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

Techincal Report

Maurizio Gabbrielli[1], Saverio Giallorenzo[1], and Fabrizio Montesi[2]

[1] Dipartimento di Informatica - Univ. di Bologna / INRIA
[2] IT University of Copenhagen

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 interactions in SOAs, however 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 Jolie SOAs (our target language), exploiting a collection of recurring control-flow patterns, called Workflow Patterns, as composable blocks of the translation. We validate our approach with a realistic use case. In addition, we pragmatically asses the expressiveness of Jolie wrt the considered WPs.

Table of Contents

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 [1] (SOA), services are composed by coordinating their communications into a flow of interactions. Several tools have been presented [2–4] to assist the process of SOA design, each focusing on a particular aspect of the system, e.g., the architectural composition, the interaction among components, etc. Coloured Petri nets [5] (CPNs) are a formal yet intuitive graphical tool, largely employed in business process modelling [6] 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* [6] (WP). In particular we remark the exhaustive set of patterns of interaction, dubbed *Control-Flow Workflow Patterns* [7][3], modelled as CPNs. Since CPNs are composable, our idea, depicted by the scheme 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.

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

To realise our proposal, we provide the implementation of a substantial set of WPs. 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

[3] here referred as Workflow Patterns for simplicity.

track the whole system. On the other hand, some scenarios strictly require a distributed approach, e.g., an interaction that comprehends different parties. In § 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 § 3. Notably, 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 [8] but we choose to implement WPs in Jolie [9,10] 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 [11] 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, and to Jolie. CPNs are a modelling language that combine elements inherited from *Petri Nets* [12] (PNs) and capabilities of high-level programming languages that allow the construction of parametrised models. The main ingredients of a 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, 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 that we use in this work are taken from [7] and we use the definitions and the assumptions made in that paper 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.

2.2 Composing services in Jolie

We now present the basic concepts needed for understanding the behaviour of services written in Jolie. Jolie programs contain two parts related to the *behaviour* of a service and to its *deployment*. Here we are interested only in the behavioural aspect, which defines the instructions to be performed and the input/output communications of a service. Remarkably, the independence of behaviour from deployment information in Jolie applications allows to seamlessly integrate heterogeneous networks of Jolie and non-Jolie services that communicate on different media (e.g., TCP/IP, Bluetooth, Java RMI, unix local sockets, etc.) and with various protocols (e.g., SODEP, SOAP, HTTP, JSON-RPC, XML-RCP, and their equivalent over SSL).

Jolie combines message-passing and imperative programming style and allows scopes, indicated by curly brackets `{...}`, to represent proced ures. 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. The `main` procedure is the entry point of execution for each service. Conditions, loops, and sequence are standard. The *parallel* operator `|` states that both left and right operands execute concurrently — note that parallel operator has always priority on sequence[4]. Jolie provides also an input-guarded choice with the following syntax: $[\eta_1]\{B_1\}\ldots[\eta_n]\{B_n\}$, where $\eta_i,\ \ i \in \{1,\ldots,n\}$, is an input statement and B_i is the related branch behaviour. 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-Way*s (OWs) and *Request-Response*s (RRs). On the sender's side, the former operation sends a message and immediately passes the thread of control to the subsequent activity in the process, while the latter sends a request and keeps the thread of control until it receives a response. On the receiver's side, OWs receive a message

[4] Scopes ease the definition of precedence between different code blocks, as shown at Lines 2-5 in Listing 1.1.

and store it into a defined variable, whilst RRs receive a message into a variable and send the content of the second variable as response.

Listing 1.1 exemplifies an SOA consisting 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`. When `B` receives a message on one of the corresponding OW operations, it stores the content of the message in the corresponding variable. 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`.

Listing 1.1: 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  }
```

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`[5], `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.

[5] default, if the `execution` statement is omitted

3 From Coloured Petri nets to Jolie SOAs

In this section we show how CPN models of Worklow 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. Furthermore, 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 [13]. In our approach, an *SOA instance* is a composition of instances of services which 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 [14] for more details). However here we omit the definition of correlation sets in our implementations, as they are not necessary for the definition of the behaviour of our services. An SOA can be realised by using a centralised or a distributed approach, usually referred to with the terminology "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. WS-BPEL [8] is the most known technology for this approach. In the second case, a choreography specifies the global behaviour of an SOA by defining, in an abstract way, the communication behaviour of the single services participating to the SOA, without introducing centralization points. Choreographic languages such as WS-CDL [15] have been specifically designed for this purpose. Recent works [16,17] introduced automatic projection techniques which 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 significative aspects concerning the expressive power of the implementation language (Jolie in our case), as we discuss in the Conclusions (§ 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`. 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[6] 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 1.2 reports the corresponding code of, respectively, the orchestrator of the centralised version and of the services in the distributed one.

Listing 1.2: Centralised (right) and distributed (left) implementations of CPN **A**.

```
1  //orchestrator
2  i1( c );
3  p1@A( c )( cond1 );
4  if( cond1 ){
5    p2@B( c )( c )
6  } else {
7    p3@C( c )( c )
8  };
9  p4@D( c )( c );
10 o1@DefaultOutput1( c )
```

```
1  //service A
2  i1( c );
3  if( cond1 ){  p1@B( c ) }
4  else {  p2@C( c ) }
5  // service B
6  p1( c ); p3@D( c )
7  // service C
8  p2( c ); p3@D( c )
9  // service D
10 p3( c ); o1@DefaultOutput1( c )
```

[6] not shown by the diagram

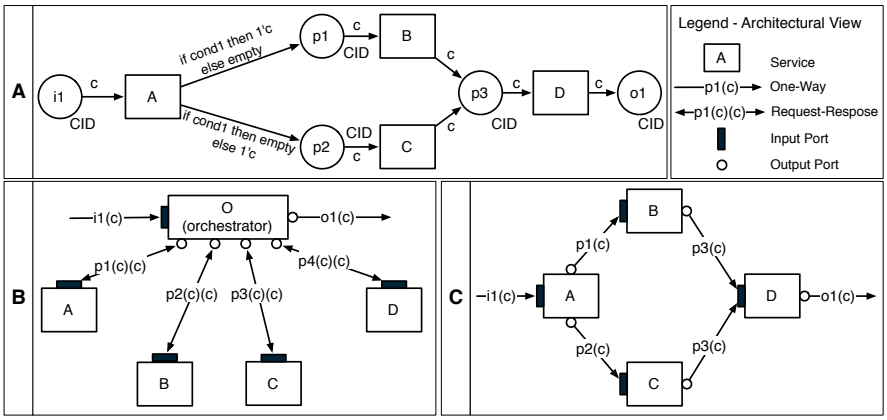

Fig. 2: **A** CPN model, its centralised (**B**), and distributed (**C**) implementations.

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 WPs in Jolie.

4.1 Basic Control-Flow Patterns

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

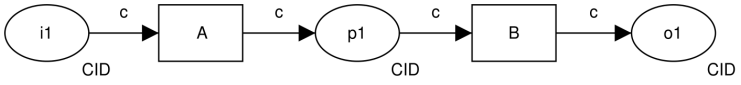

Fig. 3: Sequence pattern diagram

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

Listing 1.3: Sequence — centralised

```
1  i1( c );
2  i1@A( c )( c );
3  p1@B( c )( c );
4  o1@DefaultOutput1( c )
```

Listing 1.4: Sequence — distributed

```
1  // service A
2  {
3    i1( c );
4    p1@B( c )
5  }
6  // service B
7  {
8    p1( c );
9    o1@DefaultOutput1( c )
10 }
```

Parallel Split ***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 § 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 `ldots` to manage the parallel execution of the two branches emanating from transition A.

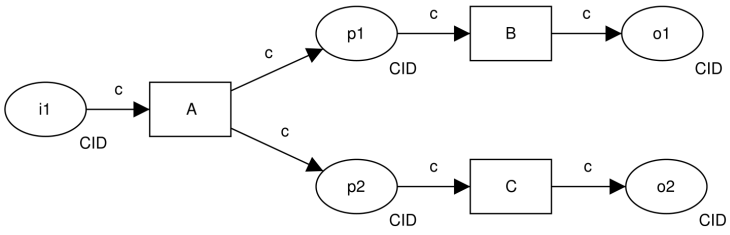

Fig. 4: Parallel Split pattern diagram

Listing 1.5: Parallel Split — centralised

```
1  i1( c );
2  {
3    p1@A( c )( c1 );
4    o1@DefaultOutput1( c1 )
5  }
6  |
7  {
8    p2@B( c )( c2 );
9    o2@DefaultOutput2( c2 )
10 }
```

Listing 1.6: Parallel Split — distributed

```
1  // service A
2  {
3    i1( c );
4    { p1@B( c ) | p2@C( c ) }
5  }
6  // service B
7  {
8    p1( c ); o1@DefaultOutput1( c )
9  }
10 // service C
11 {
12   p2( c ); o2@DefaultOutput2( c )
13 }
```

Synchronisation ***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.

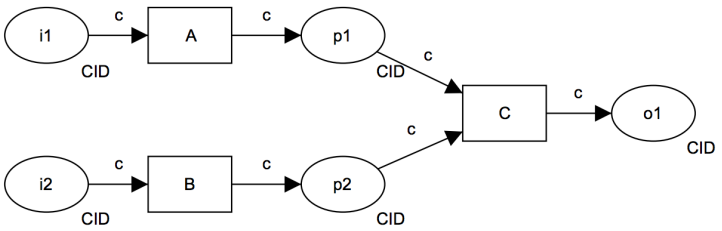

Fig. 5: Synchronisation pattern diagram

Implementation The *Synchronisation* complements the *Parallel Split*. The behaviour of the pattern is directly supported by the semantic of scopes `ldots` 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 1.7: Synchronisation — centralised

```
1  {
2    {
3      i1( c1 );
4      p1@A( c1 )( c.c1 )
5    }
6    |
7    {
8      i2( c2 );
9      p2@B( c2 )( c.c2 )
10   }
11 };
12 p3@C( c )( c );
13 o1@DefaultOutput( 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.

Listing 1.8: Synchronisation — distributed

```
1  // service C
2  {
3    {
4      p1( c1 )
5      |
6      p2( c2 )
7    };
8    o1@DefaultOutput1( c )
9  }
10 //service A
11 { i1( c ); p1@C( c ) }
12 //service B
13 { i2( c ); p2@C( c ) }
```

Exclusive Choice ***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.

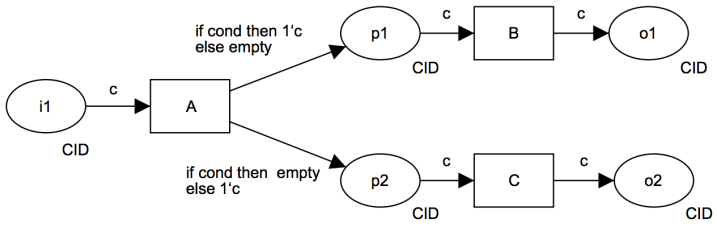

Fig. 6: Exclusive Choice pattern diagram

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 1.9, the orchestrator evaluates the condition `cond` and chooses whether to proceed on branch `B` or `C`. In Listing 1.10, we insert an additional service `P` that service `A` invokes on operations `p1` or `p2` after the evaluation of condition `cond`.

Listing 1.9: Exclusive Choice — centralised

```
1  i1( c );
2  p1@A( c )( cond );
3  if ( cond ){
4    p2@B( c )( c );
5    o1@DefaultOutput1( c )
6  } else {
7    p3@C( c )( c );
8    o2@DefaultOutput2( c )
9  }
```

Listing 1.10: Exclusive Choice — distributed

```
1  // service A
2  {
3    i1( c );
4    if ( cond ){ p1@P( c ) }
5    else { p2@C( c ) }
6  }
7  // service P
8  {
9    [ p1( c ) ]{ p3@B( c ) }
10   [ p2( c ) ]{ p4@C( c ) }
11 }
```

Simple Merge ***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.

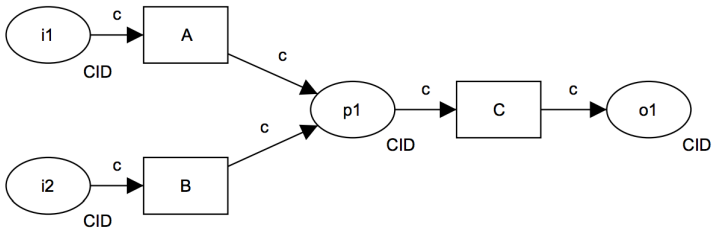

Fig. 7: Simple merge pattern diagram

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` have no dependency on the number of branches to be merged.

Listing 1.11: Simple Merge — centralised

```
1  {
2    i1( c1 );
3    i1@A( c1 )( c1 );
4    synchronized ( token ){
5      p1@C( c1 )( c1 );
6      o1@DefaultOutput1( c1 )
7    }
8  }
9  |
10 {
11   i2( c2 );
12   i2@B( c2 )( c2 );
13   synchronized ( token ){
14     p1@C( c2 )( c2 );
15     o1@DefaultOutput1( c2 )
16   }
17 }
```

Listing 1.12: Simple Merge — distributed

```
1  // service A
2  {
3    i1( c );
4    p1@C( c )
5  }
6  // service B
7  {
8    i2( c );
9    p1@C( c )
10 }
11 // service C
12 execution{ sequential }
13
14 {
15   p1( c );
16   o1@DefaultOutput1( c )
17 }
```

4.2 Advanced Branching Patterns

Multi-Choice ***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 derives from *Exclusive Choice*s composed with a *Parallel Split*. This implementation holds for both centralised and distributed approaches.

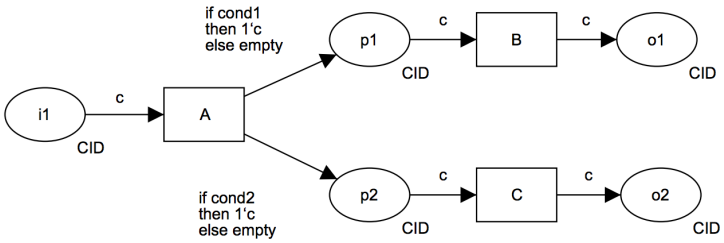

Fig. 8: Multi-Choice pattern diagram

Listing 1.13: Multi-Choice pattern — centralised

```
1  i1( c );
2  {
3    p@A( c )( cond1 );
4    if( cond1 ){
5      p1@B( c1 )( c1 );
6      o1@DefaultOutput1( c1 )
7    }
8  } | {
9    p@A( c )( cond2 );
10   if( cond2 ){
11     p2@C( c2 )( c2 );
12     o2@DefaultOutput2( c2 )
13   }
14 }
```

Listing 1.14: Multi-Choice pattern — distributed, service `A`

```
1  i1( c );
2  {
3    if( cond1 ){
4      p1@B( c )
5    }
6  }
7  |
8  {
9    if( cond2 ){
10     p2@C( c )
11   }
12 }
```

Thread Split ***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 hold for both centralised and distributed approaches, we

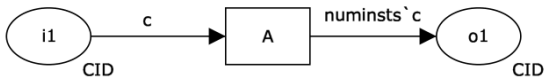

Fig. 9: Thread Split pattern diagram

provide the centralised only. *Thread Split* can be implemented in three ways: iteratively, with parallel recursion, and with the `spawn` construct.

Listing 1.15: Thread Split — iterative implementation

```
1  i1( c );
2  p1@A( c )( c );
3  for( i=0, i<numinst, i++ ){
4    o1@DefaultPort1( c )
5  }
```

Listing 1.16: Thread Split — implementation with `spawn`

```
1  i1( c );
2  p1@A( c )( c );
3  spawn( i over numinst )
4  {
5    o1@DefaultOutput1( c )
6  }
```

Listing 1.17: Thread Split — recursive implementation

```
1  define thread_split
2  {
3    {
4      if ( i < numinst ){
5        i++;
6        {
7          o1@DefaultOutput1( c )
8          | thread_split
9        }
10     }
11   }
12 }
13
14 main
15 {
16   i1( c );
17   p1@A( c )( c );
18   i=0;
19   thread_split
20 }
```

Listing 1.15 shows the iterative solution via `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 only a partial 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.

The recursive method, shown in Listing 1.17, 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.

The solution that uses the `spawn` [18] primitive offers a direct support too. Shown in Listing 1.16, the `spawn` statement creates a parallel composition of a number of processes equal to the integer evaluation of the given expression.

4.3 Advanced Synchronisation Patterns

Generalised AND-Join. 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 *Synchronization* 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, ldots, in`, ignoring additional tokens that are left in place.

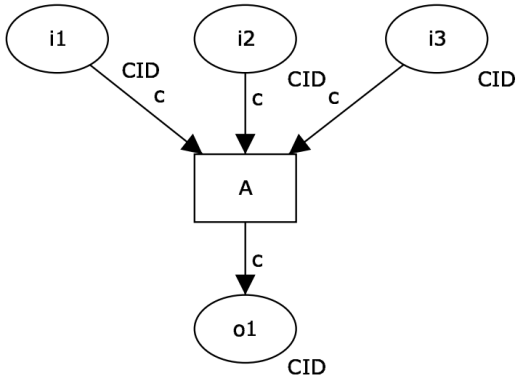

Fig. 10: Generalised AND-Join pattern diagram

Listing 1.18: Generalised AND-Join — order assumption

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

Listing 1.19: 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
  }
}
```

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[7]. In Jolie, the order of consumed messages must be coherent with the specification of execution, or the system ends in a deadlock state [14]. Listing 1.18 shows the centralised implementation of this solution, although it holds also for the distributed version. The second implementation, in Listing 1.19, fully supports the pattern's requirement 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` controls if each queue has at least one element. If enough messages arrived, the

[7] given two tuples of incoming messages $s = (i_1, \ldots, i_n)$ and $s' = (i'_1, \ldots, i'_n)$ on the same session k, if 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.

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.

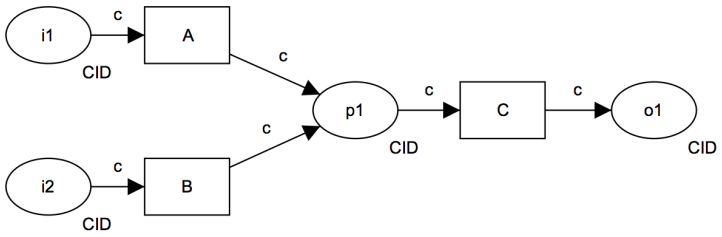

Fig. 11: Multi-Merge pattern diagram

Multi-Merge ***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 Remarkably, 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. Namely, the orchestrator removes the mutually exclusive textttsynchronized scope whilst service texttt`C` switches its execution from textttsequential to textttconcurrent in the distributed version.

Thread Merge ***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 iterative, the other that uses multiple instances. Here, we provide the solutions realised in a centralised architecture, although they hold also for the distributed one. Both solutions make use of the knowledge at design-time on the number of threads to merge (*a*). Remarkably, the employment of

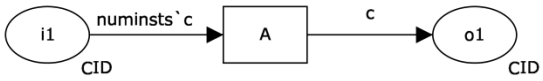

Fig. 12: Thread Merge pattern diagram

correlation sets [14] prevents non-correlated messages to be routed towards the wrong instance of the merging service, identifying the threads to coalesce (b).

Listing 1.20 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.

Similarly, the multi-instance implementation, in Listing 1.21, uses the sequential execution to receive one message per instance, storing the message in structure `c` and counting their number with variable `i`. Both `c` and `i` *alias* a *global* variable [10] to preserve the global status the system over multiple instances.

Listing 1.20: Thread Merge — iterative implementation

```
1  for( i=0, i<numinst, i++ ){
2    i1( c[ i ] )
3  };
4  p1@A( c )( c );
5  o1@DefaultOutput1( c )
```

Listing 1.21: Thread Merge — multi-instance implementation

```
1  main
2  {
3    i1( c[ i ] );
4    i++;
5    if( i == numinst ){
6      p@A( c )( c );
7      o1@DefaultOutput1( c )
8    }
9  }
```

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).

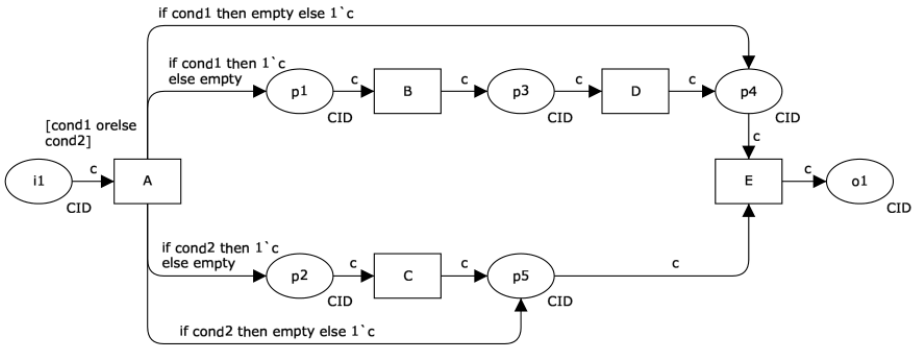

Fig. 13: Structured Synchronising Merge pattern diagram

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 [7] 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.

Both centralised and distributed implementations (respectively in Listings 1.22 and 1.23) 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 1.22: 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 )
```

Listing 1.23: 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 which 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 which 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.

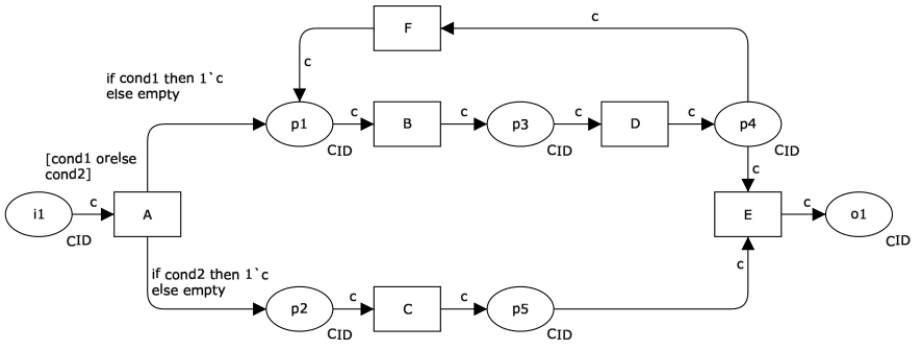

Fig. 14: General Synchronising Merge pattern diagram

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 partial 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 1.24, 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 is rendered by means of the `linkIn-linkOut` constructs that Jolie provides for implementing a token-request/token-release mechanism. The distributed version has no need for such construct 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. 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. 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 1.25. Noticeably, its realisation is equivalent to the one provided for the orchestrator.

Listing 1.24: General Synchronising Merge — centralised

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

Listing 1.25: 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 }
}
```

4.4 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 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.

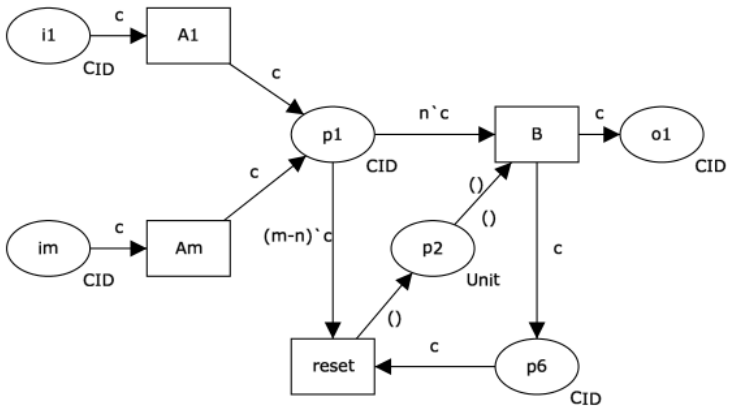

Fig. 15: Structured Partial Join pattern diagram

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 1.26, 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`. Noticeably, service `reset` is not present as its behaviour emerges from the *Synchronization* pattern. When each scope has executed, procedure `main` terminates and the master service can restart its behaviour.

The distributed solution, in Listing 1.27, derives from a *Sequence* of *Thread Merge*s and also this implementation coalesces the behaviour of service `reset` into service `B`.

Listing 1.26: Structured Partial Join — centralised

```
1  define check_and_send
2  {
3    if( i==n ){
4      p1@B( c )( c );
5      o1@DefaultPort1( c )
6    }
7  }
8
9  main
10 {
11   {
12     // code for op. i1
13     |
14     {
15       in( cn );
16       pn@An( cn )( cn );
17       synchronized( token ){
18         c[ i ] << cn;
19         i++;
20         check_and_send
21        }
22     }
23     |
24     // code for op. im
25   };
26 }
```

Listing 1.27: Structured Partial Join — distributed

```
1  //Service A1,...,Am
2  main
3  {
4    in( c );
5    p1@B( c )
6  }
7
8  //Service B
9  main
10 {
11   p1( c[ i ] );
12   for( i=1, i<n, i++ ){
13     p1( c[ i ] )
14   };
15   o1@DefaultOutput1( c );
16   for( i=0, i<m-n, i++){
17     p1()
18   }
19 }
```

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.

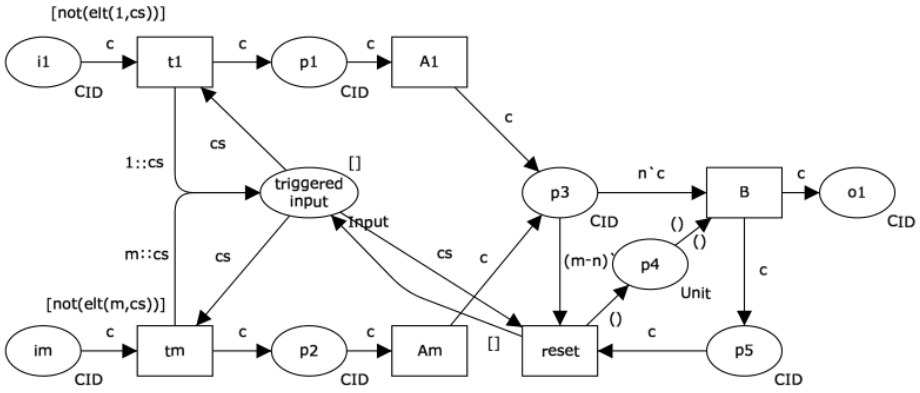

Fig. 16: Blocking Partial Join pattern diagram

Implementation We mark the support for this pattern as partial, due to its dependency from the emphGeneralised AND-Join pattern.

The centralised implementation, in Listing 1.28, 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@An` of the *Structured Partial Join* and updates the counter of the fired operation accordingly. 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 1.29, 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 1.28: Blocking Partial Join – centralised

```
1  define checkOp_in
2  {
3    if( queueSizeOp_in == 1 ||
4        ( reset_token &&
5        queueSizeOp_in > 0 ) ){
6      peekQueueOp_in;
7      pn@An( c_loc )(c[counter]);
8      counter++
9    }
10 }
11
12 define reset
13 {
14   undef( counter );
15   {
16     dequeueOp_i1
17     | // ...
18     | dequeueOp_im
19   };
20   reset_token = true;
21   {
22     checkOp_i1
23     | // ...
24     | checkOp_im
25   };
26   undef( reset_token )
27 }
28
29 define check_and_send
30 {
31   if( counter == n ) {
32     p1@B( c )( c );
33     o1@DefaultOutput1( c )
34   };
35   if( counter == m ){
36     reset;
37     check_and_send
38   }
39 }
40
41 main
42 {
43   // [i1]{ ... }
44   // ...
45   [ in( c_loc ) ]{
46     queueOp_in;
47     checkOp_in;
48     check_and_send
49   }
50   // ...
51   // [in]{ ... }
52 }
```

Listing 1.29: Blocking Partial Join — distributed

```
1  // services T1,...,Tm
2  main
3  {
4    [ in( c ) ]{
5      queueOp_in;
6      if( queueSizeOp_in == 1 ){
7        p1n@An( c )
8      }
9    }
10   [ reset() ]{
11     dequeueOp_in;
12     if( queueSizeOp_in > 0 ){
13       peekQueueOp_in;
14       p1n@An( c )
15     }
16   }
17 }
18
19 // service B
20 main
21 {
22   p3( c[ 0 ] );
23   for( i = 1, i < n, i++ ){
24     p3( c[ i ] )
25   };
26   o1@DefaultOutput1( c );
27   for( i = 0, i < m-n, i++ ){
28     p3()
29   };
30   {
31     reset@T1()
32     | // ...
33     | reset@Tm()
34   }
35 }
```

Cancelling Partial Join. 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.

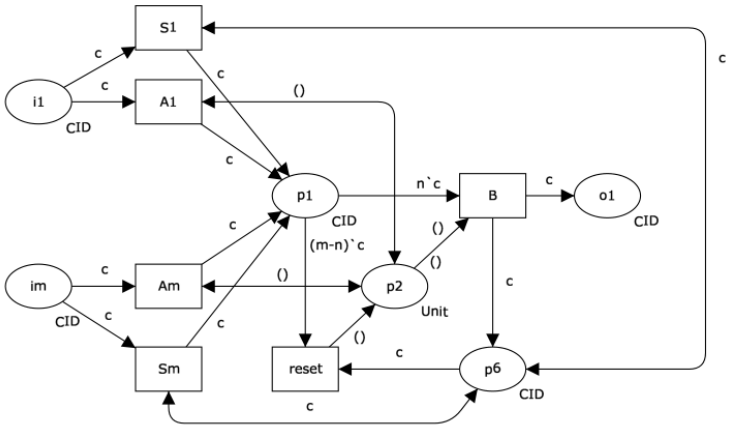

Fig. 17: Cancelling Partial Join pattern diagram

Listing 1.30: Cancelling Partial Join — orchestrator

```
1  define check_and_send
2  {
3    if( counter == n ){
4      p1@B( c )( c );
5      o1@DefaultPort1( c );
6      skip = true
7    }
8  }
9
10 main
11 {
12   {
13     // ...
14     |
15     {
16       in( cn );
17       synchronized( token ){
18         if( skip ){
19           p2n@Sn( cn )( c2 )
20         } else {
21           p1n@An( cn )( cn );
22           c[ i ] << cn;
23           counter++;
24           check_and_send
25         }
26       }
27     }
28     |
29     //...
30   };
31   undef( skip )
32 }
```

Listing 1.31: Cancelling Partial Join — distributed SOA

```
1  // Services T1,...,Tm
2  main
3  {
4    i1( c );
5    p3@B( c )( skip );
6    if ( skip ){
7      p21@S1( c )
8    } else {
9      p11@A1( c )
10   }
11 }
12
13 //service B
14 main
15 {
16   [ p1( c[ 0 ] )]{
17     for( i = 1, i < n, i++ ){
18       p1( c[ i ] )
19     };
20     o1@DefaultOutput1( c );
21     for( i = 0, i < m-n, i++){
22       p1()
23     };
24     undef( skip.( SID ) )
25   }
26   [ p3( c )( response ){
27     response = false;
28     local_skip -> skip.( SID );
29     synchronized( local_skip ){
30       local_skip++;
31       if ( local_skip > n ){
32         response = true
33       }
34     }
35   }]{ nullProcess }
36 }
```

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

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 easily 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.

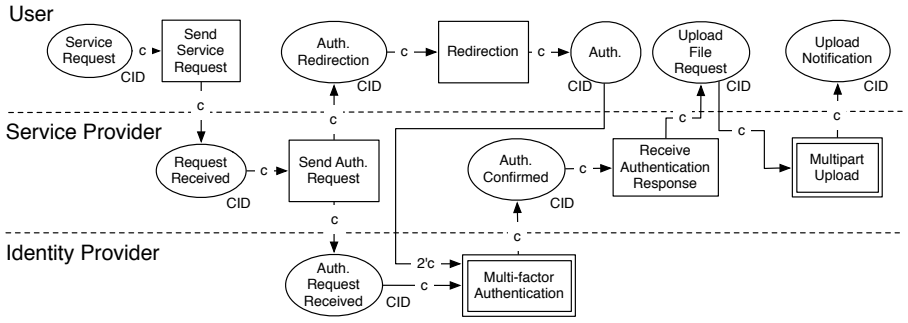

Fig. 18: The Upload Service net

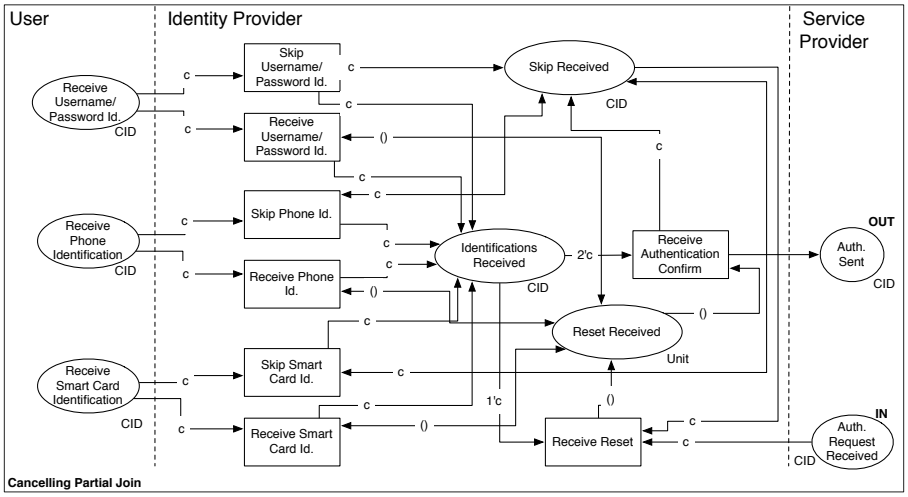

Fig. 19: Multi-factor Authentication subnet

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. Listing 1.32 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.

Listing 1.32: Multi-factor Authentication — Orchestrator

```
1  execution{ sequential }
2  constants { n = 2 }
3  init { receivedAuth -> global.receivedAuth }
4
5  define check_and_send
6  {
7    if( receivedAuth==n ){
8      receiveIds@ReceiveAuthenticationConfirm( c )( c );
9      sendAuthentication@OUT( c );
10     skip = true
11   }
12 }
13
14 main
15 {
16   authenticationRedirectReceived( request );
17   {
18     {
19       sentUP( upData );
20       if( skip ){
21         sendUP@SkipUP( c1 )( c1 )
22       } else {
23         sendUP@ReceiveUP( c1 )( c1 );
24         c[ receivedAuth ] << c1;
25         receivedAuth++;
26         check_and_send
27       }
28     }
29     |
30     {
31       sentPhone( phoneData );
32       if( skip ){
33         sendPhone@SkipPhone( c2 )( c2 )
34       } else {
35         sendPhone@ReceivePhone( c2 )( c2 );
36         c[ receivedAuth ] << c2;
37         receivedAuth++;
38         check_and_send
39       }
40     }
41     |
42     {
43       sentSIM( simData );
44       if( skip ){
45         sendSIM@SkipSIM( c3 )( c3 )
46       } else {
47         sendSIM@ReceiveSIM( c3 )( c3 );
48         c[ receivedAuth ] << c3;
49         receivedAuth++;
50         check_and_send
51       }
52     }
53   };
54   undef( skip );
55   receivedAuth=0;
56 }
```

The User-controlled part of the interaction mixes centralised and distributed WPs. Listing 1.33 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 which passes it to service `CreateChunks` (Line 3). The service employs a centralised *Thread Split* (A) to split the file into `n` chunks. Then the orchestrator implements a centralised *Thread Merge* (B) to collect triplets of chunks and send them to service `SendChunks` (Lines 5-7). Notably, 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). At Service Provider's side the service `StoreChunks` employs a centralised *Generalised AND-Join* (C) to receive the chunks (Listing 1.34 Lines 1-13). When the `n`th chunk reaches the service, it passes the thread of control with a distributed *Sequence* to service `ComposeFile` (Listing 1.35) which employs a centralised *Thread Merge* (D) 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.

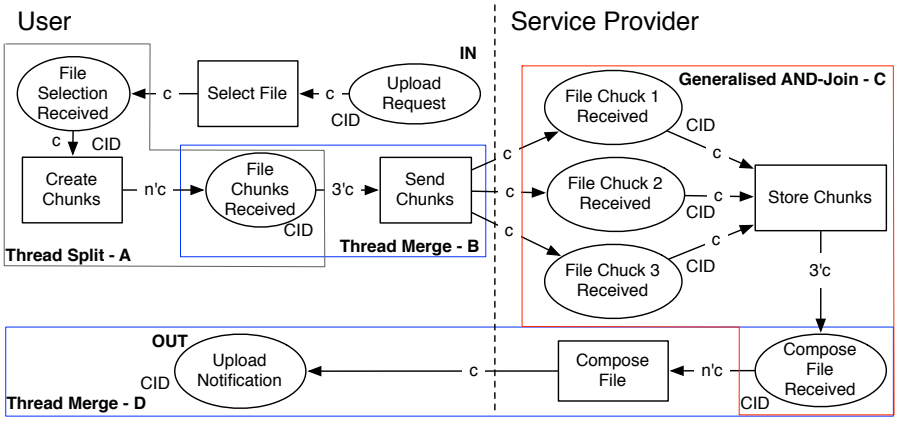

Fig. 20: Multipart Upload subnet

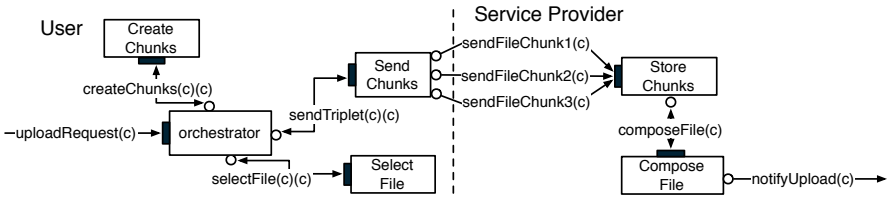

Fig. 21: The architectural view of Multipart Upload in Fig. 20

Listing 1.33: 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)
}
```

Listing 1.34: Multipart Upload, StoreChunks

```
execution{ sequential }

define check_and_send
{
  size@QueueUtils( queue1 )( chunk1_count );
  size@QueueUtils( queue2 )( chunk2_count );
  size@QueueUtils( queue3 )( chunk3_count );

  if( chunk1_count > 0 && chunk2_count > 0 && chunk3_count > 0 ){
    //Take c1, c2, and c3
    poll@QueueUtils( queue1 )( chunks.c1 );
    poll@QueueUtils( queue2 )( chunks.c2 );
    poll@QueueUtils( queue3 )( chunks.c3 );
    // and send them to ComposeFile
    composeFile@ComposeFile( chunks )
  }
}

main
{
  [ sendFileChunk1( c ) ]{
      qer.queue_name = queue1;
      qer.element << c;
      push@QueueUtils( qer )();
      check_and_send
  }

  [ sendFileChunk2( c ) ]{
      qer.queue_name = queue2;
      qer.element << c;
      push@QueueUtils( qer )();
      check_and_send
  }

  [ sendFileChunk3( c ) ]{
      qer.queue_name = queue3;
      qer.element << c;
      push@QueueUtils( qer )();
      check_and_send
  }
}
```

Listing 1.35: Multipart Upload, ComposeFile

```
1  constants { chunksNumber = n, chunkThreads = 3 }
2
3  define storeChunks
4  {
5    fileChunks[ #fileChunks ] = c.c1;
6    fileChunks[ #fileChunks ] = c.c2;
7    fileChunks[ #fileChunks ] = c.c3
8  }
9
10 main
11 {
12   for( recChunks=0, recChunks < chunksNumber, recChunks+=chunkThreads ){
13     composeFile( c );
14     storeChunks
15   };
16   receiveUploadNotification@User( c )
17 }
```

6 Conclusions

Workflow Patterns	Jolie Support	Supported by and 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, `sequential` execution
Multi-Choice	+	Parallel Split, Exclusive Choice	
Thread Split	+	iteration, recursion and Parallel Split, `spawn`	
Generalised AND-Join	+/–	Synchronization, input choice and ad-hoc queues	
Multi-Merge	+	Synchronization	Simple Merge, `concurrent` execution
Thread Merge	+	iteration, multi-instances	
Structured/Local Synchronizing Merge	+	Multi-Choice, Synchronization	
Generalised Synchronizing Merge	+/–	Structured Synchronizing Merge	
Structured Partial Join	+	Synchronization, Thread Merge	Thread Merge, Sequence
Blocking Partial Join	+/–	Generalised AND-Join, Structured Partial Join	
Cancelling Partial Join	+	Structured Partial Join	

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

Contributions of this work are: (*i*) the definition of a methodology for translating CPN-modelled SOAs into composable and executable ones. (*ii*) The creationof a collection of implemented Workflow Patterns (reported in [19]). 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 "non direct" support, i.e., the translation of the CPN of the pattern does not completely follow the rules described in § 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. Note that we report both the centralised and distributed implementations which, as expected, in some cases vary. As shown in Table 1 we can conclude that Jolie allows to implement most WPs.

6.1 Related Work

A close concept to Workflow Pattern is that of *service interaction pattern*, introduced in [20]. Service interaction patterns 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 [21] and static analysis [22]. An inspiring work which considers a direct translation from Petri nets to a service-oriented language (Abstract BPEL) is [23]. 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 WS-BPEL [24] and of BPML [25].

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 [26] 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-instance*s, *state*, *cancellation*, *completion*, *termination*, and *triggering* patterns.

References

1. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.
2. OMG, "Service oriented architecture modeling language (SoaML)," 2009.
3. OASIS, "Reference architecture foundation for SOA version 1.0," December 2012.
4. P. Mayer, N. Koch, and A. Schroeder, "The UML4SOA Profile," tech. rep., Ludwig-Maximilians-Universitaet Muenchen, July 2009.
5. K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
6. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distrib. Parallel Databases*, vol. 14, pp. 5–51, July 2003.
7. N. Russell, A. H. M. T. Hofstede, and N. Mulyar, "Workflow control-flow patterns: A revised view," tech. rep., 2006.
8. OASIS, "Web Services Business Process Execution Language." `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.
9. "Jolie Website." `http://www.jolie-lang.org/`.
10. F. Montesi, C. Guidi, and G. Zavattaro, *Service Oriented Programming with Jolie*, vol. 1 of *Web Services Foundations*.
11. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "Sock: A calculus for service oriented computing," in *ICSOC*, LNCS, pp. 327–338, Springer, 2006.
12. W. Reisig, *Petri Nets: An Introduction*, vol. 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.
13. W. M. Coalition, *Workflow Management Coalition Terminology & Glossary*. Workflow Management Coalition, 1999.
14. F. Montesi and M. Carbone, "Programming services with correlation sets," in *ICSOC*, pp. 125–141, 2011.
15. W3C WS-CDL Working Group, "Web services choreography description language version 1.0." http://www.w3.org/TR/ws-cdl-10/, 2004.
16. M. Carbone and F. Montesi, "Deadlock freedom by design: multiparty asynchronous global programming," *SIGPLAN Not.*, vol. 48, pp. 263–274, Jan. 2013.
17. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the gap between interaction- and process-oriented choreographies," in *SEFM'08*, IEEE.
18. F. Montesi, "Jolie: a service-oriented programming language," 2010.
19. M. Gabbrielli, S. Giallorenzo, and F. Montesi, "Executable design patterns for building service-oriented architectures." `http://www.cs.unibo.it/projects/jolie/executable_design_patterns_full.pdf`.
20. A. Barros, M. Dumas, and A. H. M. T. Hofstede, "Service interaction patterns," in *In Proc. of the ICBPM*, pp. 302–318, Springer Verlag, 2005.
21. J. Mendes, P. Leitao, F. Restivo, and A. Colombo, "Composition of petri nets models in service-oriented industrial automation," in *INDIN'10*, pp. 578–583, 2010.
22. N. Lohmann, O. Kopp, F. Leymann, and W. Reisig, "Analyzing bpel4chor: verification and participant synthesis," WS-FM'07, pp. 46–60, Springer-Verlag, 2008.
23. N. Lohmann and J. Kleine, "Fully-automatic translation of open workflow net models into simple abstract bpel processes," in *In Modellierung*, pp. 57–72, 2008.
24. P. Wohed *et al.*, "Analysis of web services composition languages: The case of bpel4ws," in *Proc. of ER, LNCS 2813*, pp. 200–215, Springer Verlag, 2003.
25. W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and P. Wohed, "Pattern-based analysis of BPML (and WSCI).," *FIT-TR-2002-05*, 2002.
26. M. Gabbrielli, S. Giallorenzo, and C. Guidi, "Towards a composition-based API-aaS layer," technical report, University of Bologna/INRIA, 2014. `http://www.cs.unibo.it/projects/jolie/apiaas_technical_report.pdf`.