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

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.

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 it is easy to understand the interactions of a CPN model, 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.

[3] here referred as Workflow Patterns for simplicity.

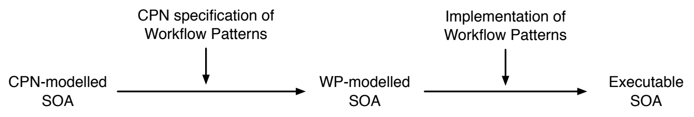
CPN specification of Workflow Patterns
Implementation of Workflow Patterns
CPN-modelled SOA
WP-modelled SOA
Executable SOA

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 track the whole system. On the other hand, some scenarios strictly require a distributed approach, e.g., an interaction that comprehends different parties. In § 3 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 § 2. 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. For reasons of presentation we delegate to the full paper [12] the analysis and the implementation of all considered WPs (*basic* and *advanced branching and synchronisation* patterns).

2 From Coloured Petri nets to Jolie SOAs

Background. We assume the reader is familiar with the basic concepts of Petri nets on which Coloured Petri nets add expressions on arcs and types to tokens[4]. In order to understand our translation procedure we briefly describe the message-passing operations in Jolie. The language supports two kinds of message-passing

[4] [12] reports a comprehensive introduction on Coloured Petri nets.

operations which send or receive the content of a variable v. Sending operations specify the location L the message is sent to. *One-Way* operations (OWs) send or receive a message and immediately pass the thread of control to the subsequent activity in the process. The syntax of a receiving OW is simply $op\_name(v)$, whilst a sending OW adds the location of the request $op\_name@L(v)$. *Request-Response* operations (RRs) either send a request and keep the thread of control until they receive a response or they receive a message, do some computation, and send a response. The syntax of a sending RR is $op\_name@L(v)(v)$, whilst a receiving RR adds a pair of brackets that enclose the code run before sending the response to the invoker $op\_name(v)(v)\{code\}$. Jolie provides also an input choice with syntax $[\eta_1]\{B_1\} \ldots [\eta_n]\{B_n\}$. When an input operation η_i is triggered it disables all $\{\eta_1, \ldots, \eta_n\} \backslash n_i$ and executes. Then B_i executes.

Workflow Patterns in Jolie. In this section we present a procedure for translating a CPN-modelled Workflow Pattern into an executable Jolie SOA. The translation follows 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 a WP composed by several services running in parallel. Following these principles, we translate CPN models of Workflow Patterns into Jolie SOAs as follows. We map input, internal, and output places into One-Way (OW) operations (principle *ii*). 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 locations `DefaultOutput1,...`. This enables composition of the implemented patterns by binding their locations. As exposed in § 1 we translate both the centralised and the distributed versions of WPs. In the centralised approach a master service, called *orchestrator* encodes the whole behaviour of a WP coordinating the interactions among the services participating to the SOA. BPEL [8] is the most known technology for this approach, called *orchestration*. By convention the orchestrator of a WP is the only service that receives and sends messages outside the SOA. The distributed approach recalls that of *choreography* languages like WS-CDL [13]. Recent works [14,15] introduced automatic techniques to project executable services of an SOA from a choreographic specification. Following a similar approach in the distributed version of a WP we maintain a direct relation between transitions and services, imposing no restrictions on the scope of external input and output operations. Fig. 2 reports an example of a CPN in box **A** and informally depicts the architectural view of the translated centralised (**B**) and distributed (**C**) implementations. Listing 1.1 reports the corresponding code of, respectively, the orchestrator of the centralised version and of the services in the distributed one. In **B** the orchestrator (`O`) receives the input message as a OW operation `i1` (Line 1). `O` sends via RR the content `c` to service `A` which returns the condition `cond1` (Line 2). Based on `cond1`, `O` either redirects a request to `B` or `C` (Lines 4-8). `O` sends `c` to `D`, which returns its response. (Line 9) Finally `O` sends `c` as output to the `DefaultOutput1`. (Line 10). Notably, `O` uses RRs to invoke operations on the services its composes, waiting for their responses. **C** maintains a direct relation between transitions and services which pass the thread of control using OW operations. Service `A` receives the input message `i1` (Line 1), evaluates condition `cond1` and redirects `c` to either service `B` (Line 3) or `C` (Line 4) which, in turn, forwards `c` to service `D`. Finally `D` receives `c` on operation `p3` and sends it to the output location `DefaultOutput1` (Line 10).

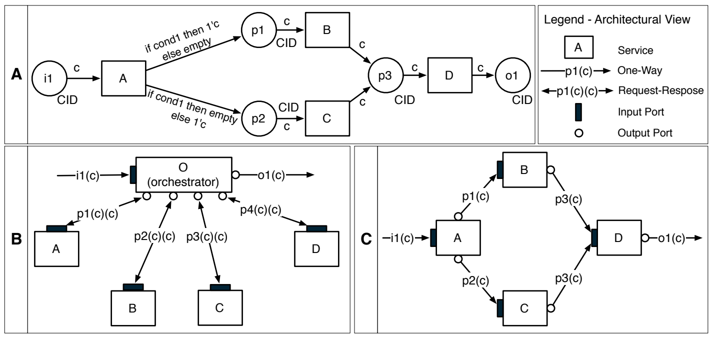

Fig. 2: (**A**) a CPN model, its centralised (**B**), and distributed (**C**) realisations.

Listing 1.1: 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 )
```

3 The Upload Service Use Case

This section shows how a realistic SOA, modelled as a Coloured Petri net, translates into an executable SOA. The goal of this section is twofold. (*i*) We show the benefits of CPNs as SOA design tool. CPNs let the designer model the system focusing on interactions, whilst the control on message flow is left to the developer. (*ii*) We exhibit how a developer can easily map a CPN into a composition of WPs. With just the flow of interactions as model, the developer can mix distributed and centralised implementations of patterns and build the system.

The use case describes a typical interaction between a *User*, a file upload *Service Provider*, and an *Identity Provider*. In particular the Identity Provider offers an OpenID-like [16] authentication with a multi-factor mechanism [17] whilst the Service Provider offers a multipart upload procedure. Fig. 3 depicts

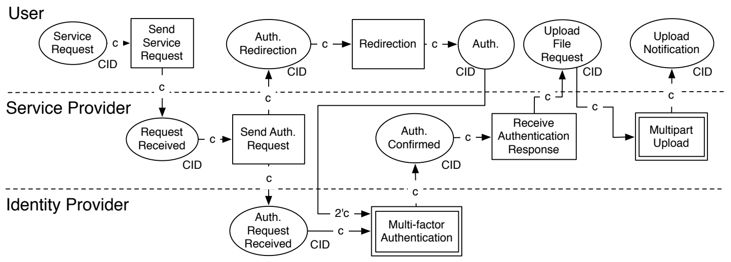

Fig. 3: The CPN model of the interactions in the Upload Service SOA.

the CPN model of the use case. The two double-line bordered boxes in the figure act as placeholders for the two subnets related to the multi-factor authentication and the Multipart Upload procedure, the latter modelled in Fig. 4.

Let us identify the WPs[5] that compose the CPN-modelled SOA. Interaction starts from the User that requests the service. This basic interaction is a distributed *Sequence* that passes the thread of control to the Service Provider. The Service Provider employs a distributed *Parallel Split* to start the multi-factor authentication and redirects the authentication request to the User. The Identity Provider allows users to identify themselves with a multi-factor authentication. Let us suppose that the Identity Provider demands a minimum number of different authentication procedures. Such *M-out-of-N* mechanism maps to the centralised version of the *Cancelling Partial Join* pattern being completely controlled by the Identity Provider. For reasons of presentation we choose not to discuss about the implementation of this pattern as it maps directly and does not show interesting interactions with other patterns. For the model and the code relative to the multi-factor authentication refer to [12]. 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. 4 depicts such interactions and highlights the most relevant WPs. Fig. 5 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 1.2 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

[5] [12] reports the description of the considered patterns and their implementations.

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 1.3 Lines 1-13). When the `n`th chunk reaches the service, it passes the thread of control with a distributed *Sequence* to service `ComposeFile` 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.

Listing 1.2: User's side

```
 1  // orchestrator
 2  uploadRequest(c);
 3  selectFile@SelectFile(c)(c);
 4  createChunks@CreateChunks(c)(c);
 5  for( i=0, i<#c, i++ ){
 6    r.c1=c[i++];r.c2=c[i++];r.c3=c[i];
 7    sendTriplet@SendChunks(r)()
 8  }
 9  // SendChunks
10  sendTriplet(c)(){
11    sendFileChunk1@StoreChunk(c.c1)
12    | sendFileChunk2@StoreChunk(c.c2)
13    | sendFileChunk3@StoreChunk(c.c3)
14  }
```

Listing 1.3: Service Provider's side

```
 1  // StoreChunks
 2  [ sendFileChunk1(c) ]{
 3    // store file chunk on queue 1
 4    // check for upload completion
 5  }
 6  [ sendFileChunk2(c) ]{
 7    // store file chunk on queue 2
 8    // check for upload completion
 9  }
10  [ sendFileChunk3(c) ]{
11    // store file chunk on queue 3
12    // check for upload completion
13  }
```

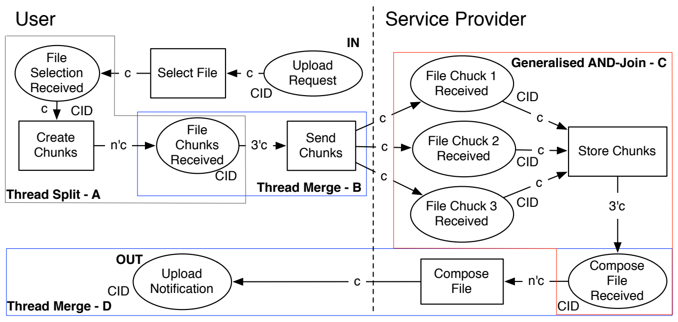

Fig. 4: Multipart Upload modelled as composition of Workflow Patterns

4 Conclusions

Contributions of this work are: (i) the definition of a methodology for translating CPN-modelled SOAs into composable and executable ones and (ii) the creation of a collection of implemented Workflow Patterns (reported in [12]). Such implementations follow both a centralised and a distributed approach to

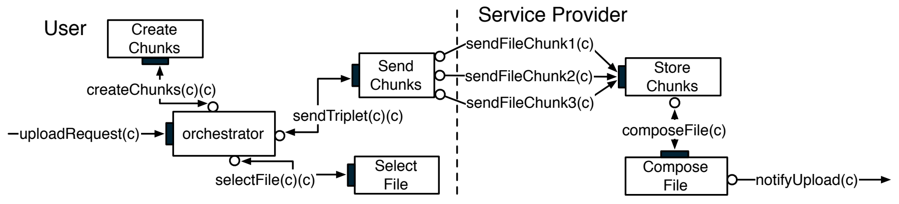

Fig. 5: The architectural view of Multipart Upload in Fig. 4

allow developers the flexibility to choose and mix them. A realistic use case proves our claim that the patterns obtained in this way can be 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 § 2 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.

Related Work. A close concept to Workflow Pattern is that of *service inter-*

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: Support of *basic* and *advanced branching and synchronisation* WPs in Jolie.

action pattern, introduced in [18]. 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 [19] and static analysis [20].

Finally WPI used WPs as a tool to evaluate the expressive power of business process languages, in particular for the cases of BPEL [21] and BPML [22].
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 investigate the remaining patterns described by the WPI and to use the implemented WP to offer pattern composition as APIs [23].

References

1. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. PH, 2005.
2. OMG, "Service oriented architecture Modeling Language," 2009.
3. OASIS, "Reference architecture foundation for SOA version 1.0," December 2012.
4. P. Mayer, N. Koch, and A. Schroeder, "The UML4SOA Profile," 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, "BPEL." `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. M. Gabbrielli, S. Giallorenzo, and F. Montesi, "Executable SOAs exploiting workflow patterns," tech. rep. `http://www.cs.unibo.it/projects/jolie/eSOAs.pdf`.
13. W3C WS-CDL Working Group, "Web services choreography description language version 1.0." http://www.w3.org/TR/ws-cdl-10/, 2004.
14. M. Carbone and F. Montesi, "Deadlock freedom by design: multiparty asynchronous global programming," *SIGPLAN Not.*, vol. 48, pp. 263–274, Jan. 2013.
15. M. Dalla Preda, I. Lanese, J. Mauro, M. Gabbrielli, and S. Giallorenzo, "Deadlock freedom by construction for distributed adaptative applications," tech. rep., `http://www.cs.unibo.it/projects/jolie/aioc.pdf`.
16. OpenID, "Specifications." `http://openid.net/developers/specs/`.
17. "Multi-factor authentication." `http://aws.amazon.com/iam/details/mfa/`.
18. 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.
19. 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.
20. N. Lohmann, O. Kopp, F. Leymann, and W. Reisig, "Analyzing bpel4chor: verification and participant synthesis," WS-FM'07, pp. 46–60, Springer-Verlag, 2008.
21. 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.
22. 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.
23. C. Guidi, S. Giallorenzo, and M. Gabbrielli, "Towards a composition-based APIaaS layer," in *CLOSER*, p. to appear in, SciTePress, 2014.