ldots, m_n\}$ and $s' = \{m'_1, \ldots, m'_n\}$ are *ordered* on the same session *k*, if, in case a message of *s* reaches the service first, no message of s' shall reach the service before all remaining messages of *s* have reached the service, and vice versa for s'. In Jolie, the order of consumed messages must be coherent with the specification of execution, or the system ends in a deadlock state [MONTESI, F, *et al.*, 2011]. Listing 18 shows the centralised implementation of this solution, which holds also for the distributed version. The second implementation, in Listing 19, fully supports the requirements of the pattern and holds for both centralised and distributed approaches. However, it achieves a "not supported" rating due to the necessity of a dedicated queuing mechanism. In order to manage multiple unordered triggers on the same session, we employ *input choice* and queues. Each time a new invocation arrives it starts a new instance of the joining service. The subsequent procedure `(queueOp_i1,…,queueOp_i3)` stores the carried message into an ad-hoc (FIFO) queue. Then, procedure `check_and_send` checks if each queue has at least one element. If enough messages arrived, the procedure pulls out the involved elements — one per queue — and triggers the finalising behaviour. We purposely omit the definitions of any of the procedures. Queuing functionalities can be implemented either within the joining service or relying on auxiliary services.

Listing 18: Generalised AND-Join – order assumption

```
{
 i1( c.c1 )
 |
 i2( c.c2 )
 |
 i3( c.c3 )
};
p@A( c )( c );
o1@DefaultOutput1( c );
```

Listing 19: Generalised AND-Join

```
main
{
 [ i1( c ) ]{
  queueOp_i1 ;
  check_and_send
 }

 [ i2( c ) ]{
  queueOp_i2 ;
  check_and_send
 }

 [ i3( c ) ]{
  queueOp_i3 ;
  check_and_send
  }
}
```

Multi-Merge

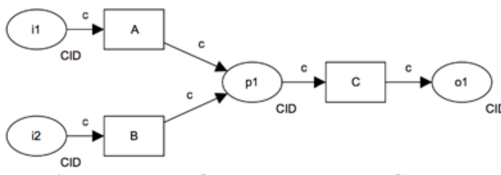


Figure 11: Multi-Merge pattern diagram

Definition
The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch. The distinction between this pattern and the *Simple Merge* is that it is possible for more than one incoming branch to be active simultaneously and there is no necessity for place `p1` to be safe.

Implementation
Jolie has a direct support for this pattern as the centralised and distributed implementations provided for *Simple Merge* require minimal changes to realise the behaviour of this pattern. In orchestration, we remove the mutually exclusive `synchronized` scopes (Lines 4 and 7 and 13 and 16 of Listing 11). In choreography, service `C` switches its execution from `sequential` to `concurrent` (Line 6 of Listing 12).

Thread Merge

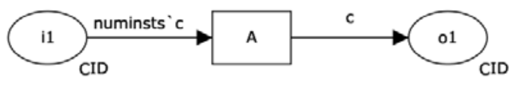


Figure 12: Thread Merge pattern diagram

Definition
At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution. There are two context considerations for this pattern. (*a*) The number of threads to merge must be known at design-time. (*b*) Only execution threads for the same process instance can be merged. If the pattern merges independently executing threads arisen from some form of activity spawning, it shall specifically identify the threads to be coalesced.

Implementation
We identify two implementations that offer direct support to this pattern. One is iterative whilst the other relies on multiple instances. Here, we provide the implementations realised in a centralised architecture, yet they remain the same also for choreography. Both solutions make use of the knowledge at design-time on the number of threads to merge (*a*). The employment of correlation sets [MONTESI, F, *et al.*, 2011] prevents non-correlated messages to be routed towards the wrong instance of the merging service, identifying the threads to coalesce (*b*).

Listing 20: Thread Merge – iterative

```
for ( i=0, i< numinst , i++ ){
 i1( c[ i ] )
};
p1@A( c )( c );
o1@DefaultOutput1( c )
```

Listing 20 shows the iterative solution, realised by means of the `for` statement. The service receives each input message on operation `i1`. For each invocation, it stores the data of the incoming message into an array. After the `numinst`-th invocation, it sends its output.

Listing 21: Thread Merge – multi-instance

```
execution { sequential }
init {
  c -> global.c;
  i -> global.i
}
main  {
 i1( c[ i++ ] );
 if( i == numinst ){
  p@A( c )( c );
  o1@DefaultOutput1( c )
 }
}
```

Similarly, the multi-instance implementation, in Listing 21, uses the `sequential` execution to receive one message per instance, storing the message in structure c and counting their number with variable `i`. In the init scope (executed before `main`) both `c` and `i` *alias* a *global variable* [MONTESI, F *et al.*, 2014] to preserve the global status the system over multiple instances.

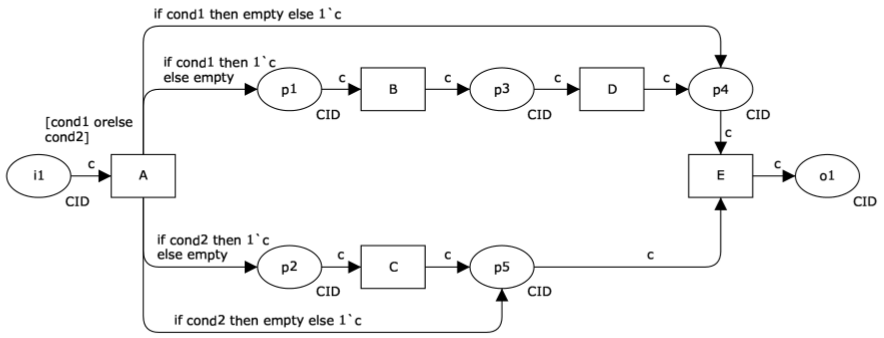

if cond1 then empty else 1`c
if cond1 then 1`c else empty
[cond1 orelse cond2]
i1
CID
c
A
p1
B
p3
D
p4
E
o1
if cond2 then 1`c else empty
p2
C
p5
if cond2 then empty else 1`c

Figure 13: Structured Synchronising Merge pattern diagram

Structured Synchronising Merge

Definition
The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The *Structured Synchronising Merge* occurs in a structured context, i.e., there must be a single *Multi-Choice* construct earlier in the process model which the *Structured Synchronising Merge* is associated with and it must merge all of the branches emanating from the *Multi-Choice*. These branches must either flow from the *Structured Synchronising Merge* without any split or join or they must be structured in form (i.e., balanced splits and joins).

Implementation
We mark the support for this pattern as direct because it derives from a composition *of Multi-Choice* and *Synchronised* patterns.

One of the challenges of this pattern is knowing when it can execute, basing this decision on local information available during the course of execution. Critical to this decision is the knowledge of how many branches emanating from the *Multi-Choice* are active and require synchronisation. In [RUSSEL, N et al., 2006] the authors define several ways to tackle this issue.
The best solution they propose is structuring the process model following a *Multi-Choice* pattern such that the subsequent *Structured Synchronising Merge* always receives precisely one trigger on each of its incoming branches (`cond1,…,condn`) and no additional knowledge is required to decide whether it should be enabled. This approach makes sure the merge construct always occurs in a structured context.

Our solution preserves a structure that requires no additional knowledge to enact the *Structured Synchronising Merge* behaviour, yet being compositional and providing a clear *bypass* path around each branch. Moreover, it inherits the property of decoupling the evaluation of the conditions and their data from the *Multi-Choice* block.

Listing 22: Structured Synchronising Merge – Centralised

```
i1( c );
p1@A( c )( c );
{
 {
  if( c.cond1 ){
   p2@B( c.c1 )( c.c1 );
   p4@D( c.c1 )( c.c1 )
  };
  p5@E( c.c1 )( c.c1 )
 }
 |
 {
  if( c.cond2 ){
   p3@C( c.c2 )( c.c2 )
  };
  p6@E( c.c2 )( c.c2 )
 }
};
o1@DefaultOutput1( c )
```

Both centralised and distributed implementations (respectively in Listings 22 and 23) of the *Structured Synchronising Merge* are composed by *i*) a set of non-splitting or balanced-splitting branches firing out of a *Multi-Choice* block and *ii*) a final *Synchronisation* block.

Listing 23: Structured Synchronising Merge — Distributed

```
// Service A
main
{
 i1( c );
 {
  if( cond1 ){
   p1@B( c )
  } else {
   p4@E( c )
  }
 }
 |
 {
  if( cond2 ){
   p2@C( c )
  } else {
   p5@E( c )
  }
 }
}
// Service E
main
{
 {
  p4( c.c1 )
  |
  p5( c.c2 )
 };
 o1@DefaultOutput1( c )
}
```

Local Synchronizing Merge

Definition
The convergence of two or more branches that diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

Implementation
The requirement of this pattern is captured by the implementation given for the *Structured Synchronizing Merge*, where the information about the enabled branches is communicated directly by the *Multi-Choice* component.

General Synchronizing Merge

Definition
The convergence of two or more branches, that diverged earlier in the process, into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any future time.

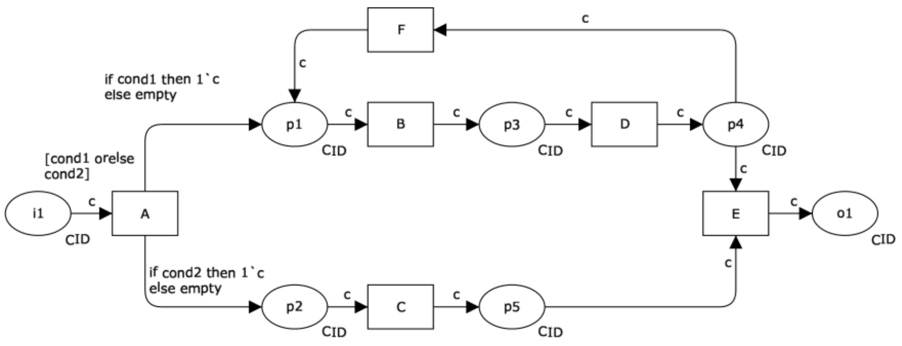


Figure 14: General Synchronising Merge pattern diagram

Implementation
To support this pattern, we need change the structure of the SOA derived from its CPN model. This is due to the races between services. Hence, we assign a “not direct” support for this pattern in Jolie. The graphical representation of the *General Synchronizing Merge* highlights that there is no bypass path for a false evaluation of `cond1` or `cond2`, thus ending with transition `E`, i.e., the synchronising construct, deadlocked. This derives from the requirement of this pattern. It models an unstructured merge where `E` has no local knowledge about which branch is enabled and if they will be enabled in the future, respectively due to lack of bypass paths and allowance for diverging loops.

The centralised implementation, in Listing 24, is similar to the one provided for the *Structured Synchronizing Merge*. However, in this case a false evaluation of `cond1` or `cond2` shall lead to a stuck state. This feature is provided by the `linkIn-linkOut` constructs, which realise a token-request/token-release mechanism. In the orchestrator, the race condition (Lines 6-25) translates into a parallel invocation of operation `p4` on both services `E` and `F`, using a variation of the *Simple Merge* to determine the winner of the race, i.e., the first that responds to the request. The distributed version has no need for such constructs because, if any condition evaluates to false, the subsequent services hang waiting for an incoming message. Transitions `F` and `E` realise a race on place `p4`. Also the distributed version is similar to the one provided for the *Structured Synchronizing Merge*. In particular, we realise the race between services `E` and `F` in service `D`, Lines 17-35 of Listing 25. Notably, the realisation of service `D` is equivalent to the one provided for the orchestrator.

Listing 24: General Synchronising Merge – Centralised

```
1  define branch_1
2  {
3   p1@B( c )( c1 );
4   p3@D( c1 )( c1 );
5   {
6    {
7     p4@F( c1 )( cF );
8     synchronized ( race_token ){
9      if(!is_defined( f_branch )){
10       f_branch = true
11      }
12     }
13    }
14    |
15    {
16     p4@E( c1 )( cE );
17     synchronized ( race_token ){
18      if(!is_defined( f_branch )){
19       f_branch = false
20      }
21     }
22    }
23   };
24   if( f_branch ){
25    undef ( f_branch ); branch_1
26   }
27 }
28
29 define branch_2
30 {
31  p2@C( c )( c2 );
32  p5@E( c2 )( c2 )
33 }
34
35 main
36 {
37  i1( c );
38  p1@A( c )( c );
39  {
40   if( cond1 ){
41    branch_1 ; linkOut(token_cond1)
42   }
43   |
44   if( cond2 ){
45    branch_2 ; linkOut(token_cond2)
46   }
47  };
48  {
49   linkIn( token_cond1 )
50   | linkIn( token_cond2 )
51  };
52  o1@DefaultOutput1( c )
53 }
```

Listing 25: General Synchronising Merge – distributed

```
// service A
main {
 i1( c );
 {
  if( cond1 ){
   p1@B( c )
  }
  | if( cond2 ){
   p2@C( c )
  }
 }
}
// service D
main {
 p3( c );
 {
  {
   race@F()();
   synchronized ( token ){
    if( ! is_defined ( resp ) ){
     branch_f = true
    }
   }
  }
  |
  {
   race@E()();
   synchronized ( token ){
    if( ! is_defined ( resp ) ){
     branch_f = false
    }
   }
  }
 }
};
if( branch_f ){ p4@F( c )}
else { p4@E( c ) }
}
// service E
main {
 [ race()(){ nullProcess }]{
  nullProcess }
 [ p4( c ) ]{
  p5( c );
  o1@DefaultPort1( c )
 }
 [ p5( c ) ]{
  p4( c );
  o1@DefaultPort1( c )
 }
}
// service F
main {
 [ p4( c ) ]{ p1@B( c )}
 [ race()(){ nullProcess }]{
  nullProcess }
}
```

6.4 Advanced Partial Synchronisation Patterns

In the context of WPs, a *Discriminator* describes a situation in which the construct waits for 1 out of *m* branches to fire its output. The *Partial Join* is a generalisation of the *Discriminator*, where n out of m branches should be merged before firing the output. Hence, since the *Discriminator* is a particular case of partial join where *n* = 1, we do not directly discuss about *Structure*, *Blocking*, and *Cancelling Discriminator* patterns as their behaviours are captured by their *Partial Join* correspondent.

Structured Partial Join

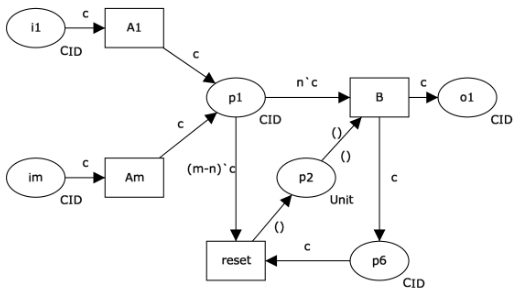


Figure 15: Structured Partial Join pattern diagram

Definition
The convergence of *M* branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when *N* of the incoming branches have been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled.

Implementation
Both centralised and distributed implementations offer a direct support for this pattern since it derives from a composition of directly supported pattern.

The centralised solution, in Listing 26, composes into a *Synchronisation* all the incoming branches `i1,…,im`. Each time an incoming operation is received, it enables a *Thread Merge* procedure, namely `check_and_send`. At the `n`-th incoming operation, the procedure sends the collected messages to service `B`.

Notably, we do not include in the implemented SOAs (both the centralised and the distributed) the service `reset`. In the centralised implementation the behaviour of `reset` emerges from the *Synchronisation* pattern. When each scope has executed, procedure `main` terminates and the master service can restart its behaviour. In the distributed solution, in Listing 27, service `B` coalesces the behaviour of service `reset` with a *Sequence* of *Thread Merges*.

Listing 26: Structured Partial Join – centralised

```
define check_and_send
{
 if( i==n ){
  p1@B( c )( c );
  o1@DefaultPort1( c )
 }
}

main
{
 {
  // code for op. i1
  |
  {
   in( cn );
   pn@An( cn )( cn );
   synchronized ( token ){
    c[ i ] << cn;
    i++;
    check_and_send
   }
  }
  |
  // code for op. im
 }
}
```

Listing 27: Structured Partial Join – distributed

```
// Service A1,…,Am
main
{
 in( c );
 p1@B( c )
}

// Service B
main
{
 p1( c[ i ] );
 for ( i=1, i<n, i++ ){
  p1( c[ i ] )
 };
 o1@DefaultOutput1( c );
 for ( i=0, i<m-n, i ++){
  p1 ()
 }
}
```

Blocking Partial Join

Definition
The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches has been enabled (where $2 = n < m$). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

Implementation
We mark the support for this pattern as “not direct”, due to its dependency from the *Generalised AND-Join* pattern.

The centralised implementation, in Listing 28, applies the same principle described by the *Generalised AND-Join*. Each incoming operation `i1,…,im` can fire multiple times and each firing is stored for future executions. Procedure `queueOp_in` stores the message of operation `in` into a specific queue. Then, procedure `checkOp_in` controls the state of the queue to decide whether to fire operation `pn` at service `An` of the *Structured Partial Join*. The procedure updates the counter of the fired operation accordingly.

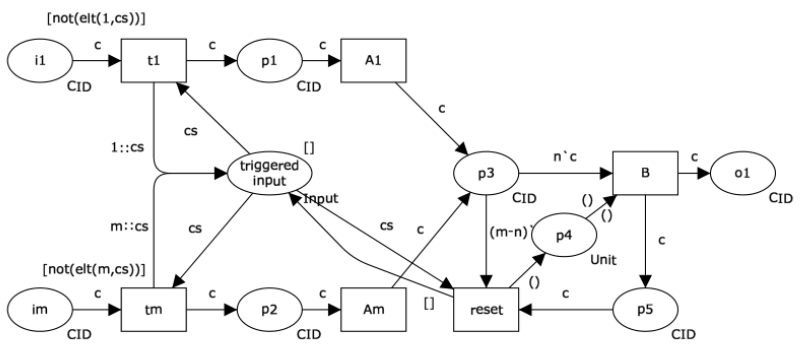


Figure 16: Blocking Partial Join pattern diagram

The procedure `check_and_send` enacts the behaviour of the pattern, depending on the number of fired operations. When the `m`-th operation has fired, procedure `reset` removes the sent messages from the queues, resets the counter of operations, and executes procedures `checkOp_i1,…,checkOp_im` in order to fire previously queued messages.

In the distributed approach, services `T1,…,Tm` represent a distributed version of the *Generalised AND-join*. In Listing 29, each `Ti`, `i` in `{1,…,m}`, controls the queue relative to its incoming operation `i1,…,im`. Service `B` implements the same merging behaviour as presented for the *Structured Partial Join*, although after the reception of the `m`-th message, it invokes the operation `reset` on all `T1,…,Tm` for resetting the pattern. The operation `reset` removes previously sent messages from the queues and checks if other messages are present for subsequent executions.

Listing 28: Blocking Partial Join – centralised

```
define checkOp_in {
 if( queueSizeOp_in == 1 ||
    ( reset_token &&
      queueSizeOp_in > 0 )){
  peekQueueOp_in ;
  pn@An( c_loc )(c[ counter ]);
  counter ++
 }
}
define reset {
 undef ( counter );
 {
  dequeueOp_i1
  | // …
  | dequeueOp_im
 };
 reset_token = true ;
 {
  checkOp_i1
  | // …
  | checkOp_im
 };
 undef ( reset_token )
}
define check_and_send {
 if( counter == n ) {
  p1@B( c )( c );
  o1@DefaultOutput1( c )
 };
 if( counter == m ){
  reset ;
  check_and_send
 }
}
main {
 // [ i1 ] { … }
 // …
 [ in( c_loc ) ]{
  queueOp_in ;
  checkOp_in ;
  check_and_send
 }
 // …
 // [ in ] { … }
}
```

Listing 29: Blocking Partial Join — distributed

```
// services T1,…,Tm
main
{
 [ in( c ) ]{
  queueOp_in ;
  if( queueSizeOp_in == 1 ){
   p1n@An( c )
  }
 }
 [ reset() ]{
  dequeueOp_in ;
  if( queueSizeOp_in > 0 ){
   peekQueueOp_in ;
   p1n@An( c )
  }
 }
}

// service B
main
{
 p3( c[ 0 ] );
 for( i = 1, i < n, i++ ){
  p3( c[ i ] )
 };
 o1@DefaultOutput1( c );
 for( i = 0, i < m-n, i++ ){
  p3()
 };
 {
  reset@T1()
  | // …
  | reset@Tm()
 }
}
```

Cancelling Partial Join

Definition
The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when *N* of the incoming branches have been enabled. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.

Implementation
The *Cancelling Partial Join* is built on top of the *Structured Partial Join* and includes it as its subcomponent.
We assign a "direct" support to this pattern as it derives from the composition of directly supported patterns. One of the difficulties with this pattern is that it realises a race among transitions `A1,…,Am`, `S1,…,Sm`, and input places `i1,…,im`.

The centralised version renders the race as a parallel composition of *Exclusive Choices* for evaluating the shared flag `skip` in each branch. When the `n`-th message arrives, the procedure `check_and_send` sets the flag `skip` to `true`, routing the firing of the remaining operations to `S1,…,Sm` until the `m`-th messages reaches the orchestrator and the pattern resets.

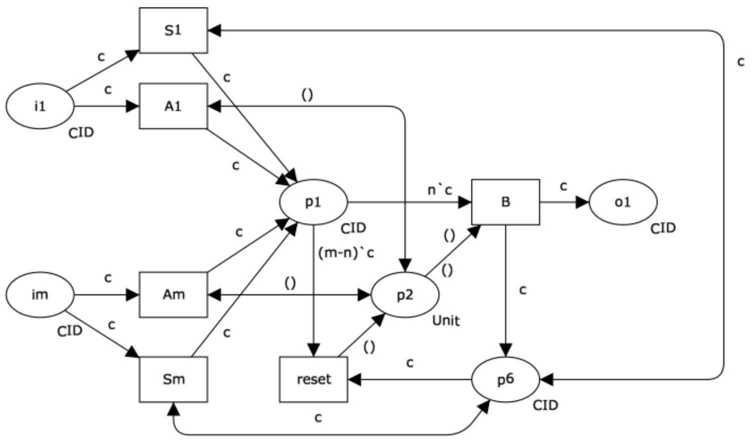


Figure 17: Cancelling Partial Join pattern diagram

We identify two difficulties in the distributed implementation of this pattern. First, we need to coalesce the race into a service that evaluates whether to route incoming messages on i1,…,im towards A1,…, Am or S1,…, Sm. To this end, we introduce in the SOA the services T1,…,Tm. These services encode the race into an internal *Exclusive Choice*. Second, we employ RRs to implement the interaction described by the double-sided arcs between transitions S1,…,Sm and place p3. T1,…,Tm invoke operation p3 each time they receive a message on operation i1,…,im. This guarantees a symmetric knowledge on the state of the pattern between T1,…,Tm and the joining service B. Services T1,…,Tm run simultaneously and invoke operation p3 in parallel, possibly interleaving with joining operation p1. To prevent inconsistencies between allowed firings on p3 and joined operations on p1, we need to specify a mechanism that coordinates these two operations of service B. To this end, we apply a modified version of the *Thread Merge* for the requests towards p3.

Listing 30: Cancelling Partial Join – Centralised

```
define check_and_send {
 if( counter == n ){
  p1@B( c )( c );
  o1@DefaultPort1( c );
  skip = true
 }
}
main {
 {
  // …
  |{
    in( cn );
    synchronized ( token ){
     if( skip ){
      p2n@Sn( cn )( c2 )
     } else {
      p1n@An( cn )( cn );
      c[ i ] << cn;
      counter ++;
      check_and_send
     }
    }
   }
  | // …
 };
 undef ( skip )
}
```

In this way, regardless to the number of invocations of p1, service T1,…,Tm would know whether to execute A1,…,Am or S1,…,Sm.

Listing 31: Cancelling Partial Join — Distributed

```
// Services T1,…,Tm
main
{
 i1( c );
 p3@B( c )( skip );
 if ( skip ){
  p21@S1( c )
 } else {
  p11@A1( c )
 }
}

// Service B
main
{
 [ p1( c[ 0 ] )]{
  for( i = 1, i < n, i++ ){
   p1( c[ i ] )
  };
  o1@DefaultOutput1( c );
  for( i = 0, i < m-n, i ++){
   p1()
  };
 undef ( skip.( SID ) )
 }
 [ p3( c )( response ){
  response = false ;
  local_skip -> skip .( SID );
  synchronized ( local_skip ){
   local_skip++;
   if ( local_skip > n ){
    response = true
   }
  }
 }]{ nullProcess }
}
```

5 The Upload Service Use Case

Here, we consider a realistic use case to illustrate how an SOA modelled as a Coloured Petri net can be translated into an executable SOA by using our design pattern implemented in Jolie. First we describe the communications in the system by means of Coloured Petri nets, showing how the most relevant patterns are employed. Then we provide the Jolie implementation of the commented patterns. Our use case describes the interactions between a User, a file upload Service Provider, and an Identity Provider. Figure 18 depicts the overall flow of interaction. In the figure, for the sake of clarity, the double-line bordered boxes act as placeholders for the two subnets reported in Figures 19 and 20.

Depicted in Fig. 18, the interaction starts from the User that requests the service. The Service Provider asks the User for authentication, redirecting the request to the Identity Provider. The Identity Provider authenticates its users through a multi- factor mechanism, allowing users to identify themselves with three different authentication procedures: *i*) HTTP basic access authentication, *ii*) mobile phone, and *iii*) smart card. In order to authenticate the User, the Identity Provider requires at least two successful authentications. Figure 19 describes the behaviour of the multi-factor authentication in terms of the *Cancelling Partial Join* pattern. In this case, the transition *Receive Authentication Confirm* fires as soon as it receives two tokens of authentication. After such a transition has fired the remaining authentication procedure is skipped.

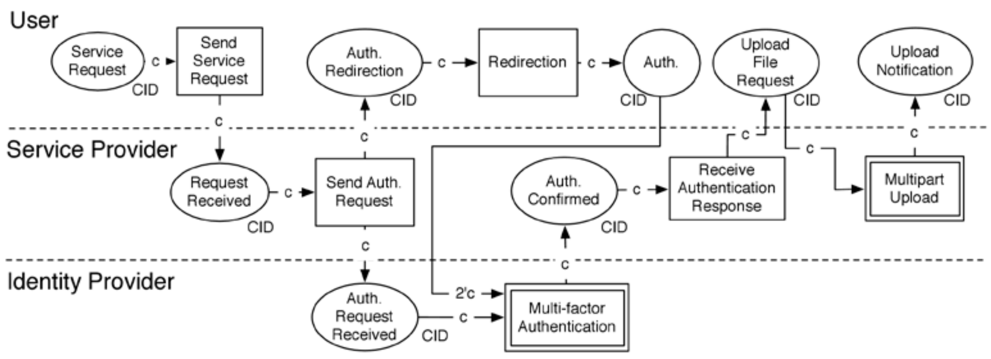


Figure 18: The Upload Service net

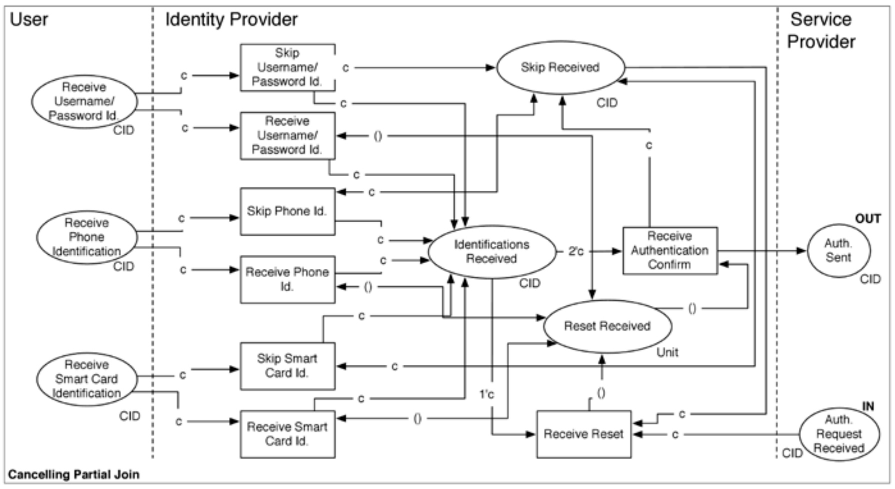


Figure 19: Multi-factor Authentication subnet

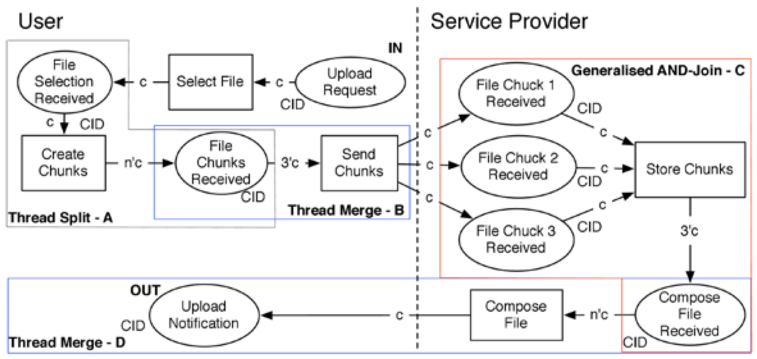


Figure 20: Multipart Upload subnet

Listing 32 shows the implementation of the multi-factor authentication as an orchestrator.

After the successful authentication, the thread of control passes back to the Service Provider with another distributed *Sequence* which notifies the User (s)he can proceed to upload the file. The User and the Service Provider enter the Multipart Upload interaction whose behaviour results from the composition of several patterns. Fig. 20 depicts such interactions and highlights the most relevant WPs. Fig. 21 depicts the architectural view of the translation following the same informal representation used in Fig. 2.

The User-controlled part of the interaction mixes centralised and distributed WPs. Listing 33 reports the code relative to the services `orchestrator` and `SendChunks` at User's side. When the `uploadRequest` arrives (Line 1), the orchestrator requires the User to select a file, passing the thread of control as a centralised *Sequence* to service `SelectFile` (Line 2). At file selection, the thread of control returns to the orchestrator that passes it to service `CreateChunks` (Line 3). The service employs a centralised *Thread Split* (A in Fig. 20) to split the file into `n` chunks. Then the orchestrator implements a centralised *Thread Merge* (B in Fig. 20) to collect triplets of chunks and send them to service `SendChunks` (Lines 5-7). Since the orchestrator passes the thread of control to the invoked service and waits for its response, we can coalesce the OW operations between them into one RequestResponse. `SendChunks` implements a distributed *Parallel Split* to forward each chunk in parallel to the Service Provider (Lines 11-13).

Listing 32: Multi-factor Authentication – Orchestrator

```
execution { sequential }
constants { n = 2 }
init {
 receivedAuth -> global.receivedAuth
}
define check_and_send {
 if( receivedAuth == n ){
  receiveIds@ReceiveAuthConfirm(c)(c);
  sendAuthentication@OUT( c );
  skip = true
 }
}
main {
 authRedirectReceived( request );
 {
  {
   sentUP( upData );
   if( skip ){
    sendUP@SkipUP( c1 )( c1 )
   } else {
    sendUP@ReceiveUP( c1 )( c1 );
    c[ receivedAuth ] << c1;
    receivedAuth++;
    check_and_send
   }
  }
  |
  {
   sentPhone( phoneData );
   if( skip ){
    sendPhone@SkipPhone( c2 )( c2 )
   } else {
    sendPhone@ReceivePhone( c2 )( c2 );
    c[ receivedAuth ] << c2;
    receivedAuth ++;
    check_and_send
   }
  }
  |
  {
   sentSIM ( simData );
   if( skip ){
    sendSIM@SkipSIM( c3 )( c3 )
   } else {
    sendSIM@ReceiveSIM( c3 )( c3 );
    c[ receivedAuth ] << c3;
    receivedAuth ++;
    check_and_send
   }
  }
 };
 undef ( skip );
 receivedAuth = 0
}
```

At Service Provider's side the service `StoreChunks` employs a centralised *Generalised AND-Join* (C in Fig. 20) to receive the chunks (Listing 34, implemented using the standard library for queues of Jolie). When the `n`-th chunk reaches the service, it passes the thread of control with a distributed *Sequence* to service `ComposeFile` (Listing 35) that employs a centralised *Thread Merge* (D in Fig. 20) to restore the chunks into a single file. Finally a distributed *Sequence* returns the thread of control to the User, notifying the success of the upload procedure.

Listing 33: Multipart Upload – User's side

```
// orchestrator
uploadRequest (c);
selectFile@SelectFile(c)(c);
createChunks@CreateChunks(c)(c);
for ( i=0, i <#c, i++ ){
 r.c1=c[i++];
 r.c2=c[i++];
 r.c3=c[i];
 sendTriplet@SendChunks(r)()
}
// SendChunks
sendTriplet( c )(){
 sendFileChunk1@StoreChunk(c.c1)
 | sendFileChunk2@StoreChunk(c.c2)
 | sendFileChunk3@StoreChunk(c.c3)
}
```

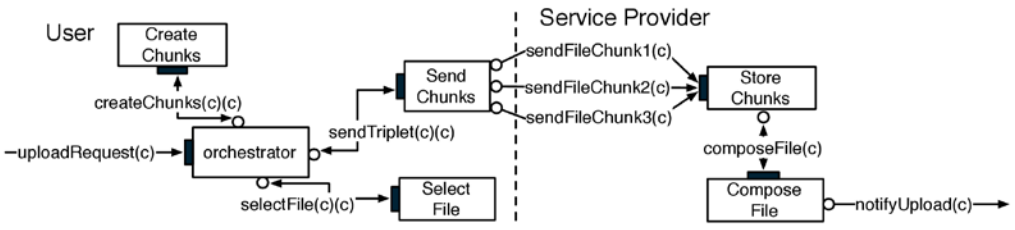


Figure 21: The architectural view of Multipart Upload in Fig. 2

Listing 34: Multipart Upload, StoreChunks

```
execution { sequential }
define check_and_send {
 size@QueueUtils( q1 )(ch1_count);
 size@QueueUtils( q2 )(ch2_count);
 size@QueueUtils( q3 )(ch3_count);
 if( chunk1_count > 0 &&
  chunk2_count > 0 &&
  chunk3_count > 0 ){
  // Take c1, c2, and c3
  poll@QueueUtils( q1 )(chks.c1);
  poll@QueueUtils( q2 )(chks.c2);
  poll@QueueUtils( q3 )(chks.c3);
  // and send them to ComposeFile
  composeFile@ComposeFile( chks )
}}
main {
 [ sendFileChunk1( c ) ]{
  qer.queue_name = q1;
  qer.element << c;
  push@QueueUtils( qer )();
  check_and_send }
 [ sendFileChunk2( c ) ]{
  qer.queue_name = q2;
  qer.element << c;
  push@QueueUtils( qer )();
  check_and_send }
 [ sendFileChunk3( c ) ]{
  qer.queue_name = q3;
  qer.element << c;
  push@QueueUtils( qer )();
  check_and_send }
}
```

Listing 35: Multipart Upload, ComposeFile

```
constants {
 chunksNumber = n,
 chunkThreads = 3
}

define storeChunks
{
 fileChunks[ #fileChunks ] = c.c1;
 fileChunks[ #fileChunks ] = c.c2;
 fileChunks[ #fileChunks ] = c.c3
}

main
{
 for ( recChunks = 0,
  recChunks < chunksNumber,
  recChunks += chunkThreads ){
   composeFile( c );
   storeChunks
 };
 receiveUploadNotification@User(c)
}
```

6 Conclusions

Contributions of this work are: *i*) the definition of a methodology for translating CPN-modelled SOAs into composable and executable ones; *ii*) the creation of a collection of implemented Workflow Patterns. Such implementations follow both a centralised and a distributed approach to allow developers the flexibility to choose one and to mix them. A realistic use case substantiate our claim that the patterns obtained in this way can be effectively used for building real SOAs starting from abstract specifications. In addition, *iii*) our work also allows us to provide a pragmatic assessment on the expressiveness of the Jolie language. Table 1 summarizes the results of such an assessment. For each pattern, we indicate in the second column the kind of support offered by Jolie: "+" means direct support, i.e., the implementation of the pattern either uses some specific primitives provided by the language or is a composition of directly supported patterns. "+/–" indicates a "not direct" support, i.e., the translation of the CPN of the pattern does not completely follow the rules described in Section 3 although it complies with the general structure of the pattern. In the third column of Table 1 we indicate the specific Jolie primitive and/or the other patterns used to implement a given pattern. We report both the centralised and distributed implementations that, as expected, in some cases vary. As shown in Table 1 we can conclude that Jolie can directly implement most of the considered Workflow Patterns.

6.1 Related Work

A close concept to Workflow Patterns is that of Service Interaction Pattern (SIPs), introduced in [BARROS, A. *et al*., 2005]. SIPs define recurring interaction patterns among services but, differently from Workflow Patterns, they are informally specified and therefore not employable in this work. Variants of Petri nets have been used for system modelling [MENDES, J *et al*., 2010] and static analysis [LOHMANN, N *et al*., 2008b]. An inspiring work that considers a direct translation from Petri nets to a service-oriented language (Abstract BPEL) is [LOHMANN, N *et al*., 2008a]. However, the proposed translation do not automatically derives all the details of the implementation, which prevents a direct execution of the code. Finally WPI used WPs as a tool to evaluate the expressive power of business process languages. Particularly relevant are the cases of BPEL [WOHED, P *et al*., 2003] and of BPML [VAN DER AALST, W *et al*., 2002].

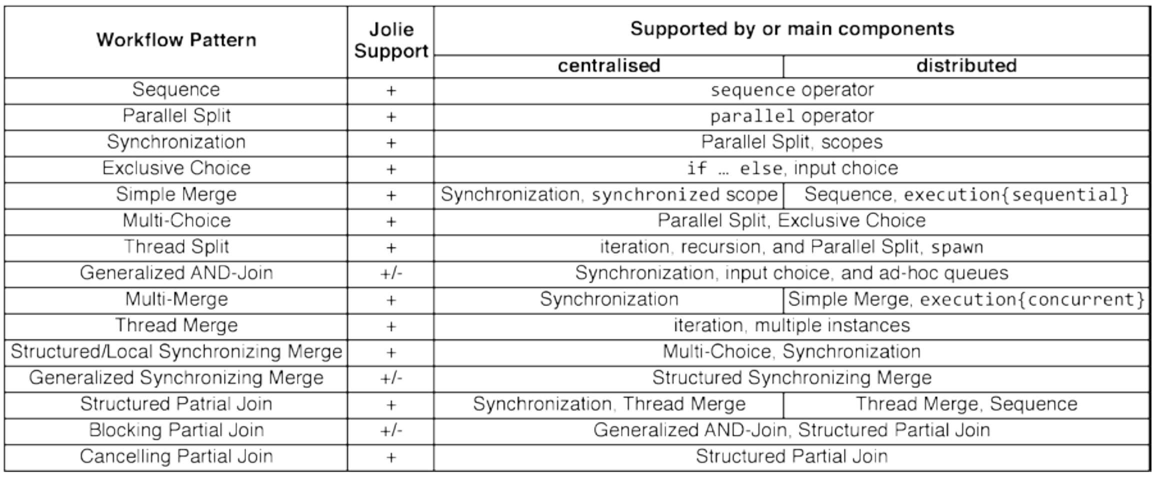
6.2 Future Work

We plan to provide a formal definition of our technique for translating CPNs into Jolie code. Such a formalisation would enable to mechanically translate CPN-modelled SOAs into executable ones, also applying known methodologies of static analysis to assess properties of SOAs implemented in Jolie.

We also plan to use the implemented Workflow Patterns developed in this work to offer pattern composition as APIs [GUIDI, C *et al*., 2014] to clients. Finally, a natural extension of this work is to investigate the implementations of the remaining patterns described by the WPI that comprehend *multiple-instances*, *state*, *cancellation*, *completion*, *termination*, and *triggering* patterns.

7 Acknowledgements

The Danish Council for Independent Research | Technology and Production supported Montesi.



Workflow Pattern	Jolie Support	Supported by or main components	
		centralised	distributed
Sequence	+	`sequence` operator	
Parallel Split	+	`parallel` operator	
Synchronization	+	Parallel Split, scopes	
Exclusive Choice	+	`if … else`, input choice	
Simple Merge	+	Synchronization, `synchronized` scope	Sequence, `execution{sequential}`
Multi-Choice	+	Parallel Split, Exclusive Choice	
Thread Split	+	iteration, recursion, and Parallel Split, `spawn`	
Generalized AND-Join	+/-	Synchronization, input choice, and ad-hoc queues	
Multi-Merge	+	Synchronization	Simple Merge, `execution{concurrent}`
Thread Merge	+	iteration, multiple instances	
Structured/Local Synchronizing Merge	+	Multi-Choice, Synchronization	
Generalized Synchronizing Merge	+/-	Structured Synchronizing Merge	
Structured Patrial Join	+	Synchronization, Thread Merge	Thread Merge, Sequence
Blocking Partial Join	+/-	Generalized AND-Join, Structured Partial Join	
Cancelling Partial Join	+	Structured Partial Join	

Table 1: Evaluation for *basic* and *advanced branching and synchronization* WPs in Jolie.

8 References

[BARROS, A *et al.*, 2005] BARROS, Alistair; DUMAS, Marlon; TER HOFSTEDE, Arthur H. M. *Service interaction patterns*. In: Business Process Management. Springer Berlin Heidelberg, 2005. p. 302-318.

[CARBONE, M *et al.*, 2013] CARBONE, Marco; MONTESI, Fabrizio. Deadlock-freedom-by-design: multiparty asynchronous global programming. ACM SIGPLAN Notices, 2013, 48.1: 263-274.

[WMC, 1999] WORKFLOW MANAGEMENT COALITION. *Workflow Management Coalition Terminology & Glossary*. 1999.

[ERL, T, 2005] ERL, Thomas. *Service-oriented architecture: concepts, technology, and design.* Pearson Education India, 2005.

[GUIDI, C *et al.*, 2014] GUIDI, Claudio; GIALLORENZO, Saverio; GABBRIELLI, Maurizio. *Towards a composition-based APIaaS layer*. to appear in CLOSER 2014, SciTePress, 2014.

[GUIDI, C *et al.*, 2006] GUIDI, Claudio; LUCCHI, Roberto; GORRIERI, Roberto; BUSI, Nadia; ZAVATTARO, Gianluigi. *SOCK: a calculus for service oriented computing.* In: ICSOC 2006. Springer Berlin Heidelberg, p. 327-338. 2006.

[JENSEN, K *et al.*, 2007] JENSEN, Kurt; KRISTENSEN, Lars Michael. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Pages I-XI, 1-384. Springer. 2009

[JOLIE, 2014] Jolie Website. http://jolie-lang.org. 2014.

[LANESE, I *et al.*, 2008] LANESE, Ivan; GUIDI, Claudio; MONTESI, Fabrizio; ZAVATTARO, Gianluigi. *Bridging the gap between interaction-and process-oriented choreographies*. In: Software Engineering and Formal Methods, 2008. SEFM'08. Sixth IEEE International Conference on. IEEE. p. 323-332. 2008.

[LOHMANN, N *et al.*, 2008a] LOHMANN, Niels; KLEINE, Jens. *Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes*. In: Modellierung. p. 14. 2008.

[LOHMANN, N *et al.*, 2008b] LOHMANN, Niels; KOPP, Oliver; LEYMANN, Frank; REISIG, Wolfgang. *Analyzing BPEL4Chor: Verification and participant synthesis*. In: Web Services and Formal Methods. Springer Berlin Heidelberg. p. 46-60. 2008.

[MAYER, P et al., 2009]	MAYER Philip; KOCH Nora; SCHROEDER Andreas. *The UML4SOA Profile*. Ludwig-Maximilians-Universitaet Muenchen, 2009.
[MENDES, J *et al.*, 2010]	MENDES, Joao Marcos; LEITAO, Paulo; RESTIVO, Francisco; COLOMBO, Armando Walter. *Composition of petri nets models in service-oriented industrial automation*. In INDIN'10, pages 578–583. 2010.
[MONTESI, F , 2010]	MONTESI, Fabrizio. Jolie: a service-oriented programming language. M.Sc. Thesis. 2010.
[MONTESI, F *et al.*, 2011]	MONTESI, Fabrizio; CARBONE, Marco. *Programming services with correlation sets*. In ICSOC 2011, pages 125–141. 2011.
[MONTESI, F *et al.*, 2014]	MONTESI, Fabrizio; GUIDI, Claudio; and ZAVATTARO, G. *Service Oriented Programming with Jolie*, volume 1 of Web Services Foundations. Pages 79-106. 2013.
[OASIS, 2006]	ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS. *Web Services Business Process Execution Language*. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. 2006.
[OASIS, 2012]	ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS. *Reference Architecture Foundation for SOA Version 1.0.* http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html. 2012.
[OMG, 2009]	OBJECT MANAGEMENT GROUP. *Service oriented architecture modeling language (SoaML)-specification for the UML profile and metamodel for services*. 2008.
[REISIG, W, 1985]	REISIG, Wolfgang. *Petri Nets: An Introduction*, volume 4 of Monographs in Theoretical Computer Science. An EATCS Series. Springer. 1985.
[RUSSEL, N *et al.*, 2006]	RUSSELL, Nick; TER HOFSTEDE, Arthur H. M. ; MULYAR, Nataliya. *Workflow control-flow patterns: A revised view*. Technical report. 2006.
[VAN DER AALST, W *et al.*, 2002]	VAN DER AALST, Wil M. P.; DUMAS, Marlon; TER HOFSTEDE, Arthur H. M.; WOHED, Petia. *Pattern-based analysis of BPML (and WSCI)*. 2002.
[VAN DER AALST, W *et al.*, 2003]	VAN DER AALST, Wil M. P.; TER HOFSTEDE, Arthur H. M.; KIEPUSZEWSI, Bartosz; BARROS, Alistair. *Workflow patterns*. Distributed. Parallel Databases, 14(1): 5–51. 2003.
[WS-CDL, 2004]	WS-CDL Working Group. *Web services choreography description language version 1.0*. http://www.w3.org/TR/ws-cdl-10/. 2004.
[WOHED, P *et al.* 2003]	WOHED; Petia; VAN DER AALST, WIL M. P.; DUMAS, Marlon; TER HOFSTEDE, Arthur H. M. *Analysis of Web Services Composition Languages: The Case of BPEL4WS*. In Proc. of ER, LNCS 2813, pages 200–215. Springer Verlag. 2003.