2012 12th International Conference on Quality Software

Direct verification of BPMN processes through an optimized unfolding technique

Damiano Falcioni, Andrea Polini, Alberto Polzonetti, Barbara Re
Computer Science Division, School of Science and Technologies
University of Camerino, Camerino (MC), ITALY
Email: {firstname.lastname}@unicam.it

***Abstract*—Business process analysis is one of the most important and complex activities of Business Process Management. Business processes are typically defined by business experts which ask for graphical and user-friendly notations. Nevertheless most notations used typically lack precisely defined semantics limiting the possibility of analysis to informal approaches such as observation techniques. To support formal verification techniques it is necessary to define a precise mapping between the adopted user-friendly notation and a formal language.**

In this paper we propose a Java based verification approach for Business Processes modeled using the BPMN 2.0 standard. In particular, we defined a precise Java mapping for the main elements of the BPMN 2.0 notation. The relations among the different elements of a BPMN 2.0 specification are supported by the inclusion of specific attributes and methods in the created Java objects. The behavior of a set of interrelated objects, corresponding to a BPMN 2.0 specification, can be explored using an algorithm we defined for the purpose. Such an algorithm permits to avoid the state explosion phenomenon using an ad-hoc unfolding technique.

A plug-in for the Eclipse IDE platform has been developed. It permits to have an integrated environment in which to design a business process, to verify it, and to check the result of the verification in order to improve the business process itself. This iterative approach can continue until all the issues highlighted by the verifier are solved. The approach and the prototype have been successfully applied to real scenarios within the Public Administration domain, with encouraging results.

***Keywords*-Business Process Management, BP modelling, BP verification, Unfolding algorithm.**

I. Introduction

Business Process Management (BPM) "includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of Business Processes" [1]. Business Processes (BP) analysis is one of the most important and critical steps in the BPM life cycle. It is deeply linked with the design phase where the use of user-friendly languages is fostered by the need of involving domain and business experts. In such a context the analysis can be carried out via informal approaches such as workshops in which the interested stakeholders can express their opinions and point to possible issues in the defined BP. In some cases BP specifications can be simulated via suitable transformations/tools so that deficits in the BP model can be directly observed in a, possibly guided, run. Formal verification is one more possibility to carry on the analysis. It can be particularly useful to discover structural error and possible critical unwanted situations.

In order to apply formal verification in the BPM context, we need to derive a formal model of the BP being verified. Typically a mapping from the user friendly language to a formal language, such as Petri Nets or Process Algebra, is defined and the verification activity is carried on using the resulting formal specification. If the main advantage of such an approach is the reuse of reliable and off-the-shelf verification tools, some relevant drawbacks can be listed too. First of all any possible mapping will inevitably suffer from the limitations and constraints coming from the characteristics of the selected formal language. For such a reason the specific objective of the verification activity should also guide the selection of the formal language to be used. The more expressive the selected language the more complex will typically be the verification activity. For instance, using BPMN 2.0 as modelling language if we consider standard Petri Nets as target formalism we will have to consider that all the information about pools will be lost and no verification involving such an aspect will be possible. On the other side if we use a Process Algebra as target language, synchronization mechanisms will be introduced even when they are not strictly needed. As an effect the state explosion phenomenon will be amplified. Finally, it is often difficult to track highlighted problems to the starting high level (e.g. BPMN 2.0) model.

In order to reduce the incidence of the listed problems we propose a Java based verification approach for BPs modelled using the BPMN 2.0 collaboration diagram. The approach is based on a state evaluation technique using an optimized unfolding algorithm. In this way after the modelling phase is performed making use of BPMN 2.0, the Java model can be automatically created and the verification can be run in order to find out possible deadlocks, livelocks, and good traces. The approach reduces as much as possible the presence of interleaving among different pools and it limits branching to the control flow of single process participants. Feedbacks coming from the verification phase will be used to drive the re-engineering of the BP specfication. For such an activity the mapping has been conceived in order to have a direct tracking to the starting model.

The approach is supported by a plug-in for the Eclipse platform. The plug-in permits to have an integrated environ-

1550-6002/12 $26.00 © 2012 IEEE
DOI 10.1109/QSIC.2012.59

ment in which to design and verify defined Business Processes. The approach and the prototype have been successfully applied to real scenarios in the public administration domain, with encouraging results.

The rest of the paper is organized as follows. The next section proposes work related to our approach, whereas Section 3 presents background information. Section 4 introduces the verification approach, and Section 5 presents the developed plug-in. Section 6 describes the results obtained from the experiments conducted. Concluding remarks and opportunities for future developments are discussed at the end of the paper.

II. Related Work

With respect to work related to the approach we propose we can follow two different investigation domains. On the one side we refer to informal or semi-formal languages for verification, on the other side we discuss different approaches for verification optimization.

A. User-friendly Verification

Few proposals have already addressed the problem of defining a formal semantics for BPMN. Wong and Gibbons have encoded BPMN in CSP [2]. In particular, they used the language and the behavioural semantics of CSP to denote the different pools in terms of CSP processes and actions. Starting from the idea of Wong and Gibbons, we have defined our mapping in CSP and created an integrated environment to verify quality properties of a BPMN model relying on PAT Model checker [3]. Another mapping to provide a formal semantics for a subset of BPMN elements was made by Dijkman et al. in [4]. The authors also implemented a translation tool from BPMN to Petri Nets. The BPMN model has to be created with ILOG BPMN Modeler and can be verified using ProM. Another interesting proposal has been introduced by [5] and [6], where the YAWL formal language is used to verify properties over BPMN. YAWL is a formal workflow specification language designed to overcome Petri Nets limits and to support more specific BP patterns [7]. In [8] the same authors present a mapping to BPEL but just in order to permit the direct execution of the model. In [9] the authors define an extensible semantical framework for business process modeling notations. Finally, one more approach for model execution has been discussed using a mapping from BPMN to agents [10].

Main differences between the approach we propose and the mentioned work refers to the mapping and resulting tool. On the one side most of the solutions in the literature implement a mapping to a formal language. They generally have to face problems of state explosion and they respect/implement synchronization policies according to the expressibility of the target language. In our case the approach is directly guided by BPMN 2.0 semantics and doing so we reduce the complexity of the verification. On the other side, in most of the cases the solutions do not provide an integrated environment and most of them ask for the installation of tool chains that are not so easy to use for most of the BP experts.

B. Approaches for Optimization

Unfolding is a widely studied partial order reduction technique first introduced by McMillan [11]. It has evolved in many papers like [12] where the authors propose an algorithm applied to Petri Nets that generates a minimal complete prefix of the unfolded net. In [13] the authors optimize the unfolding algorithm acting on the prefix cutoff point. In [14] an efficient unfolding algorithm for checking safety properties of unbounded Petri Nets has been provided. Unfolding algorithms are integrated in many tools such as PEP [15], for Petri Nets that also give the possibility to choose McMillan's or Esparza's algorithm, or SPIN [16] for Process Algebra.

In our case we extend the well-known unfolding algorithm in order to make it possible its application on a BPMN 2.0 model, so to guarantee the support of direct verification based on Java models.

III. Background

A. BPM and Process Modelling

We refer to a BP as "*a collection of related and structured activities undertaken by one or more organizations in order to pursue some particular goal. Within an organization a BP results in the provisioning of services or in the production of goods for internal or external stakeholders. Moreover BPs are often interrelated since the execution of a BP often results in the activation of related BPs within the same or other organizations*" [17].

BPM supports BP experts providing methods, techniques, and software to model, implement, execute and optimize BPs which involve humans, software applications, documents and other sources of information [1].

Recent work has shown that BP modeling has been identified as a fundamental phase in BPM. The quality of BPs resulting from the BP modeling phase is critical for the success of an organization. However, modeling BPs is a time-consuming and error-prone activity. Techniques which can help organizations to implement high-quality BPs, and to increase process modeling efficiency, has become an highly attractive topic both for industries and for the academy. Certainly many different commercial tools have been developed to support BPM. Nevertheless for what concerns the modeling phase they mainly provide support for BP editing and syntactical analysis. To the best of our knowledge none of them introduces and supports formal verification techniques.

Different classes of languages to express BPs have been investigated and defined. There are general purpose and standardized languages, such as the BPMN 2.0 [18] or the

Event-Driven Process Chain [19], and many others. There are also more academic related languages, being the Yet Another Work-flow Language [7] based on Petri Nets [20] the most prominent example. Among the listed languages there are several differences. These are related to the level of rigor, going from semi-formal, with a precise syntax and with semantic given in natural language, to formal languages for which the semantic is provided thanks to well founded mathematical theories.

In our work we refer to BPMN 2.0 [18], an Object Management Group (OMG) standard. This is certainly the most used language in practical context also given its intuitive graphical notation . The standard specifies three different views: process, choreography and collaboration. The process view refers to private and public BP. Using a private BP model intra-organizational Business Processes are represented. At the same time in public BPs the interactions between two different private BPs or participants is modelled. The Choreography view permits to represent the expected behavior between interacting participants. Finally, the collaboration diagram, that is our reference model, is used in order to have a complete representation both of internal processes as well as of the message exchange structure. In particular, the following BPMN 2.0 elements are the most commonly used in this diagram specification.

Pools are used to represent a participant or an organization involved in the BP. They contain the private BP and related elements as reported in the following.

Tasks are used to represent an action to perform that produces a result. Different types of tasks exist, just to cite a few we refer to manual, human and service. Tasks are graphically drawn as rectangles with rounded corners.

Events are used to represent something that can happen. In particular, "start events" represent the points in which the BP starts, "intermediate events" represent something that can happen during the BP execution, like time exceeding a deadline or the reception of a message, and finally "end events" are raised when the BP terminates. Different types of events can be introduced starting from these three main categories. Events are graphically drawn as circles.

Gateways are used to manage the process flow on choices and parallel activities. Different types of gateways are available, the most used ones are exclusive and parallel [21]. An exclusive gateway gives the possibility to describe choices in the BP and a single output path can be activated each time the gateway is reached. Parallel gateways have to wait all their input flows to start and then all the output paths are started in parallel. Gateways are graphically drawn as diamonds.

B. Unfolding

Unfolding is a technique of partial order reduction. It is widely applied to Petri Nets and Process Algebra in order to reduce state explosion problems occurring during verification activities. Unfolding has proved to be a good approach on deadlock detection. It is based on the concept that some decidability problems can be reduced to reachability problems (proved to be decidable in papers like [22]). In order to solve the problem, a prefix of the model is built with the objective to cover all the reachable states.

Unfolding of a model can in fact be infinite, but McMillan [11] identified the possibility of building a finite prefix of the net which could give us enough information to solve several problems. This is made ending the prefix in a specific point called "*cut-off*".

The concept of configuration is introduced, it is used to identify the current status of the model referred to a specific path, during the unfolding. So the key to terminate the unfolding is to identify configurations states acting as cut-off points. These must have the following property: any configuration containing a cut-off point must be equivalent (in terms of final states) to some configurations containing no cut-off points. From this definition, it follows that any successor of a cut-off point can be safely omitted in the unfolded model, without sacrificing any reachable state of the original model. This means that a cut-off point is reached in a configuration only when the current final state of the configuration is already present as the previous state in the same configuration. So this current final state becomes our cut-off point. If we do not stop the prefix of the unfolded model in this cut-off point, we are sure that this state will happen again infinitely often in the future because it has already happened once in the past.

Once the prefix has been constructed, the deadlock detection problem is reduced to a graph search. This problem is NP-complete as shown by [11]. However this problem is readily solved in practice even for very large unfoldings. Problems, such as liveness and deadlock-freeness are recursively equivalent to reachability, so that they are also decidable [23]. Approaches are also proposed in literature to verify temporal logics through unfolding [24].

There are several facts that are worth mentioning about the unfolding. The first is that the unfolding is an acyclic graph, defining a partial order on its nodes. Second, branching occurs naturally in the structure where the actual choice occurs . The advantage is that we can explore the state space of concurrent systems without considering all possible interleaving of concurrent events. This becomes particularly significant if in the model there is a cycle.

In this work we base our verification approach on an optimized unfolding algorithm that exploits advantages provided by the use of BPMN 2.0. In particular, we reduce the state explosion problem reducing the interleaving among BPMN 2.0 elements, in a way that:

- synchronizations happen only on parallel gateways in a pool and messages exchanged between pools;
- branching in the configuration tree happens only with reference to exclusive gateways.

IV. BPMN 2.0 Direct Verification

The approach we have defined supports Java-oriented verification on BPMN 2.0 collaboration models. Starting from the results of a BP design phase we explore the model and we identify paths from BPMN 2.0 start events reaching a termination condition. Path definitions rely on synchronization and branching rules that are affected by the semi-formal semantics of the following BPMN 2.0 elements: pool, sending and receiving tasks, and parallel and exclusive gateways.

In the collaboration diagram each pool encapsulates all the private process elements, and a pool can interact via message exchanges with other participants. The approach we propose uses such encapsulation in order to eliminate interleaving among the elements in different pools that do not require synchronization. Synchronization is needed just for those tasks and events receiving or sending a message. Moreover, synchronization is observed in a pool for parallel gateways with references to input flows. Exploring the BP model, branching occurs only with reference to exclusive gateways. This gives us the possibility to explore alternative paths. The specific characteristics of BPMN 2.0 elements suggests to conceive an optimization of the McMillan unfolding algorithm [11] specifically adapted to work on BPMN 2.0 models. In order to do that, we reviewed key concepts such as configuration and cut-off points in a BPMN 2.0 context.

In Figure 1 we report a simple collaboration model in order to make clear to the reader relevant aspects of the approach we propose. The BP is composed by two participants (pools) exchanging a message, the participant A internally decides how to behave according to the evaluation of the choice statement.

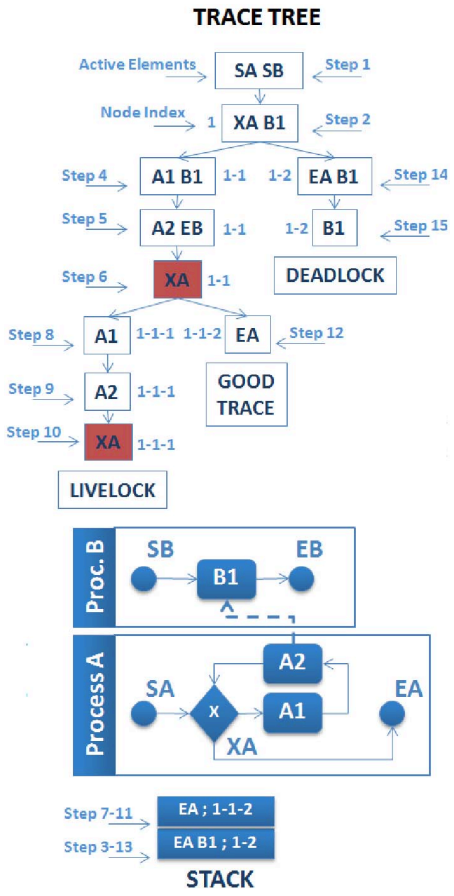

Figure 1. Running Example

A. Configuration Definition

A configuration represents a possible partial run of the BP model under study. It is used to identify its current status during the execution of the unfolding algorithm. In Petri Nets the configuration contains the marked transitions. With reference to our approach a configuration refers to a specific path in the BP model. It differs from Petri Nets because in our case a configuration contains all the activated BPMN 2.0 elements included in the collaboration diagram together with their status. In our approach all the configurations are stored in a tree structure. This is inspired from the coverability tree for Petri Nets introduced by Karp and Miller [25]. The coverability tree is an abstraction of the reachability tree which is precise enough to decide some important problems like coverability, boundedness and place boundedness problems for Petri Nets, and that were shown to be decidable [25]. Each node of the tree contains the BPMN 2.0 elements in the collaboration diagram that are currently active in terms of diagram exploration.

The application of the unfolding algorithm in BPMN 2.0 satisfies the following conditions:

- If an element is in the configuration, then all of its ancestors are in the configuration too (a configuration is downward closed);
- A configuration cannot contain two BPMN 2.0 elements in conflict, meaning that both are inputs from the same exclusive gateway.

The status of the BPMN 2.0 element is mainly relevant for what concerns those tasks sending and receiving messages. In particular, we accept as valid only those messages coming from tasks sending or receiving messages from an ancestor node in the configuration. We choose to evaluate such status at run-time instead of saving it in the configuration tree, in order to reduce memory consumption. As an example we refer to the steps 9 and 15 of the BPMN 2.0 exploration in Figure 1. During step 9 a message is sent from task A2 to B1 task, but it is never consumed because the task B1 has already been executed. As an effect during the backtracking (in step 13) B1 has a message that could be consumed but such messages do not come from an ancestor node of the current execution status so that such message is invalid and the B1 task remains blocked.

In order to correctly do the backtracking activities and

complete the exploration of the tree we use a stack. It contains the tree nodes that have to be reloaded during backtracking and the index of the tree structure referring to the position in the tree of such nodes. The index is in the form *1-n-n-...-f*, parsed from left to right, where:

- *1* represents the root node and all the successive nodes that do not split the tree;
- - separates between splitting nodes;
- *n* is a number representing the successive node chosen from the current one and it includes all the successive nodes after that choice till the next one;
- *f* represents the last branching node index and it is the same for all the successive nodes until the path is ended.

This index structure makes easy the run-time evaluation with respect to message synchronization, in particular when there are messages sent from a task which is an ancestor node with respect to the current configuration.

B. Livelock Identification (Cut-off Points)

The approach we propose exploits the configuration tree in order to find out cut-off points to identify livelock situations. A path is in livelock iff the current node is already observed during the exploration phase in one ancestor node of the configuration tree. In that case there are no cycles in the BP model, and then it is obvious that cut-off points can not be observed. In the configuration tree in Figure 1 the node referring step 10 is an example cut-off point. As a matter of fact the node involves the element XA and it is already observed in the ancestor node with reference to step 6.

C. Deadlock Identification

The approach follows the BPMN 2.0 termination paradigm in order to find a deadlock. In BPMN 2.0 a BP terminates when end or termination events are reached during the process execution. The approach we propose adapts the unfolding algorithm in order to remove end events from the configuration each time they occur. The path results in deadlock iff in the current configuration there are only blocked elements (i.e. tasks or events waiting for a message and a parallel gateways waiting for incoming flow that will never arrive). In the example in Figure 1 we observe a deadlock in step 13 when after backtracking, due to the gateway XA, the end event EA is reached and removed and the task B1 is blocked waiting for a message that will never arrive.

D. Good Trace Identification

The application of the approach returns with a good path when removing all the end events from the configuration then the tree node results to be empty. A node can be also emptied if a termination end event occurs. This means that the path is good and the process execution will stop correctly.

E. Proposed Algorithm

The idea at the base of the proposed algorithm is to explore the model till it is possible. In other words this means that the elements in a pool are consumed and substituted by the next one in the process flow, till synchronization or branching elements are reached. The reported pseudocode provides a simplified version of the algorithm and we provide here some detail on how the algorithm works.

```
1  confTree.initialize(StartEventSet, treeindex.inizialize())
2  currConfNode = confTree.current()
3  while (currConfNode != null){
4   while (currConfNode != empty AND !deadlock AND !livelock) {
5     foreach ele in the currConfNode then {
6       if ele is terminationEvent then currConfNode.erase()
7       if ele is endEvent then currConfNode.remove(element)
8       if ele is unblocked then currConfNode.sub(ele.allnexts)
9       if ele is blocked then skip
10    }
11    if currConfNode contain only exGat and blockedEl then {
12      foreach exGat in the currConfNode then {
13     currConf.branch
14        manage.firstPath
15        stack.Push(allOtherPaths)
16      }
17    }
18    checkDeadlock(currConfNode)
19    checkLivelock(currConfNode, confTree)
20  }
21  currConfNode = Stack.PopElement
22  treeindex = Stack.PopIndex
23 }
```

1. In the initialization step the algorithm inserts in the configuration tree root the list of all start events available in the collaboration diagram (line 1) for the BPMN 2.0 model under study. In this phase also the tree index is initialized.
2. The algorithm continues considering each BPMN 2.0 element in the collaboration model under study. This means that for each possible path (line 3) starting from the initial configuration, the path is explored till it is possible (line 4).
3. The elements in each node of the configuration tree are considered one after the other (line 5) according to the following rules.

1) If there is at least one termination or error end event then it is executed and all the other elements in the tree node are removed (line 7).
2) If the element is an end event then it is executed and it is removed from the node (line 8).
3) If the element is different from parallel and exclusive gateways and from tasks receiving a message it is consumed and the algorithm substitutes the elements with its successors providing a partial configuration run of the model (line 9). Such substitution is done till possible.
4) If the element is a parallel gateway or a task receiving messages it waits for all the incoming flows or messages. As soon as the synchronization occurs the algorithm substitutes the element with the following elements till possible (line 10).
5) If the element is an exclusive gateway and eventually all the other elements in the node are blocked elements, then the configuration will include different branches (line 13 - 19). The exploration of the first branch goes on and all the other branches are pushed in the stack with the tree index. They will be reloaded

from the stack as soon as the first path exploration terminates.

4. Each exploration path terminates considering one of the following cases:

1) If the current configuration node is empty the exploration of the path terminates with success. This can occur in the case a) and b) of the previous item list.
2) If all the elements in the current configuration node are blocked due to synchronization then the path is in deadlock (line 20).
3) If all the elements available in the current configuration are already observed in one ancestor node in the tree, then cut-off is applied. The path under examination results in livelock (line 21).

5. When an exploration of a path terminates then another is loaded from the stack if it is available and in this case backtracking occurs.

6. Considering that paths and elements in a collaboration diagram are available in a finite number and cycles are managed via cut-off, the algorithm terminates when the stack is empty and all the exploration paths terminate with deadlock, livelock or with success.

According to the proposed algorithm the exploration of the simple BP in the right side of Figure 1 produces the trace tree presented in the upper part of the same figure. The exploration results in one livelock, one deadlock and one good trace. The couple SA and SB composes the tree root (step 1) respectively they are the start events of processes A and B. Both the start events are not blocked, so that they are immediately substituted with their successors. When the exploration reaches the gateway XA it is splitted in two exploration paths (step 2). The first partial exploration under exam considers the path A1, A2 in process A and B1, EB in process B (step 4-5). At this time the elements in process B are terminated and the end event EB is removed (step 5). One more branch occurs considering the second evaluation of gateway XA in the process A (step 6). Two cases are possible if the process A consumes the element EA the exploration terminates (step 12) and the path is completed with success, while if the path A1, A2 is explored again (step 8-9) the process A will never terminates and the path is in livelock. Back to the first branching (step 2) the process terminates with a deadlock due to the task B1 that is waiting for a message (step 15). Such a message will never arrive. As a matter of fact process A is already ended (step 14) when the EA end event is removed.

The proposed trace tree in Figure 1 shows all the steps. In practice what we store in the tree are just the branching nodes and the rest is evaluated at run time, this gives us the possibility to optimize memory consumption.

The starting point to construct and demonstrate termination of the algorithm is the Petri Nets formalism, due to some commonalities shared with the BPMN 2.0 notation. Such similarities are particularly significant with reference to a Petri Nets extension research proposal [26], and covering decidability problems. In particular, the extension introduces in the Petri Nets formalism a construct similar to BPMN 2.0 pool. Modular analysis is used so to analyse the behaviour of the Petri Net without explicitly constructing the full reachability graph. Only the so called synchronisation graph between the modules is constructed. The synchronisation graph includes the external moves, i.e., moves where several modules participate. All internal moves are hidden. The idea is that the possibility to walk trough the BP, without considering all possible interleavings between elements of different pools, not involved in synchronization, is equivalent to merge all these elements into one module that hides internal steps. So in modular Petri Nets our BP can be seen as a normal Petri Net where all the nodes that do not represent consecutive exclusive gateways or tasks or events receiving messages are grouped together in a module. The main advantage is that this grouping does not require a pre-analysis step but is made on-the-fly without loosing the meaning of the model during the verification.

F. Handling False Positive Deadlock Detection

Thanks to the BPMN semantics we can also exclude some kind of deadlock optimizing the efficiency of our approach.

- **Deadlock due to message start events should not be considered.** In BPMN it is usual to have a pool that will start only on specific conditions expressed by others pools. A message start events is observed in these cases. Indeed waiting pools that never start due to waiting messages should not be considered in deadlock status.

- **Deadlock caused by a choice before synchronization messages can be avoided.** BPMN analysis shows that many deadlocks are due to the impossibility to bind choices between different pools. This is due to the fact that in BPMN, exclusive gateways take their choice on the basis of data values that usually cannot be automatically evaluated without user interaction. Verification will thus result in a number of false positive deadlock situations that in reality, using data, will never occur. To better understand the case we refer to the example in Figure 2. Here we are not interested in all possible interleavings of XA and XB choices such as the cases in which XA go on A1 and XB go on B2 or XA go on A2 and XB go on B1. Nevertheless the identification of the possible deadlock can help the designer in the selection of the right condition. To avoid the possible occurrence of these situations we have integrated in the verifier the possibility to bind path choices of different gateways. This means that if an exclusive gateway performs a

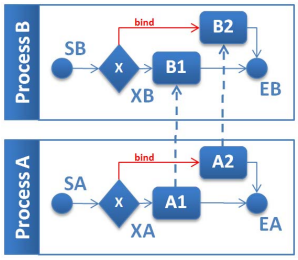

Figure 2. Path binding example

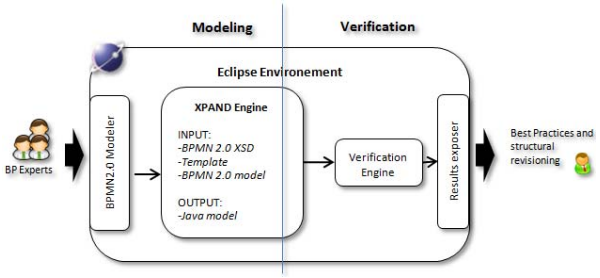

Figure 3. Tool Architecture

specific bounded choice, then all the gateways that have an output path bounded, have to take it. In this way we do not manage data, but evaluate the model as if the data were available. The information about bounded paths can be specified in the design phase by the BP experts using standard extensions of BPMN 2.0.

V. Tool Design and Implementation

A. Overview

The approach we propose is supported by a plug-in for the Eclipse platform permitting to have an integrated environment in which to design and verify BP models. The plug-in architecture is shown in Figure 3.

The plug-in integrates the Eclipse BPMN 2.0 SOA Modeller[1] and is built over the BPMN 2.0 meta-model defined using the Eclipse Modeling Framework (EMF)[2], which has been defined respecting the official XSD schema for the standard. The plug-in also integrates the XPAND[3] Eclipse engine. This is a language specialized for code generation which is based on EMF models and on the definition of transformation templates. This means that we are able to directly generate Java artefacts starting from a BPMN 2.0 model. The template specifies what to do when a given graphical element of the BPMN 2.0 notation is found in a defined BP model. Thanks to the XPAND engine the Java model of the BP is returned in output, so that it is possible to successively manage the model as Java objects. Verification runs on such a model implementing the algorithm. This is supported by the verification engine that gives back information about deadlocked, livelocked and good traces in a textual way. Information about the number of nodes checked, memory and time used are also shown. The engine has been built trying to minimize memory consumption and algorithm efficiency so that the Java Virtual Machine (JVM) interpretation process maximize performances.

The developed prototype can be downloaded from the repository hosted by Sourceforge[4].

B. From model to code

The Java formal semantics for the BPMN 2.0 model is given using the XPAND Engine that creates the appropriate Java code for each element in the BPMN 2.0 model.

In order to deploy the template different steps have been carried out. The collaboration diagram is represented by a Java class named *Environment*. It collects all the instances of the *Process* Java classes representing each single pool in the diagram. This is done by the *addProcess* method of the class *Environment*. At the same time the *Environment* allows to define the binding path in order to eliminate false positive deadklock detection using the *addPathbinding* method.

With reference to the following code fragment related to the collaboration diagram in Figure 1 we create the `env` *Environment* and two processes `ProcessA` and `ProcessB`. Both are added to the *Environment* using the *addProcess* method. *addPAthbinding* method is not used in this specific case.

```
1  Environment env = new Environment ( );
2  Process ProcessA = new Process("ProcessA");
3  Process ProcessB = new Process("ProcessB");
4  env.addProcess(ProcessA);
5  env.addProcess(ProcessB);
6  ...
```

Considering the BPMN 2.0 elements in the model the Java class *PObject* is also instantiated. *PObject* can be typed in order to represent each category of BPMN 2.0 element (i.e. start, gateways, events, etc.). For each type specific method suitable to support verification are introduced. They follow the semantics of BPMN 2.0. We refer to *setNext* in order to define the flow of the BP and *sendMsg* to specify that such *PObject* is sending a message that can be defined with the method *setMsgToSend*. They are the most significative methods. As soon as the *PObject*s are created they are added to the specific *Process* using the function *addPObject*.

With reference to the following code fragment referring to the elements in `ProcessA` in Figure 1 we create SA, XA, A1, A2, B1, EA as PObject with START, G_EXCL, TASK, TASK, TASK, END as different types respectively. For each element we set the next element in the diagram using the *setNext* function and for the task sending a message such as A2 we set such message using *setMsgToSend* and than we send it with the function *sendMsg*. Finally the objects are added to the *Process*.

[1]http://www.eclipse.org/bpmn2-modeler/
[2]http://wiki.eclipse.org/MDT-BPMN2
[3]http://wiki.eclipse.org/Xpand
[4]https://sourceforge.net/projects/cowslip

```
POBject SA = new PObject(PObject.START);
POBject XA = new PObject(PObject.G_EXCL);
POBject A1 = new PObject(PObject.TASK);
POBject A2 = new PObject(PObject.TASK);
POBject B1 = new PObject(PObject.TASK);
POBject EA = new PObject(PObject.END);
...
SA.setNext(XA);
XA.setNext(A1);
XA.setNext(EA);
A1.setNext(A2);
A2.setMsgToSend(...);
A2.sendMsg(B1);
...
ProcessA.addPObject(XA);
ProcessA.addPObject(A1);
...
```

C. Verification into practice

Once the *Environment* is completed with all the relevant information about the model, verification parameters are set and the path exploration can start running the proposed algorithm. This process starts calling the *runWithoutThreads* method of the *Environment* class.

```
env.runWithoutThreads(Environment.MODELCHECKING_MODE);
```

As you can see from the code above the method requires a parameter to specify the operating mode. The operation mode that we implemented in our approach is *.MODELCHECKING_MODE*. Anyway the transformation we implemented gives also the possibility to execute the resulting Java code in simulation mode just setting this parameter to *Environment.SIMULATION_MODE*. The simulation mode differs from the verification mode only in the branching phase of the algorithm. During this phase just one path is chosen randomly or depending on predefined priorities and the stack for backtracking activities is never used. This method executes the process in verification or simulation mode without the use of Java Threads, so in a strictly sequential manner. A simulation mode can also be executed in a more realistic way using parallelization with Java Threads. The thread enabled mode can obviously be activated just in simulation mode and not for verification purpose.

```
env.setmodelCheckerMode(false); Thread envThread=new Thread(env);
envThread.start();
```

The listing which follows referes to a pseudo-code version of Environment method implementing the verification algorithm we propose.

```
1  public configurationTree runWithoutThreads(int mode){
2   currConfNode = diagram.getAllStartObject();
3   treeIndexPos = InitializeTreeIndex();
4   while(currConfNode != null){
5    while( !currConfNode.isempty() && !isdeadlock && !islivelock) {
6     Foreach( PObject ele in currConfNode){
7      if (ele.type==TerminationEvent)
8       currConfNode.erase();
9      if (ele.type==EndEvent)
10      currConfNode.remove(ele);
11     if (ele.status==UnBlocked)
12      currConfNode.substitute(ele, ele.allNexts());
13     if (ele.status==Blocked)
14      Skip();
15        if (mode==Environment.SIMULATION_MODE)
16         if (ele.type==Exclusive)
17          currConfNode.substitute(ele, ele.chooseANext());
18    }
19    if (mode==Environment.MODELCHECKING_MODE){
20     if (currConfNode.contain(Exclusive) && !currConfNode.contain(unblockedEl)) {
21      configurationTree.add(currConfNode,treeindex);
22      List ExclusiveList = currConfNode.getAll(Exclusive);
23      List AlternativeNodes = currConfNode.cartesianProduct(ExclusiveList);
24       Foreach ( Node in AlternativeNodes) {
25        configurationTree.add(Node,treeindex.getUpdated());
26        stack.Push(Node)
27       }
28      }
29     }
30     isdeadlock=checkDeadlock(currConfNode);
31     islivelock=checkLivelock(configurationTree, treeindex);
32    }
33    currConfNode = Stack.PopElement();
34    treeindex = Stack.PopIndex();
35   }
36  }
```

From the code above you can notice that the switch between verification and simulation without thread affects only how exclusive gateways are managed (line 15 and 19). Tasks, Intermediate events and ParallelGateways are all managed on line 11. Indeed they have the same BPMN 2.0 behavior on output flow, and are the only ones reaching a blocked status. It is also interesting to notice that gateways are evaluated only when the current configuration status contains only exclusive gateways and eventually some blocked elements (line 20) as previously described. This is necessary to guarantee that all possible paths will be reached. The method returns the filled configurationTree containing all the information about explored path and their final status (successfully ended, livelocked or deadlocked) and all relevant statistical informations. From the pseudo-code reported in the previous section it is clear that the configuration tree is updated only on branching, so results can easily be shown on the model, marking each exclusive gateway output in relation to the information present on every node of the resulting configuration tree.

VI. Case Studies

The approach and the corresponding implementation have been experimented in the Public Administration domain and used by domain experts in the field. In particular, the tool has been used to model, verify and improve three different business processes deployed within a regional office of the Italian home affairs ministry. All the BPs are examples of inter-organizational BPs asking for several interactions among different Public Administration offices. In particular, the considered services are:

Family reunion – this is a service available for people legally residing in Italy which can apply on behalf of their relatives (spouse, depending parents, children less than 18 years old) for the purpose of family reunion and only after having provided evidence of their status with respect to "sufficient" income and a permanent address;

Grant citizenship – this is a service used to ask for Italian citizenship by a foreigner or stateless person who has married an Italian citizen or who is continuously residing in Italy for more than three years;

Bouncer registration – this is a service used to register bouncers to permit their activity within public places.

The first and the second service require complex and inter-organizational BPs and they are in place for several years now, therefore can be considered deeply tested. To

Service	Pools	Activities	Decision Points	Message Flow	Execution Time (min)	Nodes in the Tree	N. of Deadlock	N. of Livelock	N. of Good traces
Family reunion	8	131	29	36	30	253	40	8	0
Grant Citizenship	11	168	42	62	17	123	18	21	15
Buoncer Reg.	6	40	14	17	6	187	25	24	1

Table I
EXPERIMENTAL RESULTS FOR THE FIRST ITERATION (INCLUDING FALSE POSITIVES)

give a quantitative indication in 2010 the Prefecture of Ancona (the capital city of Marche Region, in Italy) received 469 applications for family reunion and 760 applications for granting citizenship. For what concerns the bouncer registration service, even if it presents a simple scenario, we choose it because its deployment is still on-going.

In Table I we report the data resulting from the first round of conducted experiments. Several structural problems in terms of deadlocks and livelocks were observed. BP and domain experts involved in the modelling activity confirmed most of the highlighted issues, and thanks to an iterative approach of BP modelling, verification, and improvement the tool helped in the derivation of higher quality BPs.

As for any approach using verification techniques it is important to check whether the state explosion phenomenon could hinder its applicability to real case studies. In our case we experimented with the three real processes already discussed. For all the considered processes relatively small number of nodes in the tree were generated. In particular, the experiments we have conducted using a desktop PC equipped with a Core 2 Duo 2,20 GHz and 4GB RAM, have highlighted that defined BPs can be checked with respect to deadlocks and livelocks in less than 30 minutes, for the most complex BP scenario. Moreover the most complex BP generated around 253 tree nodes for a total of around 50 possible paths explored. This data seems to support the idea that in the current status (i.e., complexity of BP in the e-government domain, mapping we have defined and quality properties to be checked) the approach is applicable in real scenarios and can be a useful support for BP designers. The application of our approach has been really useful to avoid design errors resulting from the complexity of the considered PA scenarios. In particular, we were able to highlight several issues with the defined BP.

As could be expected the main issues are hidden within exceptional behaviours. An interesting result refers to the fact that similar "bad traces" could be observed within different BP specifications. This could lead to the identification of a list of risky "anti-patterns" interactions so to provide indications for improving future BP modeling. In particular, in all the different BP we could detect livelock conditions when a document is required and it is not properly compiled, so it needs further integrations. In the first phase of the Bouncer Registration a livelock occurs when the prefecture requires a document from the PA Manager, that is not properly compiled. An example of deadlock occurs when there is an exclusive gateway in which at least two of its output flows will converge, after several steps, into a parallel gateway. This means that the parallel gateway does not start until all its input flows are consumed but this will never happen. This was the case in the grant citizenship process model with reference to the request of the "nulla-osta" (clearance) from the prefecture.

One more problem occurs in the Bouncer Registration process model: a deadlock involving the PA manager. It can happen that the PA manager waits for a document from the prefecture that will never arrive since the prefecture already has a version of the document and it proceeds without considering the PA Manager. This deadlock does not have a huge impact within a human driven scenario where the civil servant will adapt to the situation possibly dynamically introducing a communication between offices not foreseen by the BP specification. Obviously much different is the case of a scenario driven and supported by IT systems. In most of the cases the PA Manager has to provide automatically the document and the BP results in resource starvation. In the grant citizenship BP model an example of deadlock occurs in the whole BP when livelock is observed in one of the involved pools. It happens in the last step when the municipality waits for a call from the prefecture. In this case if the request is not approved, due to document mismatching, a livelock occurs that also results in the deadlock of the municipality that still waits for the prefecture call.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented an approach for direct verification of BP specified using the BPMN 2.0 standard. The approach intends to support BP and domain experts to easily check and improve their model. For such a reason particular attention has been devoted to the possibility of backtracking possibly highlighted issues in a user-friendly way. The approach is based on an adaptation of the unfolding exploration technique to the specific characteristics of BPMN 2.0. In particular, the exploration reduces the required space and time thanks to a more efficient management of the interleaving among different activities, taking into account the characteristics of a BPMN 2.0 model and of pools, parallel and exclusive gateways, and tasks sending and receiving messages within the model.

The approach is supported by a plug-in for the Eclipse IDE. The result is an integrated environment in which to model, verify, and improve BP definitions. The approach had been applied to real scenarios in the Public Administration

domain with encouraging results. In particular conducted experiments permitted to highlight the occurrence of "anti-patterns" i.e. of general ways of organizing interactions among offices that could easily lead to structural problems. Indeed the identification of possible anti-patterns in different domains seems one of the most interesting "side-effects" of the approach. We would like to further explore such an aspect with the application of the approach to other domains. Finally we plan to extend the mapping considering all the constructs foreseen by the BPMN 2.0 specification.

ACKNOWLEDGEMENTS

The work reported in this paper has been partially supported by the European Project FP7 IP 257178 - CHOReOS.

REFERENCES

[1] M. Weske. (2007, Nov.) Business process management concepts, languages, architectures.

[2] P. Y. H. Wong and J. Gibbons, *A Process Semantics for BPMN*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5256, ch. 22, pp. 355–374.

[3] F. Corradini, A. Polini, A. Polzonetti, and B. Re, "Business Processes Verification for e-Government Service Delivery," *Information Systems Management*, vol. 27, no. 4, pp. 293–308, Oct. 2010.

[4] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Inf. Softw. Technol.*, vol. 50, no. 12, pp. 1281–1294, Nov. 2008.

[5] J. Ye, S. Sun, L. Wen, and W. Song, "Transformation of BPMN to YAWL," in *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, ser. CSSE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 354–359.

[6] G. Decker, R. Dijkman, M. Dumas, and L. G. B. nuelos, "Transforming BPMN Diagrams into YAWL Nets," in *Proceedings of the 6th International Conference on Business Process Management*, ser. BPM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 386–389.

[7] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245–275, Jun. 2005.

[8] C. Ouyang, M. Dumas, W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, and J. Mendling, "From business process models to process-oriented software systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 1, pp. 1–37, Aug. 2009.

[9] E. Börger and B. Thalheim, "Advances in software engineering," E. Börger and A. Cisternino, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. A Method for Verifiable and Validatable Business Process Modeling, pp. 59–115.

[10] H. Endert, B. Hirsch, T. Küster, and S. Albayrak, "Towards a mapping from BPMN to agents," in *Proceedings of the 2007 AAMAS international workshop and SOCASE 2007 conference on Service-oriented computing: agents, semantics, and engineering*, ser. AAMAS'07/SOCASE'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 92–106.

[11] K. L. McMillan, "Symbolic model checking: an approach to the state explosion problem," Ph.D. dissertation, Pittsburgh, PA, USA, 1992.

[12] J. Esparza, S. Römer, and W. Vogler, "An Improvement of McMillan's Unfolding Algorithm," *Form. Methods Syst. Des.*, vol. 20, no. 3, pp. 285–310, May 2002.

[13] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten, "Analysis of Petri Nets by Ordering Relations in Reduced Unfoldings," *Form. Methods Syst. Des.*, vol. 12, no. 1, pp. 5–38, Jan. 1998.

[14] P. A. Abdulla, S. P. Iyer, and A. Nylén, *Unfoldings of Unbounded Petri Nets*. Springer Berlin Heidelberg, 2000, ch. 37, pp. 495–507.

[15] S. Melzer, S. Römer, and J. Esparza, "Verification Using PEP," in *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, ser. AMAST '96. London, UK, UK: Springer-Verlag, 1996, pp. 591–594.

[16] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[17] A. Lindsay, D. Downs, and K. Lunn, "Business process – attempts to find a definition," *Information and Software Technology*, vol. 45, pp. 1015–1019, 2003.

[18] *Business Process Model And Notation (BPMN) Version 2.0*, Object Management Group, Jan. 2011.

[19] J. Mendling, *Metrics for process models: empirical foundations of verification, error prediction, and guidelines for correctness*. Springer, 2008.

[20] C. A. Petri, "Kommunikation mit Automaten." *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, vol. 1, 1966.

[21] M. Kunze, A. Luebbe, M. Weidlich, and M. Weske, "Towards understanding process modeling - the case of the bpm academic initiative," in *BPMN*, ser. Lecture Notes in Business Information Processing, R. M. Dijkman, J. Hofstetter, and J. Koehler, Eds., vol. 95. Springer, 2011, pp. 44–58.

[22] J. Leroux, "Vector addition system reachability problem: a short self-contained proof," *SIGPLAN Not.*, vol. 46, no. 1, pp. 307–316, Jan. 2011.

[23] M. Hack, "Decidability Questions for Petrin Nets," Cambridge, MA, USA, Tech. Rep., 1976.

[24] J. Esparza and K. Heljanko, *Unfoldings A Partial-Order Approach to Model Checking*. Springer-Verlag Berlin Heidelberg, 2008.

[25] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, vol. 3, no. 2, pp. 147–195, May 1969.

[26] T. Latvala and M. Mäkelä, *LTL Model Checking for Modular Petri Nets*. Springer Berlin Heidelberg, 2004, pp. 298–311.