Formal Specification and Verification of Service Composition using LOTOS

Christophe Dumez, Mohamed Bakhouya, Jaafar Gaber and Maxime Wack
University of Technology of Belfort-Montbéliard
Belfort, France
{christophe.dumez,mohamed.bakhouya,gaber,maxime.wack}@utbm.fr

ABSTRACT

Despite the fact that several languages have been proposed for Web service composition (BPEL, WSCI, BPMN to name a few), their lack of well-defined formal semantics does not support formal analysis. The verification of Web service composition is thus a complicated task that can benefit from the use of formal methods. In this paper, an approach to specify, verify and validate the service composition using the LOTOS formal specification language is proposed. To achieve this task, we provide a translation into LOTOS for each workflow control-flow pattern. A working example is then presented to show how a service composition workflow can be specified using these patterns and validated using CADP toolkit.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Verification

Keywords

Service composition, Model-driven engineering, Workflow patterns, Process algebra, LOTOS, Formal specification and validation

1. INTRODUCTION

Web services are distributed and independent components realizing specific tasks, which can communicate with each other through message exchange. This process is called service *composition* and it usually results in the creation of a new *composite* service, which can be defined as an aggregation of elementary or composite services. Many researches focus on Web services *composition* [1, 2, 7, 22].

The process of service composition can be separated into the following tasks: (i) requirements analysis, (ii) specification, (iii)implementation and (iv) validation. Graphical languages such as BPMN [26] and textual ones such as BPEL [18] have been proposed to specify and implement Web services composition. However, their lack of well-defined formal semantics does not support formal analysis. As a consequence, the validation of Web service composition remains a complicated task. Formal methods such as Petri net or process algebra make good candidates to address this issue because of the variety of validation tools available.

This paper thus proposes to specify the service composition using the standard LOTOS [4] formal specification language. LOTOS is based on process algebra and it is extremely useful for temporal logic verification and for simulation. To support virtually any business process modeling language, we provide a direct mapping into LOTOS for each workflow control-flow pattern that identified by Aalst in [24]. Any service composition specified in LOTOS can then be automatically validated using tools such as the CADP toolkit [9]. Validation is achieved through the verification of properties expressed using temporal logic.

Section 2 provides a short survey of existing approaches to validate service composition. An overview of LOTOS is then given in Section 3. The translation into LOTOS for the main workflow patterns is then presented in Section 4. In Section 5, a working example is translated into LOTOS and validated using CADP. Finally, Section 6 draws the conclusions.

2. RELATED WORK

Several language-specific approaches focus on the validation of service composition expressed in a particular business process modeling language using formal methods. For example, rules are presented in [5] to translate Web service choreographies written in WSCI into CCS [16] and then check whether two Web services are compatible to interoperate. In [13], the semantics of WS-CDL are described in CSP [11]. A two-way mapping between LOTOS and BPEL was proposed in [10]. In [15], the semantics of BPEL were specified using π-calculus. A more exhaustive survey of formal verification for business process modeling can be found in [17].

More generic approaches involve instead the translation of higher-level workflow control-flow patterns into formal specifications. These patterns were identified and described using coloured Petri nets by Aalst [24]. Other researchers then specified these patterns using CCS [23], π-calculus [19].

In [25], a translation of a well known collection of workflow patterns were translated into Promela, which is the input specification language of SPIN verification tool [12]. In this tool, systems are described in Promela and the properties to be verifed are expressed as Linear Temporal Logic (LTL) formulas. The effectiveness of this approach in business process specification and verification was illustrated with two business process scenarios, namely Loan Request and the Travel Agency examples. 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.

In this paper, an approach based on the mapping of these patterns into LOTOS formal specification language is proposed to specify, verify and validate the service composition.

3. LOTOS: AN OVERVIEW

LOTOS (Language Of Temporal Ordering Specification) [4] 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 such as CCS (Calculus of Communicating Systems) [16].

The LOTOS behaviour operators are summarized in Table 3 where `G` refer 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` a behaviour.

Table 1: LOTOS behaviour operators

Behaviour Operator	Meaning
`stop`	inaction
`G !V ?X:S ; B`	Action Prefix
B_1 `[]` B_2	Choice
`[E] -> B`	Conditional
$B_1 \;\vert\vert[G_1,\ldots,G_n]\vert\; B_2$	Parallel composition
$B_1 \;\vert\vert\vert\; B_2$	Interleaving
`exit`	Successful termination
$B_1 \gg B_2$	Sequential composition
$P\ [G_1,\ldots,G_n]\ (V_1,\ldots,V_m)$	Process call

A more detailed introduction to LOTOS can be found in [3].

In this paper, LOTOS was chosen for the specification of service composition workflows because of (i) its ISO standardization, (ii) its high expressiveness and its support for value passing between processes (iii) its formalism and (iv) the existence of validation tools that support it such as CADP [9].

4. WORKFLOW PATTERNS TRANSLATION

In this section, we address the translation into LOTOS process algebra for most of the control-flow patterns in the original set, introduced by Aalst et al. in [24]. We also take into account their reviewed definition by Russel et al. [21]. 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 [18]. Due to space limitation, we will not consider in this paper the state-based patterns nor the cancellation patterns that belong to the original set.

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. Being a standardized formal specification language, LOTOS is an excellent candidate for checking an verifying service composition. As a matter of fact, automated validation of LOTOS specifications can be achieved using existing tools such as CADP [9].

4.1 LOTOS specific considerations

For reusability purpose, we model each workflow pattern as an independent LOTOS process. 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 (activity) as a LOTOS process and each edge as a gate. However, as explained by Raymond in [20], 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 in [20] is to use a single gate to specify all communication and to use a *communication medium* constraint process to ensure that communication only occurs along edges of the graph. This is the solution that is employed in this paper and our communication medium process is specified in LOTOS in Listing 1. As one can see, we use two gates (SEND and RECV) instead of one to model the two-way communication between the bus and the services (processes), as precognized in [6].

```
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
```

Listing 1: LOTOS code for communication bus

All services being composed are modeled as LOTOS processes, which execute concurrently and communicate through a software bus, which is also modeled by a LOTOS process (see Figure 1). 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`. 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.

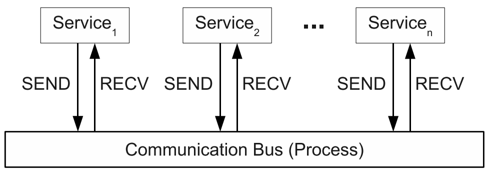

Figure 1: Architecture of the communication between LOTOS processes

4.2 Basic Control Flow Patterns

Aalst identified five basic flow control patterns, namely the **sequence**, the **parallel split**, the **synchronization**, the **exclusive choice** and the **simple merge**. These patterns capture elementary aspects of the process control. This subsection provides a definition and a rigorous translation into LOTOS for each of them.

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. Our proposal consists in having the current process (`Service1`) send a RUN message to the next process (`Service2`) via the communication bus. The next process awaits this message before starting its execution.

```
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
```

Listing 2: LOTOS translation for sequence pattern

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. We propose the `ParallelSplit` process provided in Listing 3 to model this behavior. 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 cyclical behavior 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.

```
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
```

Listing 3: LOTOS translation for parallel split pattern

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. To clarify, 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.

```
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
```

Listing 4: LOTOS translation for synchronization pattern

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 our LOTOS specification (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, our specification guarantees that only one of the output branches will be executed.

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

Listing 5: LOTOS translation for exclusive choice pattern

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).

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.

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

Listing 6: LOTOS translation for simple merge pattern

4.3 Advanced 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 our LOTOS specification (Listing 7), the choice between the output branches is nondeterministic.

```
(* 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
```

Listing 7: LOTOS translation for multi-choice pattern

In this specification, a random branch is chosen 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 issue, our `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 8. 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`).

```
(* 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
```

Listing 8: LOTOS translation for structured synchronizing merge pattern

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 (ie. `[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 9. 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). To clarity, 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).

```
(* 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
```

Listing 9: LOTOS translation for multi-merge pattern

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.

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 complete. 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 complete. This pattern can only be used after a parallel split pattern. Its LOTOS specification is given in Listing 10. 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 complete, the Discriminator process calls the Synchronization process in order to wait from the other input branches to complete before exiting.

```
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
```

Listing 10: LOTOS translation for structured discriminator pattern

4.4 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.

4.5 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 instantiate several instances of the same process and have them run concurrently using the ||| operator, provided that the number of instances is known at design time. LOTOS does not require these instances to be synchronized upon exit.

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. As explained for the previous pattern, LOTOS supports such behavior and the Synchronization specification can easily be adapted to synchronize the instances at completion.

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.

5. CASE STUDY

In this section, a working scenario is studied to show how to use the workflow patterns specifications to translate a workflow into LOTOS in order to verify and validate the service composition using CADP.

5.1 Field Emergency Response

This case study consists in an field emergency response process where a user can report an emergency. First, the system will check if the accident was already reported. If it is not the case, it will find the closest hospital to the accident. Then, it will concurrently send paramedics and a police patrol, before marking the accident as reported. This service composition workflow is depicted in a UML activity diagram in Figure 2. It is worth noting that traditional UML is used here to model the considered scenario. Such model is not exhaustive enough to support full and automatic BPEL code generation. However, we are currently working on an open source framework[1] 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 [7] or *UML for Services* 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 UML models expressiveness in the context of service composition.

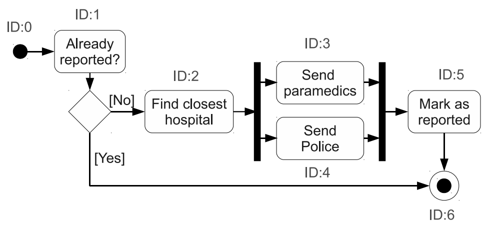

Figure 2: Emergency response workflow

This scenario composes 4 existing services:

- **HospitalLocator**: A service to find the closest hospital to the 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).

5.2 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 Figure 2 into LOTOS in order to create a process for each step of the activity (including initial and final nodes) and to assign an identifier (integer) to each of them. The identifiers are already specified in Figure 2 for a better understanding. The instantiation of these processes in LOTOS is provided in Listing 11. All service processes are executed concurrently using the ||| operator, which means that they are independent and they do not communicate directly with each other. Note however that the `|[SEND, RECV]|` operator is used the synchronize the services with the `Bus` process through the gates `SEND` and `RECV`.

```
specification EmergencyResponse [SEND, RECV]:
    noexit
behaviour
      (
      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
```

Listing 11: Processes instantiation in LOTOS

The next step in the translation into LOTOS is to identify 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 12.

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

Listing 12: LOTOS specification for Init process

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 13.

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

Listing 13: LOTOS specification for CheckIfReported process

The *FindHospital* process waits for a `RUN` message from the `CheckIfReported` process before starting concurrently the `SendParamedics` (*Id:3*) and `SendPolice` (*Id:4*) processes,

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

thus realizing a parallel split pattern. The corresponding specification is provided in Listing 14.

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

Listing 14: LOTOS specification for FindHospital process

The `SendParamedics` and `SendPolice` processes both await a `RUN` message from the `FindHospital` process (*Id:2*) before executing and finally starting the `MarkAsReported` process (*Id:5*). Only the definition for `SendParamedics` is given in Listing 15 since `SendPolice` has the exact same implementation.

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

Listing 15: LOTOS specification for SendParamedics process

The `MarkAsReported` process synchronizes its two incoming branches (paramedics and police calls) before executing and finally start the `Final` process. This behavior is defined in Listing 16.

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

Listing 16: LOTOS specification for MarkAsReported process

Finally, the `Final` process (corresponding the final node of the activity diagram) will simply merge its two incoming branches that were earlier split by an exclusive choice. As a consequence, and as stated in Listing 17, it realizes a simple merge between the branches of `MarkAsReported` and `CheckIfReported` processes.

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

Listing 17: LOTOS specification for Final process

5.3 Properties Verification

CADP toolkit provides an on-the-fly model checker for *regular* alternation-free μ-calculus formulas on Labelled Transition Systems. The temporal logical used as input language is an extension of the alternation-free μ-calculus [8, 14] with boolean formulas over actions and regular expressions over action sequences.

As a consequence, once the service composition scenario has been specified in LOTOS, the developer can describe properties on the composition using μ-calculus and proceed with their automatic verification 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 18, we provide a few macros that will be used to validate our service composition workflow. The first macro takes two actions in parameters (`A` and `B`) and verifies that action `A` necessary leads to action `B`.

```
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
```

Listing 18: Verification macros definition

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 the previously defined macros, it is now possible to define properties of the composition and verify them. One of the properties to check is that the process never sends the paramedics nor the police if the emergency was already reported. This properties can be expressed in temporal logic as follows:

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

Another important property to verify is that whenever an accident is reported, the process always sends the paramedics, unless the emergency was already previously reported. This translates into temporal logic as follows:

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

Both these properties were automatically verified using CADP evaluator and evaluated to `TRUE`, meaning that they are verified.

6. CONCLUSION AND FUTURE WORK

The formal verification of Web service composition is an important task that is not supported by current business process modeling approaches, due to their lack of well-defined formal semantics. This issue can be addressed using formal methods such as Petri net or process algebra.

In this paper, we presented an approach based on LOTOS formal specification language to verify and validate service composition expressed as a workflow. A mapping into LOTOS for each of the 20 original workflow control-flow patterns was provided to translate virtually any business process modeling language into LOTOS. A working example regarding field emergency response was also studied in order to show both how to translate it into LOTOS and then validate it. This validation was achieved through the verification of important behavioral properties expressed in temporal logic, using CADP toolkit.

Future work will involve the development of a tool for automating the translation of UML activity diagrams into LOTOS.

7. ACKNOWLEDGMENTS

This work is supported by the ASSET project (Advanced Safety and Driver Support for Essential Road Transport, 2008-2011) of the EU seventh Framework Programme (FP7).

8. REFERENCES

[1] M. Bakhouya and J. Gaber. Service composition approaches for ubiquitous and pervasive computing environments: A survey. *Agent Systems in Electronic Business, Ed. Eldon Li and Soe-Tsyr Yuan, IGI Global*, (978-1-59904-588-7):323–350, 2007.

[2] S. Ben Mokhtar, N. Georgantas, and V. Issarny. Cocoa : Conversationbased service composition for pervasive computing environments. *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 29–38, June 2006.

[3] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *The Formal Description Technique LOTOS*, pages 23–73, 1989.

[4] E. Brinksma. Information processing systems–open systems interconnection–lotos–a formal description technique based on the temporal ordering of observational behaviour. *International Standard, ISO*, 8807, 1988.

[5] J. Camara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing web services choreographies. *Electronic Notes in Theoretical Computer Science*, 154:159–173, 2006.

[6] M. A. Cornejo, H. Garavel, R. Mateescu, and N. D. Palma. 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*, pages 229–244, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[7] C. Dumez, A. Nait-sidi moh, J. Gaber, and M. Wack. Modeling and specification of web services composition using uml-s. *Next Generation Web Services Practices, International Conference on*, 0:15–20, 2008.

[8] E. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st LICS*, pages 267–278. IEEE Computer Society Press, 1986.

[9] J. C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. Cadp-a protocol validation and verification toolbox. *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, 1996.

[10] A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[12] G. J. Holzmann et al. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[13] M. Koshkina and F. v. Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *AV-WEB Proceedings/ACM SIGSOFT SEN*, 29(5), 2004.

[14] D. Kozen. Results on the propositional [mu]-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[15] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2006.

[16] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[17] S. Morimoto. A survey of formal verification for business process modeling. *Lecture Notes in Computer Science*, 5102:514–524, 2008.

[18] OASIS. Business process execution language (bpel). *http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html*, 2007.

[19] F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. *Lecture Notes in Computer Science*, 3649:153, 2005.

[20] K. Raymond. 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), http://sky.fit.qut.edu.au/ raymondk/a-challenge-to-lotos-as-a-fdt-for-odp.pdf, Oct 1989.

[21] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns : A revised view. Technical report, BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[22] D. Skogan, R. Grønmo, and I. Solheim. Web service composition in uml. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:47–57, 2004.

[23] C. Stefansen. Expressing workflow patterns in ccs. unpublished, 2005.

[24] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.

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

[26] S. A. White. Introduction to bpmn. *IBM Cooperation*, pages 2008–029, 2004.