Journal of Network and Computer Applications 36 (2013) 1102–1115

Contents lists available at SciVerse ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Model-driven approach supporting formal verification for web service composition protocols

C. Dumez [a], M. Bakhouya [b], J. Gaber [a], M. Wack [a], P. Lorenz [c,*]

[a] University of Technology of Belfort-Monbeliard, 90000 Belfort Cedex, France
[b] Aalto University, Otakaari 4, FIN-00076 Aalto, Finland
[c] University of Haute Alsace, IUT, 34 rue du Grillenbreit, 68008 Colmar, France

ARTICLE INFO

Article history:
Received 23 April 2012
Received in revised form
12 December 2012
Accepted 13 January 2013
Available online 10 February 2013

Keywords:
Web service composition
Model-driven architecture
Workflow patterns
LOTOS
Formal verification

ABSTRACT

Composite Web services development is a complex task involving specification, verification, implementation, and testing. Despite the fact that several languages have been proposed for composing Web services (e.g., BPEL, WSCI), there is a lack of well-defined formal semantics for formal analysis and verification. Moreover, current approaches are specific to a given programming language (e.g. BPEL) and they focus only on the verification of already implemented composite services. This paper proposes an approach for specifying, verifying and implementing composite services according to the Model-Driven Architecture principles. It makes use of formal methods, especially the LOTOS formal description language, to support composition verification at specification time. The benefit is that the composition specification is proven to be correct before its implementation with a programming language such as BPEL. A case study is also presented to show how a service composition can be specified in a workflow and then formally verified before executable code generation.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Web services are currently used by organizations to share their knowledge over the network and facilitate Business-to-Business (B2B) collaboration. Services are independent software components realizing specific tasks, which can communicate with each other by exchanging messages. This process, called service *composition*, usually results in the creation of a new *composite* service that can be defined as the aggregation of other elementary or composite services (Gaber and Bakhouya, 2006; Bakhouya and Gaber, 2008).

The composition of independent existing Web services is a difficult and time consuming task. Programming languages, such as BPEL (OASIS, 2007), are gaining popularity and overall acceptance by the software industry. However, these languages are meant for the implementation and execution rather than providing a visual or formal representation of the composition. This lack of consideration to the specification and verification stages decreases the global understanding of the composition and makes the application harder to understand and therefore to maintain. In the past few years, the research community has been trying to tackle this issue by proposing model-driven approaches based on formal methods (ter Beek et al., 2006). The downside of these approaches is that the composite service is not verified in the early design process. This results in increased development time and costs when errors are detected late in the development cycle (Achilleos et al., 2008).

In this paper, an approach according to the principles of Model-Driven Architecture (MDA) is proposed for specifying, verifying and implementing composite services. MDA (Soley, 2000) focuses on creating abstract models rather than computing or algorithmic concepts. It uses models that make sense from the user's point of view and that are precise enough to serve as a basis for implementation. In the context of service composition, examples of such models include BPMN (White, 2004a) and UML activity diagram (OMG, 2009). These models are de facto standards for Business Process Modeling (BPM), but their lack of clearly defined formal semantics does not allow formal analysis and verification.

For this purpose, the solution introduced in this paper embraces formal methods, especially LOTOS formal specification language, in order to prove that a service composition is correct at design time. This is an important step towards reliable service composition, since problems could be detected early in the development cycle, before even starting the implementation. To achieve this objective, the following methodology composed of five phases is considered:

- Selection from a service repository, such as UDDI, of the services required for the service composition.

* Corresponding author. Tel.: +33 632630204.
E-mail addresses: dchris@gmail.com (C. Dumez), mohamed.bakhouya@aalto.fi (M. Bakhouya), gaber@utbm.fr (J. Gaber), maxime.wack@utbm.fr (M. Wack), lorenz@ieee.org, pascal.lorenz@uha.fr (P. Lorenz).

1084-8045/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jnca.2013.01.009

- Elaboration of a specification model of the composition using business process modeling languages, such as UML and BPMN. We have adapted UML activity diagrams to model the interactions between the services in a workflow. These business process modeling languages are well-known in the software designers community and are able to model all workflow control-flow patterns supported by popular composition languages, such as BPEL.
- Automated translation of the specification model into a formal description. Any formal description language may be used as long as it is supported by automated verification tools. We have selected LOTOS because it provides constructs that are adequate to specify composition owing to their compositionality properties, such as sequential, parallel, and conditional execution constructs and it is supported by a powerful verification tool, CADP. A mapping method by translating workflow models into LOTOS formal specification language is also proposed.
- Automated verification of desired behavioral properties using CADP.
- Generation of BPEL code from the business process modeling languages as recommended by MDA. For example, BPMN is currently supported by more than 60 tools for BPMN to BPEL translation. In this work, a tool has been developed to allow the translation of UML activity diagrams (UML-AD) to BPEL. Once the composite Web service is implemented, the last step is usually to publish it into a service registry so that it can be reused later.

More precisely, the objective is to be able to generate composite services proven to be correct, according to MDA principles. The advantage is that the composite service is verified in the early design process before code generation. Moreover, we put emphasis in this paper on the control-flow patterns that are supported by BPEL, which is the most common language to build service compositions.

The remainder of this paper is structured as follows. Section 2 provides a state-of-the-art review of service composition approaches involving formal methods. In Section 3, the model-driven methodology is presented together with the formal specification of basic workflow patterns using LOTOS. In Section 4, a composition scenario is modeled using UML, translated into LOTOS and then verified using the CADP tool set before generating the corresponding BPEL code. Finally, Section 5 draws the conclusions and presents future work.

2. Related work

This section focuses on service composition approaches involving formal methods. Two research directions are identified: *language-oriented approaches* and *model-oriented approaches*. The former ones focus on the composite services verification *afterwards*. These services are expressed in a particular business process modeling language, such as BPEL or WSCI. For example, one takes a BPEL code, translates it into a formal description, and verify the desired properties. The main formalisms used for reasoning about the service composition are Petri nets, state transition systems (STSs), and process algebras.

In the second research direction, the designer uses graphical representations, such as BPMN or UML, to specify the service composition before generating the corresponding executable code with BPEL for example. These business process modeling languages are widely adopted or service composition, because of the design features they provide (support most of control-flow patterns) and being comprehensive enough to generate executable codes. However, it is hard for designers to deal with formal verification of service composition at design time because neither BPMN nor UML are supported with suitable tools. Therefore, research direction towards modeling methods that allow designers to specify and verify the composition during the early design phase are required, thus avoiding iterative cycle between the implementation and the specification. The rest of this section presents brief descriptions of some approaches proposed in the literature. Furthermore, a classification and a comparison of composition approaches is developed in Appendix A.

2.1. Language-oriented approaches

Several approaches that belong to the first research direction have been proposed in the literature. For example, Tan et al. (2009) propose an approach to analyze the compatibility of two services by translating their BPEL abstract processes (i.e., description) into CPN (Colored Petri Net) and check if their composition violates the constraints imposed by either side. Service/Resource Net (SRN), introduced by Tang et al. (2004), is an extended Petri net-based model with some new elements, such as time, resource taxonomy, and conditions to model Web service composition. Once the composition is modeled using a SRN, it can be analyzed and evaluated using PN analysis methods. Finally, the authors discuss the development of a framework to generate BPEL code from their model.

Alike Petri nets, state transition systems, such as finite state automata, Input/Output (I/O), and timed automata, have been used to describe and verify Web services composition. For example, an approach was proposed by Pu et al. (2006) by providing a mapping between a subset of BPEL activities and timed automata in order to use the UPPAAL model checker. Mitra et al. (2007) propose to describe the goal and component services as I/O automata whose states represent the configuration of the services, while the transitions define the way in which the services evolve from one configuration to another. This shifts the problem from verifying the existence of the goal service to a simulation problem over automata. Diaz et al. have proposed an approach to automatically translate Web services with time restrictions into timed automata and use UPPAAL tool to simulate and verify the system behavior (Diaz et al., 2006).

Foster et al. (2006) proposed an approach for modeling and verifying services composition using the Finite State Processes (FSP). In their work, the BPEL specification is modeled in UML, in the form of message sequence charts, and then translated into the FSP notations. FSP notation is then mapped to LTS (Labelled Transition System) to reason and verify its correctness, such as safety and liveness properties. Another approach called COCOA was proposed by Ben Mokhtar et al. (2006). COCOA is a composition framework which supports complex behaviors for both services and tasks (i.e. user requests) described as OWL-S (Martin et al., 2004) processes. In Ravn et al. (2011), the authors propose an approach that allows manually translating WS-Business Activity (mainly WS-BA standard) specified by state-transition tables to model checker UPPAAL and verifies its correctness, mainly its correctness and its boundedness.

Process algebras, such as CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), LOTOS (Language of Temporal Ordering Specification) and π-calculus, have been used for verifying service composition. This is due to their high expressiveness together with constructs that are adequate to easily specify service composition. In other words, they provide constructs with compositionality properties, such as sequential, parallel, and conditional execution constructs (Bordeaux and Salaun, 2005; Beek et al., 2007). For example, in Li et al. (2007), rules to map each syntactic construct of WS-CDL to its corresponding CSP process are given. In Camara et al. (2006), the authors propose rules to translate Web service

choreographies written in WSCI into CCS and then check whether two or more Web services are compatible or not, i.e., all exchanged messages are mutually understood, and that their communication is deadlock-free. In Peng et al. (2009), the authors propose a mapping approach to translate WS-CDL choreographies into models for model checker SPIN to verify properties about channel passing.

In Lucchi and Mazzara (2007), the semantics of BPEL are specified using π-calculus. Because BPEL lacks formal semantics, the authors formally describe a BPEL subset, especially the specification of basic activities, structured activities and fault handling. This allows designers to formally reason about orchestration processes. In Camara et al. (2006) and Mateescu et al. (2008) a methodology for the generation of adaptors capable of solving behavioral mismatches between services interface descriptions was presented. The composite service is supposed to be given in ABPEL and the description of the orchestration conversation is available. A tool was developed to generate STSs from service interfaces that can be translated to LOTOS and then composition issues can be found such as unmatching number of parameters, different ordering, etc before the generation of BPEL model. It should be noted that the objective is to detect the behavioral mismatches between services interface during the service invocation and it does not deal with service composition at design time.

2.2. Model-oriented approaches

As described in the previous section, several works towards the first research direction have been done to specify WS composition using formal methods. However, these approaches are used too late in the development process because their aim is to reason on services already implemented using for example BPEL. Little work has been done, to the best of our knowledge, into the second research direction in order to specify and verify the service during the early design phase, thus avoiding iterative cycle between the implementation and the specification. For example, an approach that combines MDA and Petri Net is proposed in Achilleos et al. (2008) to provide the design, the verification and the code generation process. The main advantage is that a service is verified early on and prior its implementation. In Vaz and Ferreira (2008), a translation of a collection of workflow patterns were translated into Promela, which is the input specification language of SPIN verification tool (Holzmann and et al., 1997). In this tool, systems are described in Promela and the properties to be verified are expressed as Linear Temporal Logic (LTL) formulas. The main advantages of this approach is that Promela's C-like syntax makes it more accessible to non-experts and SPIN is a model checker that allows the automatic verification of business processes.

BPMN (White, 2004a) and UML (OMG, 2009) are graphical modeling languages, fairly easy to understand while being comprehensive enough to generate executable code from them. For example, Wohed et al. (2006) show that BPMN provides direct support for the majority of the control-flow patterns. BPMN is currently supported by over 60 tools.[1] Gardner et al. presented, in Amsden et al. (2003), a UML profile for modeling automated business processes as well as a mapping to BPEL 1.0. An adaptation of this profile for BPEL 1.1 was then implemented by Mantel in ws-uml2bpel (Mantell, 2003). Dumez et al. proposed a UML 2.0 profile called UML-S (Dumez et al., 2008a,b,c) together with translation rules where models are detailed enough to allow automatic executable code generation with WS-BPEL 2.0. A framework has also been developed in order to model the composition using UML-S diagram and automatically generate the corresponding BPEL code (Dumez et al., 2009).

The work presented in this paper also falls into the second research direction by proposing a model-driven methodology that integrates both business process modeling (i.e., UML activity diagram) with formal methods, mainly process algebra, LOTOS (Brinksma, 1988). Business process modeling supports the service design, specification, and implementation stages while LOTOS with its verification features and constructs that are more adequate to easily model service composition, is used for service verification. The main aim of this methodology is to allow designers to specify and verify the composition during the early design phase are required, thus avoiding iterative cycle between the implementation and the specification.

3. Proposed methodology

3.1. Development phases

The development phases involved in the proposed MDA approach are depicted in Fig. 1 and explained in the remainder of this section.

As illustrated in Fig. 1, the designer should start by selecting the services he wishes to compose. A service repository such as UDDI (Bellwood et al., 2002) is usually employed to find services that provide required functionalities. Service *discoverability* is an important architectural principle of a service oriented architecture which states that services are designed to be outwardly descriptive so that they can be found and accessed via available discovery mechanisms (Erl, 2007).

Once the existing services are selected, the second step consists in elaborating the specification model of the composition. Such model should be clear, readable and implementation-independent to facilitate the comprehension of the composition by the user. It should also be precise enough to serve as a basis for the implementation, usually meaning that programming code can be directly generated from the specification model. Compliant modeling

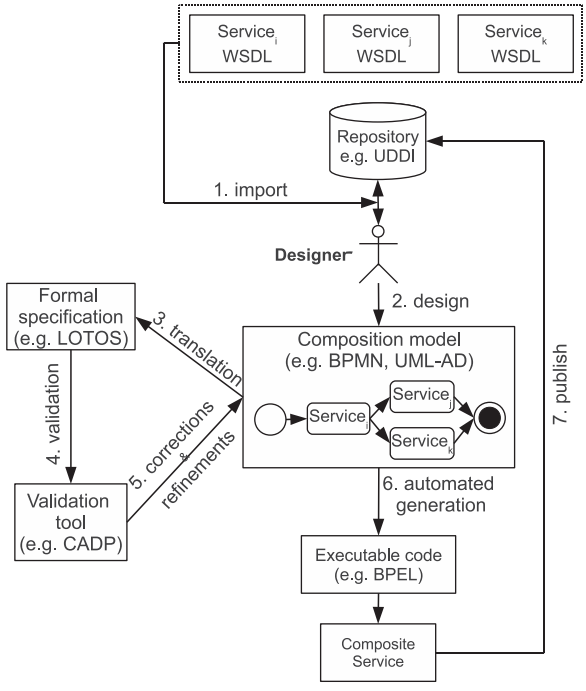

Fig. 1. Model-driven development of composite Web services supporting formal verification.

[1] See http://www.bpmn.org

languages include the Business Process Modeling Notation (BPMN) (White, 2004a) and the UML Activity Diagram (UML-AD) (OMG, 2009). Both languages are competing standards maintained by the OMG for BPM. They are well-known in the software designers community. By considering the service composition as a business process, BPMN and UML-AD can be used to model the interactions between the services in a workflow. Furthermore, they are able to model all workflow control-flow patterns supported by popular composition languages such as BPEL, as shown by White (2004b).

The third step lies in the automated translation of the specification model into a formal description. Any formal description language may be used as long as it is supported by automated verification tools. This formal specification stage is thoroughly described in Section 3.2, focusing on LOTOS as a formal specification language.

The fourth step consists in using an automated verification tool to prove that the composition realizes the expected tasks. This is done by verifying desired behavioral properties using the formal specification model. In the case where the composition is specified in LOTOS, the designer can use the CADP toolset (Fernandez et al., 1996) to verify temporal properties. These properties may describe any characteristic of the composition using temporal logic. CADP toolset will then be able to prove or disprove these properties. If errors are detected in the composition through formal verification, the designer should correct and refine the composition model (fifth step) until it is proven to be correct.

When the composition is formally verified, the next step corresponds to the implementation. As previously stated, executable code such as BPEL can directly be generated from the specification model (e.g. BPMN or UML-AD workflow). Depending on how precise the specification model is, the developer may need to edit the code of the composite Web service. Basic workflow models would only provide the structure of the code. More expressive models can be obtained, for example, by defining a profile to adapt UML to the domain of Web service composition (Gardner, 2003; Dumez et al., 2008c). The most expressive ones could be used to generate the whole code. Finally, once the composite Web service is implemented, the last step is usually to publish it into a service registry so that it can be reused later.

In the rest of this paper, we focus on LOTOS formal specification phase of workflow control-flow patterns required for web service composition description. A case study then is presented to show the usage of workflow patterns for designing, analyzing, and verifying service composition before generating the executable code. The LOTOS formal specification language has been selected to fulfill this purpose (see Appendix B for a brief overview). LOTOS is an excellent candidate due to (i) its ISO standardization, (ii) its high expressiveness and its support for value passing between processes, (iii) its formalism and (iv) the existence of verification tools that support it such as CADP (Fernandez et al., 1996).

3.2. Workflow pattern specification

To promote reusability, we model each workflow control-flow pattern as an independent LOTOS process. LOTOS does not support the concept of *function* and code may only be grouped in processes to achieve a similar result. For inter-process synchronization and to realize the desired workflow patterns, we make use of the value exchange feature in full LOTOS. Values can be exchanged synchronously at *gates*. A workflow can be seen as a directed graph where each node represents an activity. Therefore, an intuitive approach would be to represent each node (step on an activity) as a LOTOS process and each edge as a gate. However, as explained by Raymond (1989), LOTOS declares gates in static lists both as parameters to processes and in the parallel composition of processes. This mechanism does not allow for the number of gates passed to a process to be dynamic. As a consequence, by choosing this approach, we would not be able to model accurately a pattern such as the exclusive choice where a choice is made between *two or more* execution branches. Indeed, the number of possible output branches in an exclusive choice is undetermined, which makes it impossible to pass a static list of gates (branches) to the exclusive choice process. The solution to this issue proposed by Raymond (1989) is to use a single gate to specify all communications and to use a *communication medium* constraint process to ensure that communication only occurs along edges of the graph.

In this paper, we adopt the same solution by considering that every service being composed is modeled as a LOTOS process. The processes execute concurrently and communicate through a software bus, as illustrated in Fig. 2. The services can send or receive messages (events) via gates `SEND` and `RECV` respectively. The `Bus` process acts an unbounded buffer that is initially empty which accepts messages on gate `SEND` and delivers them on gate `RECV`.

The communication medium process is specified in LOTOS in Listing 1. We use two gates (SEND and RECV) instead of one to model the two-way communication between the bus and the services (processes), as suggested in Cornejo et al. (2001). Each service process is assigned with an identifier that is an integer. When communicating via the bus the services provide the identifier of the destination service, their identifier (sender) and a message *action*. The most common action we define is `RUN`, which corresponds to service invocation message. Upon reception of a `RUN` message, a process (service) will start its execution.

Listing 1. LOTOS code for communication bus.

```
process Bus [SEND, RECV] (B:Buffer) : noexit :=SEND ?R:Int ?S:
Int ?D:Cmd ?P:Int;
   Bus [SEND, RECV] (B + Message (R, S, D, P))
   []
   [not(empty(B))]– >
      (let M:Msg=head (B) in
         RECV !getrcv (M) !getsnd (M) !getcmd (M) !getprm
(M);
         Bus [SEND, RECV] (tail (B))
      )
endproc
```

After specifying the communication medium, each control-flow pattern in the original set, introduced by Aalst et al. in van der Aalst et al. (2003) could be translated into LOTOS formal specification language. We also take into account their reviewed definition by Russell et al. (2006). These patterns can be used to describe the control flow perspective of workflow systems and most of them are supported by service composition languages such as BPEL (OASIS, 2007). Instead of providing a mapping of BPEL constructs into LOTOS, we prefer to focus on the translation of generic workflow constructs. The benefit is that our mapping can be applied to virtually any workflow language. We present here the basic control flow patterns, but we refer reader to Dumez et al. (2010) for advanced patterns specification.

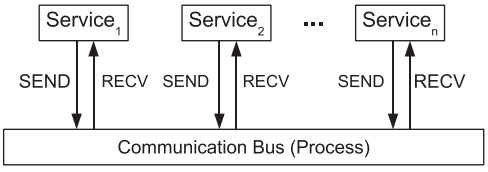

Fig. 2. Architecture of the communication between LOTOS processes.

As described by Aalst five *basic* flow control patterns, namely the *sequence*, the *parallel split*, the *synchronization*, the *exclusive choice* and the *simple merge*, are identified. These patterns capture elementary aspects of the process control. This subsection provides a definition and a rigorous translation into LOTOS for each of them. LOTOS descriptions of advanced control-flow patterns, such as the *multi-Choice*, the *structured synchronizing merge*, the *deferred choice*, the *cancel case* and the *structured loop*, are described in Appendix C.

Sequence—An activity identified by `id_dst` should be executed after the completion of the activity identified by `id` in the workflow. The LOTOS specification is provided in Listing 2 in which the current process (`Service1`) sends a `RUN` message to the next process (`Service2`) via the communication bus. The next process awaits this message before starting its execution.

Listing 2. LOTOS translation for sequence pattern.

```
process Service1 [SEND, RECV] (Id:Int) : exit :=
  (* Do work *)
  Sequence [SEND, RECV] (Id,2) >> exit
  where
  process Sequence [SEND, RECV] (Id:Int, Id_dst:Int): exit :=
SEND !Id_dst !Id !RUN !void; exit
  endproc
endproc

process Service2 [SEND, RECV] (Id:Int) : exit :=
  (* Wait for message *)
  RECV !Id ?Sender:Int !RUN !void;
  (* Do work *)
endproc
```

Note that the comments `(* Do work *)` should be replaced by the details of each activity.

Parallel split—Mechanism to execute several execution branches concurrently. A single branch diverges into two or more parallel execution branches. Each of these parallel branches contains activities that will be executed at the same time. The LOTOS specification of `ParallelSplit` process is provided in Listing 3. The identifiers of the activities (`Ids_dst`) to be executed in parallel are passed in parameters to the process as a set of integers (`IntSet`). The process needs to iterate over this set and send a `RUN` message to each activity identified in the set. However, recursion is the only way to realize cyclic behaviors in LOTOS. As a consequence, the `ParallelSplit` process is calling itself recursively and removing already processed *Ids* from the set in order to iterate over it.

Listing 3. LOTOS translation for parallel split pattern.

```
process ParallelSplit [SEND, RECV] (Id:Int, Ids_dst:IntSet) :
  exit :=
  [empty(Ids_dst)] -> exit
  []
  [not(empty(Ids_dst))] ->
    (let Dest:Int=pick(Ids_dst) in
      SEND !Dest !Id !RUN !void;
      ParallelSplit[SEND, RECV](Id, remove(Dest, Ids_dst))
    )
endproc
```

The `pick` operation returns an element from the set which is then stored in `Dest` variable using the LOTOS `let` operator.

Synchronization—Mechanism to merge two or more execution branches into a single subsequent branch with synchronization. More precisely, it waits for all input execution branches to terminate before passing the thread of execution to the output branch. This pattern is used after a parallel split in the workflow process. The corresponding LOTOS specification is given in Listing 4. The `Synchronization` process waits for the reception of one `RUN` message per input branch before exiting, thus allowing the calling process to continue its work.

Listing 4. LOTOS translation for synchronization pattern.

```
process Synchronization [SEND, RECV] (Ids_src:IntSet, Id:Int)
    : exit :=
  [empty(Ids_src)] -> exit
  []  [not(empty(Ids_src))] ->
    RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
    Synchronization [SEND, RECV] (remove(Id_src, Ids_src),
Id)
endproc
```

Exclusive choice—A split in the control flow between two or more exclusive execution paths. The thread of control is passed to one (and only one) outgoing branch. In the LOTOS specification provided in Listing 5, the choice between the output branches is nondeterministic, meaning that there is no evaluation criteria to make the decision between the branches and any one of them may be chosen in a random fashion. However, the specification we provided guarantees that only one of the output branches will be executed. The LOTOS `choice` operator in combination with the `[Dest isin Ids_dst]` guard is used to pick randomly an *Id* in the `Ids_dst` set of possible outgoing activities (branches).

Listing 5. LOTOS translation for exclusive choice pattern.

```
process ExclusiveChoice [SEND, RECV] (Id:Int,
    Ids_dst:IntSet): exit :=
  (choice Dest:Int []
    [Dest isin Ids_dst] ->
      SEND !Dest !Id !RUN !void;
      exit)
endproc
```

Simple merge—Mechanism to merge two or more exclusive execution branches into one subsequent branch. Only one of the input execution may be active. As a consequence, the simple merge pattern is used after an exclusive choice in the workflow process. The corresponding LOTOS specification is defined in Listing 6. The `SimpleMerge` process awaits a `RUN` message from any of the input activities identified in the `Ids_src` set. This is achieved through a synchronous `RUN` message reception directive whose destination is the current process (`Id`) in combination with a guard on the identifier of the sender (`Id_src`) to make sure that it belongs to the `Ids_src` set.

Listing 6. LOTOS translation for simple merge pattern.

```
process SimpleMerge [SEND, RECV] (Ids_src:IntSet, Id:Int) :
    exit :=
  RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
  exit
endproc
```

4. Case study

In this section, a working scenario is studied to show how to use the pattern specifications to translate a workflow into LOTOS in order to formally verify the service composition using CADP toolbox. This toolbox is a state-of-the-art tool set that provide an

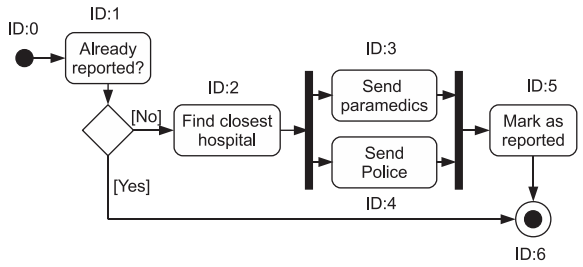

Fig. 3. Emergency response workflow described as a UML-AD.

integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification (Cornejo et al., 2001). CADP provides a wide panel of features ranging from step-by-step simulation to massively parallel model-checking (e.g., proofs of temporal properties, compositional verification, etc).

4.1. Field emergency response

This case study consists in an field emergency response process where a user can report an emergency situation. Upon receiving a report, the system will first check if the accident was already reported by other users. If it is not the case, it will find the closest hospital to the accident. Then, it will concurrently send the paramedics and a police patrol, before marking the accident as reported. This service composition workflow is modeled as a UML-AD in Fig. 3. This scenario makes use of 4 existing services:

- *HospitalLocator*: A service to find the closest hospital to a location.
- *Paramedics*: A service to send paramedics to a location.
- *PoliceDispatch*: A service to send a police patrol to a location.
- *ReportsDatabase*: A service to check if an emergency was already reported and to mark an emergency as reported (two methods).

4.2. Corresponding LOTOS model

In this subsection, a LOTOS specification for the field emergency response, based on the control-flow patterns presented in this paper, is provided. The first step to translate the activity diagram depicted in Fig. 3 into LOTOS is to create a process for each step of the activity (including initial and final nodes) and to assign a unique identifier (integer) to each one of them. The identifiers were already specified in Fig. 3 for a better understanding. The instantiation of these processes in LOTOS is provided in Listing 7. All service processes are executed concurrently using the "|||" operator, which means that they are independent and they do not communicate directly with each other. The |[SEND, RECV]| operator is used the synchronize the services with the `Bus` process through the gates `SEND` and `RECV`.

Listing 7. Processes instantiation in LOTOS.

```
specification EmergencyResponse [SEND, RECV]: noexit
behavior
  (
  Init [SEND, RECV](0) |||
  CheckIfReported [SEND, RECV](1) |||
  FindHospital [SEND, RECV] (2) |||
  SendParamedics [SEND, RECV](3) |||
  SendPolice [SEND, RECV](4) |||
  MarkAsReported [SEND, RECV](5) |||
  Final [SEND, RECV](6)
  )
  |[SEND,RECV]|
  BUS [SEND,RECV] ( < > )
where
  (* Processes definition *)
endspec
```

The next step is the translation into LOTOS by identifying the control-flow patterns in the workflow in order to provide a definition (implementation) for each process. The `Init` process (*Id:0*) merely starts the `CheckIfReported` process (*Id:1*). As a consequence, it uses the sequence pattern before exiting, as defined in Listing 8.

Listing 8. LOTOS specification for Init process.

```
process Init [SEND, RECV] (Id:Int) : exit :=
  Sequence [SEND, RECV] (Id, 1)
  > > exit
endproc
```

The `CheckIfReported` process waits for a `RUN` message from `Init` before starting. After that, it realizes an exclusive choice between `FindHospital` (*Id:2*) and the `Final` process (*Id:6*), as defined in Listing 9.

Listing 9. LOTOS specification for CheckIfReported process.

```
process CheckIfReported [SEND, RECV] (Id:Int) : exit :=
  RECV !Id !0 !RUN !void;
  ExclusiveChoice [SEND, RECV] (Id, insert(6, insert(2, {})))
  > > exit
endproc
```

The *FindHospital* process waits for a `RUN` message from the `CheckIfReported` process before starting concurrently the `SendParamedics` (*Id:3*) and `SendPolice` (*Id:4*) processes, thus realizing a parallel split pattern. The corresponding specification is provided in Listing 10.

Listing 10. LOTOS specification for FindHospital process.

```
process FindHospital [SEND, RECV] (Id:Int) : exit :=
  RECV !Id !1 !RUN !void;
  ParallelSplit [SEND, RECV] (Id, insert(4, insert(3, {})))
  > > exit
endproc
```

The `SendParamedics` and `SendPolice` processes both await a `RUN` message from the `FindHospital` process (*Id:2*) before executing and eventually starting the `Mark-AsReported` process (*Id:5*). Only the definition for `Send-Paramedics` is given in Listing 11 since `SendPolice` has the exact same implementation.

Listing 11. LOTOS specification for SendParamedics process.

```
process SendParamedics [SEND, RECV] (Id:Int) : exit :=
  RECV !Id !2 !RUN !void;
  Sequence [SEND, RECV] (Id,5)> > exit
endproc
```

The `MarkAsReported` process synchronizes its two incoming branches (paramedics and police calls) before executing and

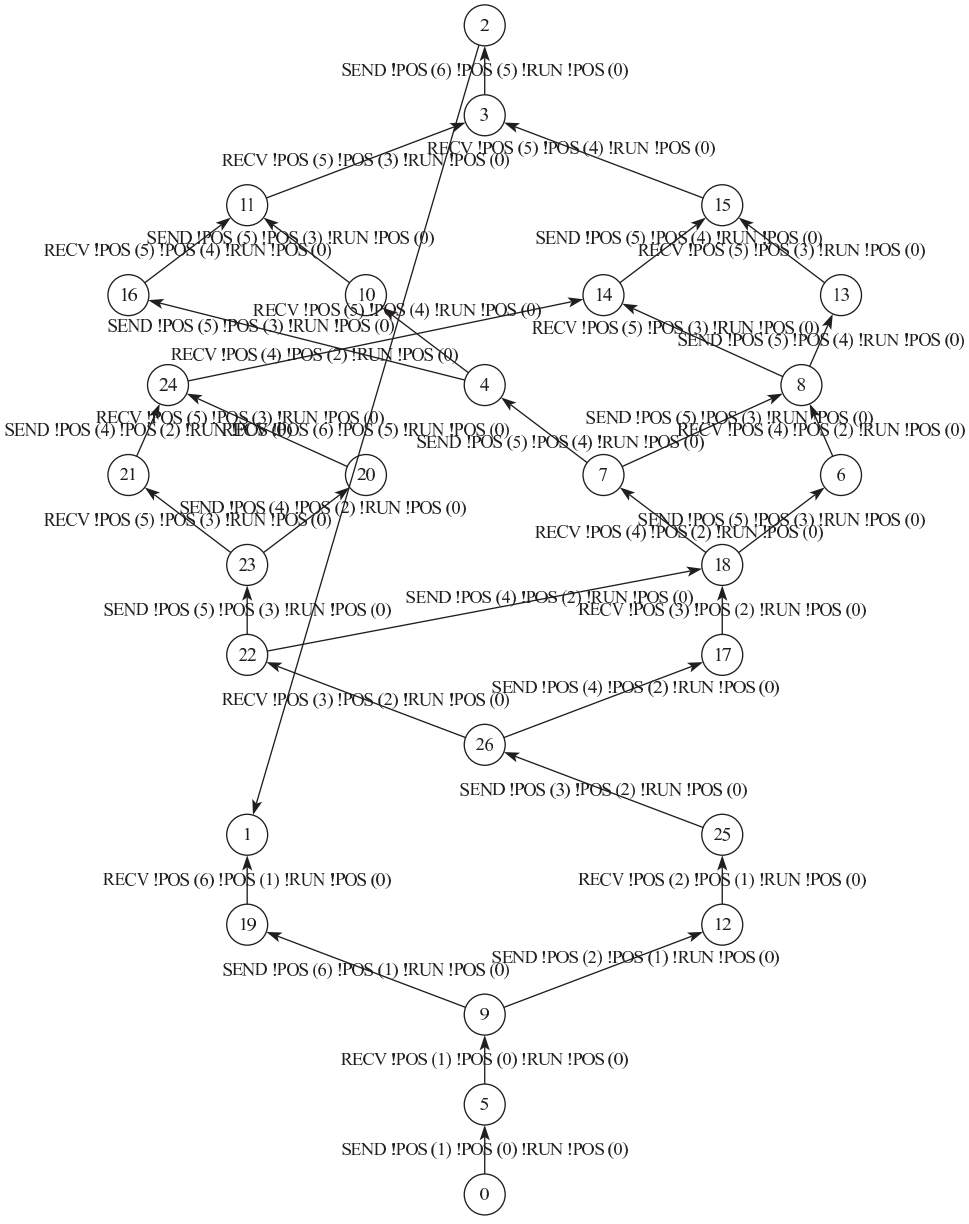

Fig. 4. LTS generated by CADP.

eventually starting the `Final` process. This behavior is defined in Listing 12.

Listing 12. LOTOS specification for MarkAsReported process.

```
process MarkAsReported [SEND, RECV] (Id:Int) : exit :=
    Synchronization [SEND, RECV] (insert(4, insert(3, {})), Id)
    >>
    Sequence [SEND, RECV] (Id,6) >> exit
endproc
```

Finally, the `Final` process (corresponding the final node of the activity diagram) will merge its two incoming branches that were earlier split by an exclusive choice. As a consequence, and as stated in Listing 13, it realizes a simple merge between the branches coming from the `MarkAsReported` and `CheckIfReported` processes.

Listing 13. LOTOS specification for Final process

```
process Final [SEND, RECV] (Id:Int) : exit :=
    SimpleMerge [SEND, RECV] (insert(5, insert(1, {})), id)
    >> exit
endproc
```

4.3. Properties verification

To analyze the behavior of the composite service described above, we used the CADP toolkit, which provides an on-the-fly model checker for *regular* alternation-free μ-calculus formulas on Labeled Transition Systems (LTS). The temporal logical used as input language is an extension of the alternation-free μ-calculus (Kozen, 1983; Emerson and Lei, 1986) with boolean formulas over actions and regular expressions over action sequences. As a consequence, once the service composition scenario is specified in LOTOS, CADP starts by translating the LOTOS specification into a LTS. The LTS generated from the emergency response LOTOS specification by CADP is provided in Fig. 4. To increase its readability, a minimization according to branching bissimulation was

achieved using CADP. To realize such graph reduction, CADP uses the algorithm presented in Groote and Vaandrager (1990).

The developer can then describe behavioral properties regarding the composition using μ-calculus and proceed with their automatic verification on the generated LTS using CADP evaluator. The evaluator will not only tell if the property is verified or not but it will also construct examples (or counter examples) to understand why. CADP evaluator input language can be extended via macro-expansion in order to improve readability. In Listing 14, we provide a few macros used to verify this service composition workflow. The first macro takes two actions in parameters (`A` and `B`) and verifies that action `A` necessarily leads to action `B`.

Listing 14. Verification macros definition.

```
macro A_inev_B (A, B)=
  [true* . (A) ] mu X . ( <true> true and [not (B) ] X)
end_macro
macro ACC_REPORT()='SEND !POS (1) !POS (0) !RUN.*'
end_macro
macro ALREADY_REPORTED()='SEND !POS (6) !POS (1)
!RUN.*' end_macro
macro SEND_PARAMEDICS()='SEND !POS (3) !POS (2) !RUN.*'
  end_macro
macro SEND_POLICE()='SEND !POS (4) !POS (2) !RUN.*'
  end_macro
```

The last four macros correspond to actions in the workflow. For example, the `SEND_PARAMEDICS` macro refers to `SendParamedics` activity execution. These *action* macros use UNIX regular expressions in order to match the `RUN` messages corresponding to each activity. Note that in these macros, `POS` merely refers to the constructor for a positive integer.

Using these macros (see Listing 14), it is now possible to define properties of the composition and verify them. In other words, we can verify that the service composition being specified realizes the expected behavior. One of the properties we can check is that the process never sends the paramedics nor the police if the emergency was already reported. This is actually a *safety* property because it is used to make sure an undesirable event will never occur. This property can be expressed in temporal logic as follows:

```
([ALREADY_REPORTED . SEND_PARAMEDICS ] false) and
  ([ALREADY_REPORTED . SEND_POLICE ] false)
```

Another important property we can verify is that whenever an accident is reported, the process always sends the paramedics, unless the emergency was already previously reported. This is actually a *liveness* property because it is used to make sure a desirable event *will* happen. This translates into temporal logic as follows:

```
A_inev_B (ACC_REPORT, SEND_PARAMEDICS or
ALREADY_REPORTED)
```

Both these properties were automatically verified using the CADP evaluator and they were both evaluated as being `TRUE`, meaning that they are verified.

4.4. BPEL code generation

After the verification of the service composition specification, the next development stage is the implementation. In a MDA approach such as the one presented in this paper, the implementation time is greatly decreased because part of the programming code can be generated from the specification model. This is widely used already in software engineering where code such as Java is generated from UML diagrams.

In the context of Web service composition, the computation is actually achieved by the services being composed and not the composite service itself. As a consequence, an important part of the composite Web service code can be generated from the workflow model. It is thus possible to use the activity diagram previously presented in Fig. 3 to generate the BPEL code for the composite Web service shown in Listing 15. It is worth noting that this figure only provides the overall structure BPEL process because the UML activity diagram does not specify any Web service related information nor any data information. However, we developed an open source development environment[2] that allows the developer to import existing services, specify the composition using UML and generate the BPEL code. To be able to generate the whole code for the composite Web service, the framework uses a specific UML profile called UML-S (Dumez et al., 2008c) or *UML for Services engineering* that customizes UML 2.x for the specific purpose of Web service composition. UML-S has the same metamodel as standard UML, it merely defines specific stereotypes, tagged values and constraints to increase the expressiveness of UML models in the context of service composition.

Listing 15. The BPEL executable code generated from UML-AD.

```
<process name = EmergencyResponse>
  <sequence>
    <receive operation = ReportEmergency  />
    <invoke name = "wasAlreadyReported" ... />
    <if name = "notAlreadyReported">
      <condition> not_reported </condition>
      <sequence>
        <flow name = "Parallel">
          <invoke name = "SendParamedics" ... />
          <invoke name = "SendPolice"  />
        </flow>
        <invoke name = "markAsReported"  />
      </sequence>
    </if>
    <reply operation = ReportEmergency ... />
  </sequence>
</process>
```

5. Conclusions and future work

The formal verification of Web service composition is an important task that is not supported by current composition languages, due to their lack of well-defined formal semantics. This issue can be addressed using formal methods and existing formal verification tools.

In this paper, we presented a MDA approach to specify, verify and implement service composition using existing specification and implementation languages. To support the formal verification of the composition, we proposed to translate the composition workflow model into a LOTOS formal specification. To achieve this task, a mapping into LOTOS for each of the 20 original workflow control-flow patterns was provided. The CADP toolset can then be used to verify the composition via its LOTOS specification, before generating the programming code.

[2] http://sourceforge.net/projects/uml-s/

The applicability and the effectiveness of the proposed methodology was illustrated through an example. The UML activity diagram was used to model the composition in a workflow. It was then translated into LOTOS formal specification language to proceed with the verification of temporal properties using CADP toolset. Finally, the structure of the BPEL code was generated from the workflow model. Future work addresses the enhancement of the tool supporting the proposed methodology with additional automation features as well as its application to complex case studies that includes advanced control-flow patterns.

Appendix A. Classification and comparison of composition approaches

Composition approaches described in the related work section are depicted in Fig. A1. These approaches are compared based their support of control-flow patterns used by WS-BPEL 2.0, which is the most common executable code for Web service orchestration. The mapping between these Flow-control patterns (WCP) and the BPEL constructs is provided in Table A1.

Let us first consider the five *basic* control-flow patterns, namely the *sequence* (WCP-1[3] the *parallel split* (WCP-2), the *synchronization* (WCP-3), the *exclusive choice* (WCP-4) and the *simple merge* (WCP-5). The sequence enables the developer to execute activities in a given order, as opposed to the parallel split that is used to execute them simultaneously. The synchronization joins parallel execution paths and waits for all of them to finish before continuing. The exclusive choice (or XOR-split) where only one of several branches gets chosen according to a condition. Finally, the last basic pattern is the simple merge that joins two or more alternative branches without synchronization.

Most of the models depicted in Fig. A1 support the five basic control flow patterns. Only the models based on state automata do not directly support the modeling of the parallel split and the synchronization patterns. This is due to the fact that the semantics of automata cannot model concurrency. However, solutions have been proposed to solve this issue. For example, Yan and Dague (2007) proposed to model each of the parallel branches as an individual automaton and define synchronization events to build their connections.

In the following, the models are compared based on their support for more advanced control-flow patterns such as the *multi-Choice* (WCP-6), the *structured synchronizing merge* (WCP-7), the *deferred choice* (WCP-16), the *cancel case* (WCP-20) and the *structured loop* (WCP-21). The multi-choice (*OR-split*) is an extension of the exclusive choice where more than one output branch can be executed. The structured synchronizing merge appears in a workflow after a multi-choice in order to merge the input branches with synchronization. The deferred choice is a point in a process where one of several branches is chosen based on interaction with the operating environment. Before this point, all output branches represent possible future courses of execution. The cancel case terminates a whole process instance, meaning that all executing tasks are stopped. Finally, the structured loop has the ability to execute a task or sub-process repeatedly.

The automata support all the advanced patterns except the multi-choice and the structured synchronizing merge due to their lack of support for concurrent modeling. The classical and timed Petri nets cannot represent these two patterns either because they lack conditions on the arcs. Colored Petri nets, however, can model all patterns as presented by Aalst et al. in van der Aalst et al. (2003) and Russell et al. (2006). CPN represents the multi-choice as an exclusive choice, except that the condition of the arcs are not necessarily distinct so that one or more output transitions can be fired at once. For process algebras, we limited our study to CSP, CSS and π-calculus. According to Wong and Gibbons (2007), CSP supports all patterns. However, we are uncertain about the structured synchronizing merge because their paper does not mention it. CCS representation for all patterns have been provided by Stefansen (2005). However, we consider that the structured synchronizing merge representation is not entirely satisfactory because it requires the number of input branches that were executed to be known in advance. The same thing is true for the π-calculus representation of this pattern that was given by Puhlmann and Weske (2005). Finally, both UML-AD and BPMN provide support for all considered patterns, as presented by White (2004b). The comparison results for advanced control-flow patterns support are summarized in Table A2.

Table A3 summarizes the features these formalisms provide for designers mainly (Ferrara, 2004; Frappier and Habrias, 2006):

- Model checking: determines if properties (e.g., liveness and safety) about the specification can be checked by enumeration of the system states.
- Temporal logic: determines if the notation includes some first-order (or higher-order) logic notation. Logic is useful to express properties in an abstract manner.
- Provability: determines if properties about the service specification can be proved using a formal proof system.
- Bisimulation: checks whether the behaviors of two services or two versions of the same service are equivalent; or, if they are different.
- Simulation: checks whether the behavior of a service is included within the behavior of other interacting services.

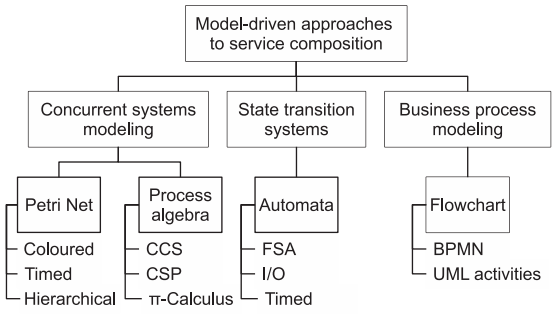

Fig. A1. Model-driven approaches to WS composition: a classification.

Table A1
Main flow-control patterns supported by BPEL.

Flow-control pattern	WS-BPEL 2.0 construct
Sequence (WCP-1)	<sequence>
Parallel split (WCP-2)	<flow>
Synchronization (WCP-3)	<flow>
Exclusive choice (WCP-4)	<if>/<else> or links within a <flow>
Simple merge (WCP-5)	<if>/<else> or links within a <flow>
Multi-choice (WCP-6)	Links within a <flow>
Structured synchronizing merge (WCP-7)	Links within a <flow>
Deferred choice (WCP-16)	<pick>
Cancel Case (WCP-20)	<terminate>
Structured loop (WCP-21)	<while>

[3] WCP-*i* refer to the control-flow pattern identifiers at www.workflowpatterns.com.

- Execution traces of the service: to understand the behavior of the service.

For example, process algebra formalisms provide all facilities that allow designers to verify the WS composition. Note that these formalisms do not include a logic notation. However, their semantics are defined in terms of a set of inference rules expressed in a logic notation (Ferrara, 2004). Simulation and bisimulation features are also provided, which are required for automatic composition. However, in spite of the facilities they provide, specification using process algebra is rather complicated and difficult to read for complex WS composition.

Petri net formalisms do not include logic and do not allow simulation or bisimulation, but they provide facilities to allow designers to verify properties, which can be checked by enumeration of the system states, or using formal proof systems or by execution traces of the service. The graphical nature of these formalisms and the available verification tools make Petri net formalisms more attractive to the community.

As described in Table A3, business process methods do not support verification, since their semantics are not based on formal notations. If the verification of business process models is required, it is necessary to convert these models to formal ones such as Petri net or π-calculus (Staines, 2008; Dijkman et al., 2008). Such approach is interesting because business process models are compact, easy to use and understand. Service composition development using tools such as BPMN or UML-AD is thus very popular.

Appendix B. Overview of LOTOS

LOTOS (Brinksma, 1988) is a formal description developed within ISO (International Standards Organization) for the specification of open distributed systems, especially protocols in OSI standards. LOTOS is based on process algebraic methods by combining the features provided by CCS (Milner, 1982) and CSP (Hoare, 1978).

The LOTOS behavior operators are summarized in Table A4, where G refers to a gate (channel of communication between processes), X to a variable, P to a process, S to a sort, V to a value and B to a behavior.

The symbol *stop* denotes an inactive behavior (Cornejo et al., 2001; Ferrara, 2004). It can also be viewed as the end of a behavior. The symbol *exit* is used to describe a normal termination of a process. The expression `G !V ?X:S ; B` denotes that the behavior B is interacting on gate G, sending value V and receiving in variable X a value of sort S, then continues its execution. The non-deterministic choice between two behaviors is represented using ``[]''. `[E]->B` denotes that the process should behave as B if E is true. LOTOS has three parallel composition operators. The general case is given by the expression $B_1 \,|[G_1,\ldots,G_n]|\, B_2$, which denotes the parallel execution of B_1 and B_2 with synchronization on gates $G_1,\ldots,G_n$. In other words, B_1 and B_2 evolve independently but synchronize on gates $G_1,\ldots,G_n$. B_1 and B_2 also synchronize on the termination *exit*. The two other operators are particular cases of the former one: $B_1 \,||\, B_2$ denotes the execution of B_1 and B_2 in parallel without synchronization (interleaving) and $B_1 \,|||\, B_2$ means that B_1 and B_2 synchronize on all actions. The sequential execution is described by $B_1 >> B_2$, which denotes the execution of B_1 is followed by B_2 when B_1 terminates. The expression $B[G_1,\ldots,G_n]\,(V_1,\ldots,V_m)$ is a call to the process B with gate parameters $G_1,\ldots,G_n$ and value parameters $V_1,\ldots,V_m$. More precisely, the full syntax of a process is described as follows:

process P[G_{0}, …, G_{m}] (X_{0}:T_{0}, …,
X_{n} : T_{n}) : func :=
B
endproc

where *B* is the behavior of the process *P* and *func* corresponds to the functionality of the process: either the process loops endlessly

Table A2
Advanced flow-control patterns support.

Family	Model	Multi-choice	Deferred choice	Cancel case	Sync. merge	Struct. loop
State-transition	Automata	–	+	+	–	+
Petri Net	Classical	–	+	+	–	+
	Colored	+	+	+	+	+
	Timed	–	+	+	–	+
Process algebra	CSP	+	+	+	?	+
	CCS	+	+	+	+/–	+
	π-calculus	+	+	+	+/–	+
	LOTOS	+	+	+	+/–	+
Business Process	UML-AD	+	+	+	+	+
	BPMN	+	+	+	+	+

Table A4
LOTOS behavior operators.

Behavior operator	Meaning
`stop`	Inaction
`exit`	Successful termination
`G !V ?X:S ; B`	Action Prefix
B_1 [] B_2	Choice
[E]–>B	Conditional
$B_1 \,\|[G_1,\ldots,G_n]\|\, B_2$	Parallel composition
$B_1 \,\|\|\|\, B_2$	Interleaving
$B_1 \,\|\|\, B_2$	Full synchronization
$B_1 >> B_2$	Sequential composition
$P[G_1,\ldots,G_n](V_1,\ldots,V_m)$	Process call

Table A3
Verification features.

Family	Model	Model checking	Bisimulation	Simulation	Execution trace	Logic	Provability
State transition	Automata	+	–	–	+	+	+
Petri net	Classical	+	–	–	+	–	+
	Colored	+	–	–	+	–	+
	Timed	+	–	–	+	–	+
Process algebra	CSP	+	+	+	+	+	+
	CCS	+	+	+	+	+	+
	π-calculus	+	+	+	+	+	+
	LOTOS	+	+	+	+	+	+
Business process	UML-AD	–	–	–	–	–	–
	BPMN	–	–	–	–	–	–

(*noexit*), or it terminates (*exit*) possibly returning results of type $T_{j \in 1..n}$ (*exit*($T_0, \ldots, T_n$)).

Appendix C. LOTOS descriptions of advanced patterns

C.1. Branching and synchronization patterns

This subsection provides the LOTOS translation for four more complex branching and merging concepts that arise in business processes: the *multi-choice*, the *structured synchronizing merge*, the *multi-merge* and the *structured discriminator*.

Multi-choice—A split in the control flow between two or more execution paths. The thread of control is passed to one or several outgoing branches. This pattern is essentially an analogue of the exclusive choice pattern in which multiple outgoing branches can be chosen and executed. In the LOTOS specification provided in Listing 16, the choice between the output branches is nondeterministic.

Listing 16. LOTOS translation for multi-choice pattern.

```
(* Initially, Nb_active=0 *)
process MultiChoice [SEND, RECV] (Id:Int, Ids_dst:IntSet,
    Id_merger:Int, Nb_active:Int): exit :=
  [empty(Ids_dst)]-->
    SEND !Id_merger !Id !ACT !Nb_active; exit
  []
  [not(empty(Ids_dst))]-->
    (choice Dest:Int []
      [Dest isin Ids_dst]--> SEND !Dest !Id !RUN !void;
      (MultiChoice [SEND, RECV] (Id, remove(Dest, Ids_dst),
          Id_merger, Nb_active+1)
      []
      (** Notify which branches are active for later merging
          **)
      SEND !Id_merger !Id !ACT !Nb_active+1; exit)
    )
endproc
```

In this specification, a random branch is chosen in the same way as in the exclusive choice. The difference lies in the fact that the ''[]'' operator is used without guards after the selection to realize a nondeterministic choice between recursion and exit. If the recursion is chosen, the pattern will select an additional outgoing branch before reproducing the same nondeterministic choice. This branching pattern makes the future merging of the parallel execution branches difficult, especially when synchronization between the input branches is required (i.e. structured synchronizing merge). Indeed, synchronization should only be made on *activated* input branches (the branches that were selected by the previous multi-choice pattern). The branches that have been activated can only be known at runtime. To solve this problem, the `MultiChoice` process sends an `ACT` message to the process that will merge the branches. This message contains a parameter (`Nb_active`) which indicates the number of branches that have been activated.

Structured synchronizing merge—Mechanism to merge two or more execution branches such that synchronization is achieved on the activated input branches before passing the thread of execution to the subsequent branch. This structure provides a means of merging the branches resulting from a multi-choice construct earlier in the workflow. A specification in LOTOS for this pattern is proposed in Listing 17. It takes in parameters the identifiers of the activities (branches) to merge (`Ids_sec`), the identifier of the current merging process (`Id`), the number of active input branches (`Nb_active`) and the number of input branches that have already terminated (`Nb_synced`).

Listing 17. LOTOS translation for structured synchronizing merge pattern.

```
(* Initially, Nb_active=Nb_synced=0 *)
process SynchronizingMerge [SEND, RECV] (Ids_src:IntSet,
    Id:Int,
Nb_active:Int, Nb_synced:Int): exit :=
  [Nb_active = 0]-->
    (RECV !Id ?dummy:Int !ACT ?Nb:Int;
      ([Nb_synced = Nb]--> exit
      []
      [Nb_synced < Nb]-->
SynchronizingMerge [SEND, RECV] (Ids_src, Id, Nb,
            Nb_synced) )
    )
  []
  (RECV !Id ?Source:Int !RUN !void [Source isin Ids_src];
    ([Nb_synced+1 = Nb_active]--> exit
    []
    [Nb_synced+1 <> Nb_active]-->
      SynchronizingMerge [SEND, RECV](remove(Source,
          Ids_src), Id, Nb_active, Nb_synced+1))
  )
endproc
```

The number of input branches that were activated (`Nb_active`) is retrieved from the `ACT` message that was sent to the merging process by the earlier multi-choice process. The number of input branches that have already terminated (`Nb_synced`) is incremented on recursion, upon reception of a `RUN` message from an input branch. To achieve synchronization, the `SynchronizingMerge` process waits for all active input branches to terminate (i.e. `[Nb_synced=Nb_active]`) before exiting. Upon exit, the process that called the synchronizing merge can continue its execution.

Multi-merge—Mechanism to merge two or more execution branches without any synchronization. The termination of each input branch will result in the thread of control being passed to the subsequent branch. This structure provides a means of merging the branches resulting from a multi-choice construct earlier in the workflow. A specification in LOTOS for this pattern is proposed in Listing 18. The specification is similar to the one of the structured synchronizing merge. However, the `MultiMerge` process takes one more parameter: the identifier of the next process to be executed upon the completion of an input branch (`Id_nxt`). More precisely, the merging process realizes this behavior by sending a `RUN` message to the next process (activity) in the workflow, whenever it receives a `RUN` message from an incoming process (branch). The number of input branches that were activated by the earlier multi-choice construct is used in this specification to detect the end of the merging process resulting in the *exit* action.

Listing 18. LOTOS translation for multi-merge pattern.

```
(* Initially, Nb_active=Nb_merged=0 *)
process MultiMerge [SEND, RECV] (Ids_src:IntSet, Id:Int,
    Id_nxt:Int, Nb_active:Int, Nb_merged:Int): exit :=
  [Nb_active = 0]-->
    (RECV !Id ?dummy:Int !ACT ?Nb:Int;
      ([Nb_merged = Nb]--> exit
      []
  [Nb_merged < Nb]-->
MultiMerge [SEND, RECV] (Ids_src, Id, Id_nxt, Nb,
```

```
            Nb_merged))
    )
  []
  (
  (* Wait for incoming branch to terminate *)
  RECV !Id ?Source:Int !RUN !void [Source isin Ids_src];
  (* Call next activity *)
  SEND !Id_nxt !Id !RUN !void;
  (* Check if there are more active branches to merge *)
  ([Nb_merged+1 = Nb_active]-> exit
  []
  [Nb_merged+1 <> Nb_active]->
    MultiMerge [SEND, RECV] (remove(Source, Ids_src), Id,
        Id_nxt, Nb_active, Nb_merged+1))
  )
endproc
```

Structured discriminator—Mechanism to merge two or more parallel execution branches so that the thread of execution is passed to the subsequent branch when the first input branch is finished. The termination of other input branches does not result in the thread of control being passed on. The structured discriminator resets once all input branches are finished. This pattern can only be used after a parallel split pattern. Its LOTOS specification is given in Listing 19. The `Discriminator` process takes the identifier of the next process in the workflow to be executed (`Id_nxt`) upon the completion of the first (fastest) input branch. Once the first input branch is finished, the `Discriminator` process calls the `Synchronization` process in order to wait from the other input branches to finish before exiting.

Listing 19. LOTOS translation for structured discriminator pattern.

```
process Discriminator [SEND, RECV] (Ids_src:IntSet, Id:Int,
    Id_nxt:Int): exit :=
  (* Wait for first branch to complete *)
  RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
  (* Call next process *)
  SEND !Id_nxt !Id !RUN !void;
  (* Wait for other branches to complete and ignore them *)
  Synchronization [SEND, RECV] (remove(Id_src, Ids_src), Id)
  >> exit
endproc
```

C.2. Structural patterns

Structural patterns show restrictions on workflow languages, for example that arbitrary loops are not allowed. LOTOS easily handles both of the following patterns.

Arbitrary cycles—The ability to represent cycles (loops) that have more than one entry or exit point. This is also referred to as an iteration pattern. In LOTOS, arbitrary cycles can be achieved using an exclusive choice together with a simple merge.

Implicit termination—A given process instance should terminate when there is no remaining work to do either now or at any time in the future and the process instance is not in deadlock. This is also referred to as a termination pattern. In LOTOS, a process simply executes the `exit` action once it finishes its work.

C.3. Multiple instance patterns

Multiple instance patterns describe situations where there are multiple threads of execution active in a process model, which relate to the same activity (and hence share the same implementation definition). LOTOS only provides a partial support for these patterns because it cannot create a dynamic number of new instances of an activity. The number of instances needs to be specified at design time.

Multiple instances without synchronization—Within a given process instance, multiple instances of a task can be created. These instances are independent from each other and run concurrently. There is no requirement to synchronize them upon completion. LOTOS can create several instances of the same process and have them run concurrently using the "|||" operator as presented in Listing 20, provided that the number of instances is known at design time. In the code sample provided, three instances for the process `S1` are created. LOTOS does not require these instances to be synchronized upon exit. Once all the instances have been created, they should all await for a `RUN` message before starting their execution. The LOTOS code provided in Listing 21 can then be used to start the instances. Basically, the `StartInstances` process takes in parameter the identifier of the process whose instances should be started (`ProcessId`) and the number of instances to start (`NbInstances`). It will then send as many `RUN` messages as instances to start.

Listing 20. LOTOS translation for multiple instances without synchronization pattern.

```
behavior
  S1[SEND,RECV](1) ||| S1[SEND,RECV](1) |||
S1[SEND,RECV](1)
    (*||| ... *)
where
  (* ... *)
```

Listing 21. LOTOS process for starting multiple instances of a process.

```
process StartInstances [SEND, RECV] (Id:Int, ProcessId:Int,
NbInstances:Int) : exit :=
  SEND !ProcessId !Id !RUN !void;
  ([NbInstances = 1]-> exit
  []
  [NbInstances > 1]->
    StartInstances [SEND, RECV] (Id, ProcessId,
        NbInstances-1))
endproc
```

Multiple instances with a priori design time knowledge—Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent from each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered. The process for creating process instances and starting them is identical as in the previous pattern. However, to achieve the multiple instances with a priori design time knowledge pattern, it is necessary to synchronize the instances on completion. This can be achieved in LOTOS using the code provided in Listing 22. It is an adaptation of the `Synchronization` specification where all awaited processes have the same identifier (`ProcessId`).

Listing 22. LOTOS code to wait for the completion of several instances of a process.

```
process SynchronizeInstances [SEND, RECV] (Id:Int,
    ProcessId:Int, NbInstances:Int) : exit :=
  RECV !Id !ProcessId !RUN !void;
  ([NbInstances = 1]-> exit
  []
```

```
    [NbInstances > 1]– >
      SynchronizeInstances [SEND, RECV] (Id, ProcessId,
        NbInstances – 1))
endproc
```

Multiple Instances with a priori Run-Time Knowledge—This pattern is similar to the previous one, except that the number of instances is not at run-time only. As explained earlier in this subsection, LOTOS does not support such behavior because the number of instances must be specified at design time.

Multiple Instances without a priori Run-Time Knowledge—This pattern is also similar to the previous ones except that the number of instances is not known a priori. At any time, whilst instances are running, it is possible for additional instances to be initiated, based on a number of run-time factors. This behavior is not supported by LOTOS either.

C.4. State-based patterns

State-based patterns reflect situations that are more easily realized in process languages that support the notion of state. While this is not the case for LOTOS, it is still possible to specify these patterns.

Deferred Choice—A deferred choice is much like an exclusive choice except that the selection of the output branch is not made explicitly in the activity, but rather by the environment. This behavior can be specified in LOTOS using the ``[]'' operator without any guard, resulting in the choice being made by the environment instead of the activity itself.

Interleaved Parallel Routing—Mechanism to execute a set of tasks in a sequential order that is only partially defined. This pattern offers the possibility of relaxing the strict ordering that a process usually imposes over a set of tasks. This behavior can be achieved in LOTOS by making a random choice between the processes and by using recursion to reproduce the choice until all processes in the sequence have been executed. The corresponding LOTOS specification is given in Listing 23. The LOTOS `choice` operator is used here to make the random choice between the processes. For the `UnorderedSequence` process to work, it is necessary that all processes in the sequence notify the `Unordered-Sequence` process when they complete their execution. The notification can be made using a simple `RUN` message. As a consequence, the `UnorderedSequence` process awaits a `RUN` message whenever it calls another process to make sure that the process has terminated before calling the next process in the sequence.

Listing 23. LOTOS translation for interleaved parallel routing pattern.

```
process UnorderedSequence [SEND, RECV] (Id:Int,
    SeqIds:IntSet) : exit :=
  [empty(SeqIds)]– > exit
  []
  [not(empty(SeqIds))]– >
    (choice SeqId:Int []
    [SeqId isin SeqIds]– >
      Sequence [SEND, RECV] (Id, SeqId) > >
      RECV !Id !SeqId !RUN !void; (* Wait for process to
          complete *)
      UnorderedSequence [SEND, RECV] (Id, remove(SeqId,
          SeqIds))
    )
endproc
```

Milestone—A task is only executed when the process instance is in a specific state. This state is a specific execution point in the process that is called *milestone*. Whenever the milestone is reached, the task may be executed. If the process instance has progressed beyond the milestone, then the task can no longer be executed. We propose to model this in LOTOS by considering that the milestone is a `RUN` message. If two processes wait for the same `RUN` message then only one of them will be executed, in a nondeterministic fashion. For example, we consider that a process *A* is followed by a process *B* and that the milestone is reached when *A* is complete. Upon completion, *A* sends a `RUN` message to *B*. The solution is to have another process *C* wait for the `RUN` message from *A* to *B* and execute the milestone task if such message is received. Either *B* or *C* can receive the *RUN* message from *A*. If *B* receives the message, then the milestone task will *not* be executed. The LOTOS specification for the process that waits for the milestone is provided in Listing 24. The task that can only be executed when the milestone is reached is identified by `Id_nxt`.

Listing 24. LOTOS translation for milestone pattern.

```
process Milestone [SEND, RECV] (Id_src:Int, Id:Int,
    Id_nxt:Int): exit :=
  (* Wait for milestone *)
  RECV !Id !Id_src !RUN !void;
  (* Call next process *)
  SEND !Id_nxt !Id_src !RUN !void;
  (* Loop in case we reach the milestone again *)
  Milestone [SEND, RECV] (Id_src, Id, Id_nxt)
endproc
```

C.5. Cancellation patterns

The cancellation patterns describes the withdrawal of one or more activity from the workflow process.

Cancel Task/Activity—An enabled task is withdrawn prior to starting its execution. In LOTOS, we can specify this by adding a new message action that would cause the withdrawal of destination activity (e.g. a `CANCEL` message). Processes would then wait either for a `RUN` message that would cause them to execute normally, or a `CANCEL` message that would cause them to exit. An example of such LOTOS process is provided in Listing 25.

Listing 25. LOTOS code for task cancellation.

```
process S1 [SEND, RECV] (Id:Int): exit :=
  (RECV !Id ?Dummy:Int !CANCEL !void;
  (* Task is cancelled *)
  exit)
  []
  (RECV !Id ?Sender:Int !Run !void;
  (* Do normal work *)
  exit)
endproc
```

Cancel Case—A complete process instance is removed, including currently running tasks and the ones that may execute at some future time. This is specified in LOTOS as in the previous pattern except that the `CANCEL` message should be broadcasted to all tasks in the process instance. Note, however, that one can only withdraw tasks that have not already started.

References

Achilleos A, Yang K, Georgalas N, Azmoodech M. Pervasive service creation using a model driven petri net based approach. In: IWCMC'08. International wireless communications and mobile computing conference; 2008. p. 309–14.

Amsden J, Gardner T, Griffin C, Iyengar S. Draft UML 1.4 profile for automated business processes with a mapping to BPEL 1.0. Technical report, 〈http://www.128.ibm.com/developerworks/rational/library/content/04April/3103/〉. 2003.

Bakhouya M, Gaber J. Service composition approaches for ubiquitous and pervasive computing environments: a survey. In: Yuan Soe-Tsyr, editor. Agent systems in electronic business, vol. 3. Idea Group Publication; 2008. p. 323–50.

Beek MHt, Bucchiarone A, Gnesi S. Formal methods for service composition. Annals of Mathematics, Computing & Teleinformatics 2007;1(5):1–10.

Bellwood T, Clement L, Ehnebuske D, Hately A, Hondo M, Husband YL, et al. Uddi version 3.0. Published specification, Oasis. 2002.

Ben Mokhtar S, Georgantas N, Issarny V. Cocoa: conversation-based service composition for pervasive computing environments. 2006 ACS/IEEE International Conference on Pervasive Services. June 2006. p. 29–38. http://dx.doi.org/10.1109/PERSER.2006.1652204.

Bordeaux L, Salaun G. Using process algebra for web services: early results and perspectives. Lecture Notes in Computer Science 2005;3324:54–68.

Brinksma E. Information processing systems-open systems interconnection-LOTOS-a formal description technique based on the temporal ordering of observational behaviour. International Standard, ISO 8807. 1988.

Camara J, Canal C, Cubo J, Vallecillo A. Formalizing web services choreographies. Electronic Notes in Theoretical Computer Science 2006;154:159–73.

Cornejo MA, Garavel H, Mateescu R, Palma ND. Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In: Proceedings of the IFIP TC6/WG6.1 third international working conference on new developments in distributed applications and interoperable systems. Deventer, The Netherlands: Kluwer, B.V.; 2001. p. 229–44.

Diaz G, Pardo JJ, Cambronero ME, Valero V, Cuartero F. Verification of web services with timed automata. Electronic Notes in Theoretical Computer Science 2006;157(2):19–34, http://dx.doi.org/10.1016/j.entcs.2005.12.042.

Dijkman RM, Dumas M, Ouyang C. Semantics and analysis of business process models in BPMN. Information and Software Technology 2008;50(12):1281–94, http://dx.doi.org/10.1016/j.infsof.2008.02.006.

Dumez C, Bakhouya M, Gaber J, Wack M. Formal specification and verification of service composition using LOTOS. In: 7th ACM international conference on pervasive services (ICPS 2010). Berlin, Germany: ACM; 2010. http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/NWeSP.2008.17.

Dumez C, Gaber J, Wack M. Model-driven engineering of composite web services using uml-s. In: Proceedings of the 10th international conference on information integration and web-based applications & services (iiWAS2008). ACM; 2008a. p. 395–8.

Dumez C, Gaber J, Wack M. Web services composition using UML-s: a case study. In: GLOBECOM workshops, 2008 IEEE workshop on service discovery and composition in ubiquitous and pervasive environments (SUPE'08); 2008b. p. 1–6. http://dx.doi.org/10.1109/GLOCOMW.2008.ECP.55.

Dumez C, Nait-sidi moh A, Gaber J, Wack M. Modeling and specification of web services composition using uml-s. In: International conference on next generation web services practices; 2008c. p. 15–20.

Dumez C, Gaber J, Wack M. Uml-s framework. May 2009. 〈http://sourceforge.net/projects/uml-s/〉.

Emerson E, Lei C. Efficient model checking in fragments of the propositional mu-calculus. In: Proceedings of the 1st LICS. IEEE Computer Society Press; 1986. p. 267–78.

Erl T. SOA principles of service design (The Prentice Hall service-oriented computing series from Thomas Erl).Upper Saddle River, NJ, USA: Prentice Hall PTR; 2007.

Fernandez JC, Garavel H, Kerbrat A, Mounier L, Mateescu R, Sighireanu M. Cadp—a protocol validation and verification toolbox. In: Proceedings of the 8th international conference on computer aided verification; 1996. p. 437–40.

Ferrara A. Web services: a process algebra approach. In: ICSOC '04: proceedings of the 2nd international conference on service oriented computing. New York, NY, USA: ACM; 2004. p. 242–51. http://dx.doi.org/10.1145/1035167.1035202.

Foster H, Uchitel S, Magee J, Kramer J. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In: Proceedings of the 28th international conference on Software engineering; 2006. p. 771–4.

Frappier M, Habrias H. A comparison of the specification methods. In: Software specification methods: an overview using a case study; 2006.

Gaber J, Bakhouya M. An affinity-driven clustering approach for service discovery and composition for pervasive computing. In: Proceedings of IEEE ICPS'06; 2006. p. 277–80.

Gardner T. Uml modelling of automated business processes with a mapping to bpel4ws. In: Proceeding of the 17th European conference on object-oriented programming (ECOOP); 2003.

Groote JF, Vaandrager F. An efficient algorithm for branching bisimulation and stuttering equivalence. In: Proceedings of the seventeenth international colloquium on automata, languages and programming. New York; 1990. p. 626–38.

Hoare CAR. Communicating sequential processes. Communications of the ACM 1978;21(8):666–77, http://dx.doi.org/10.1145/359576.359585.

Holzmann GJ, et al. The model checker spin. IEEE Transactions on Software Engineering 1997;23(5):279–95.

Kozen D. Results on the propositional [mu]-calculus. Theoretical Computer Science 1983;27(3):333–54.

Li J, He J, Zhu H, Pu G. Modeling and verifying web services choreography using process algebra. In: 31st IEEE software engineering workshop; 2007. p. 256–68.

Lucchi R, Mazzara M. A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 2007;70(1):96–118.

Mantell, K., 2003. From UML to BPEL. 〈http://www.ibm.com/developerworks/webservices/〉.

Mateescu R, Poizat P, Salaün G. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Proceedings of the 6th international conference on service-oriented computing LNCS 5364/2008; 2008. p. 84–99.

Martin D, Burstein M, Hobbs J, Lassila O, McDermott D, McIlraith S, et al. Owl-s: semantic markup for web services. 〈http://www.daml.org/services/owl-s/1.1/〉, 2004.

Milner R. A calculus of communicating systems.New York, Secaucus, NJ, USA: Springer-Verlag, Inc.; 1982.

Mitra S, Kumar R, Basu S. Automated choreographer synthesis for web services composition using i/o automata. In: IEEE international conference on web services, 2007. ICWS 2007; July 2007. p. 364–71. http://dx.doi.org/10.1109/ICWS.2007.47.

OASIS. Business process execution language (BPEL) 〈http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html〉, 2007.

OMG, OMG. Unified modeling language 2.2 specification. February 2009. 〈http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/〉.

Peng L, Cai C, Qiu Z, Pu G. Verification of channel passing in choreography with model checking. In: IEEE international conference on service-oriented computing and applications (SOCA); 2009. p. 1–5.

Pu G, Zhao X, Wang S, Qiu Z. Towards the semantics and verification of bpel4ws. Electronic Notes in Theoretical Computer Science 2006;151(2):33–52, http://dx.doi.org/10.1016/j.entcs.2005.07.035.

Puhlmann F, Weske M. Using the pi-calculus for formalizing workflow patterns. Lecture Notes in Computer Science 2005;3649:153–68.

Ravn AP, Srba J, Vighio S. Modelling and verification of web services business activity protocol. In: 17th international conference, TACAS 2011, held as part of the joint European conferences on theory and practice of software, ETAPS; 2011. p. 357–71.

Raymond K. A challenge to LOTOS as a formal description technique for open distributed processing. Discussion document no 23, University of Queensland Centre of Expertise in Distributed Information Systems (CEDIS). Oct 1989. 〈http://sky.fit.qut.edu.au/~raymondk/a-challenge-to-lotos-as-a-fdt-for-odp.pdf〉.

Russell N, ter Hofstede AHM, van der Aalst WMP, Mulyar N. Workflow control-flow patterns: a revised view. Technical report, BPM Center Report BPM-06-22, BPMcenter.org; 2006.

Soley R. Model driven architecture, omg white paper. 〈http://www.omg.com/mda〉, 2000.

Staines TS. Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In: ECBS '08: proceedings of the 15th annual IEEE international conference and workshop on the engineering of computer based systems. Washington, DC, USA: IEEE Computer Society; 2008. p. 191–200. http://dx.doi.org/10.1109/ECBS.2008.12.

Stefansen C. Expressing workflow patterns in ccs. 2005, unpublished.

Tan W, Fan Y, Zhou M. A petri net-based method for compatibility analysis and composition of web services in business process execution language. IEEE Transactions on Automation Science and Engineering 2009;6(1):94–106, http://dx.doi.org/10.1109/TASE.2008.916747.

Tang Y, Chen L, He K-T, Jing N. Srn: an extended petri-net-based workflow model for web service composition. In: ICWS '04: proceedings of the IEEE international conference on web services. Washington, DC, USA: IEEE Computer Society; 2004. pp. 591–9. http://dx.doi.org/10.1109/ICWS.2004.106.

ter Beek MH, Bucchiarone A, Gnesi S. A survey on service composition approaches: from industrial standards to formal methods. Technical report; 2006. doi:2006-TR-15.

van der Aalst WMP, ter Hofstede AHM, Kiepuszewski B, Barros AP. Workflow patterns. Distributed and Parallel Database 2003;14(July (1)):5–51, http://dx.doi.org/10.1023/A:1022883727209.

Vaz C, Ferreira C. Formal verification of workflow patterns with spin. 〈http://pwp.net.ipl.pt/cc.isel/cvaz/TR/CVCF3939.pdf〉, 2008.

White SA. Introduction to bpmn. IBM Cooperation, 2008–029; 2004a.

White SA. Process modeling notations and workflow patterns. Workflow Handbook 2004b:265–94.

Wohed P, van der Aalst W, Dumas M, ter Hofstede A, Russell N. On the suitability of BPMN for business process modelling. In: Business process management, vol. 4102/2006. Springer-Verlag; 2006. p. 161–76.

Wong PYH, Gibbons J. A process-algebraic approach to workflow specification and refinement. In: Software composition: 6th international symposium, SC 2007, Braga, Portugal, March 24–25, 2007, Revised Selected Papers. p. 51–65.

Yan Y, Dague P. Modeling and diagnosing orchestrated web service processes. In: IEEE international conference on web services, 2007. ICWS 2007; 2007. p. 51–9.