Translating Standard Process Models to BPEL*

Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur H.M. ter Hofstede

Faculty of Information Technology
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
{c.ouyang,m.dumas,sw.breutel,a.terhofstede}@qut.edu.au

Abstract. Standardisation of languages in the field of business process management has long been an elusive goal. Recently though, consensus has built around one process implementation language, namely BPEL, and two fundamentally similar process modelling notations, namely UML Activity Diagram (UML AD) and BPMN. This paper presents a technique for generating BPEL code from process models expressed in a core subset of BPMN and UML AD. This model-to-code translation is a necessary ingredient to the emergence of model-driven business process development environments based on these standards. The proposed translation has been implemented as an open source tool.

1 Introduction

Over the past two decades, developments in the field of workflow and business process management have been hindered by the lack of a lingua franca for describing business processes, whether at the design or at implementation stages of the software lifecycle. Standardisation efforts during the 90s, led by the Workflow Management Coalition (WfMC), failed to be widely adopted for a number of reasons [1]. Recently however, consolidation has led to a single language for business process implementation: the Business Process Execution Language for Web Services (BPEL) [3]. In parallel, two process modelling notations, namely the Unified Modelling Language "Activity Diagram" (UML AD) [10] and the Business Process Management Notation (BPMN) [12], have attained some level of maturity and adoption.

There exist a number of business process execution engines that support BPEL, either natively or through import and export functions. Similarly, a large number of tools provide support for UML modelling, in particular using Activity Diagram, while BPMN, despite being a recent proposal, is already supported by about a dozen tools [5]. It appears however that support for translating models in UML AD and BPMN into BPEL code has received little attention relative to the amount of tools supporting these languages separately. Tools such as Telelogic's System Architect support the generation of BPEL code from BPMN diagrams but only for a limited subset of BPMN. More generally, proposed mappings from UML AD to BPEL [9] and from BPMN to BPEL [12] fail to address some difficult issues as discussed below.

* Supported by an Australian Research Council (ARC) Discovery Grant (DP0451092).

Both BPMN and UML AD share a common set of core constructs and for practical purposes, can be treated as variants of the same kernel language. This kernel is essentially an extension of flow charts with parallel splits (fork nodes) and synchronisation points (join nodes). As in flow charts, nodes in BPMN and UML AD (version 2.0) can be linked in arbitrary topologies, making it possible to write models with unstructured cycles. Meanwhile BPEL, which is essentially an extension of structured programming languages, only supports structured loops. Work in the field of structured programming [11, 14] has shown that it is possible to translate from unstructured to structured flow charts and from there to generate code in structured programming languages including "sequence", "if-then-else", and "while" constructs. It turns out however that after adding forks and joins to the flow chart notation these results no longer hold [7].

This paper takes on the challenge of designing a technique for translating BPMN and UML AD models with arbitrary topologies, which we term Standard Process Models or SPMs, into BPEL code. We consider this translation as being necessary to improve the connection between tools supporting process modelling and tools supporting process execution, thus enabling model-driven approaches to business process development based on standard and widely supported languages. The basic idea of the translation is to exploit an underused construct in BPEL, namely *event handlers*. This is the only construct in BPEL that allows one to capture processes with unbounded concurrency (i.e. processes with an unbounded number of threads running concurrently) without having to break down the process into several smaller ones, which may potentially lead to maintenance issues. Since some unstructured cycles in BPMN and UML AD may lead to unbounded concurrency, we argue that using event handlers is the only way to achieve a full translation from any SPM to a self-contained BPEL process. In this respect, the proposed translation goes beyond those in [9] and [12] which are essentially limited to structured models.

The rest of the paper is structured as follows. Sect. 2 gives an overview of BPEL and SPMs (the chosen abstraction of BPMN and UML AD) and reviews related work. Sect. 3 presents an initial approach to translate SPMs into BPEL. This mapping is then illustrated through a case study in Sect. 4. Sect. 5 describes an improvement to the initial translation approach which leads to more structured BPEL code. Finally, Sect. 6 concludes and outlines future work.

2 Background and Related Work

A Standard Process Model (SPM), also known as Standard Workflow Model [6], is constructed from a set of *process elements* and *transitions* connecting process elements. The process elements can be further divided into activities and control nodes which are AND-Split, XOR-Split, OR-Split, AND-Join and XOR-Join. Splits have exactly one incoming transition and at least one outgoing transition, and joins have exactly one outgoing transition and at least one incoming transition. For any split, each of its outgoing transitions has either an explicit guard (i.e. boolean expression) or an implicit "true" guard if none is explicitly given.

Activities have at most one incoming transition and one outgoing transition, and this implies that the use of implicit control nodes is not allowed. Activities with no incoming transitions are *initial activities*, and those with no outgoing transitions are *final activities*. Each SPM has exactly one initial activity and one final activity. It can be described using a process modelling notation such as BPMN or UML AD. In other words, SPMs can be seen as an abstraction of a subset of BPMN and UML AD, wherein constructs corresponding to advanced workflow patterns, e.g. *deferred choice* and *cancellation* [2], are not included.

BPEL [3] is essentially an extension of imperative programming languages (e.g. Pascal, C) with constructs related to the implementation of web service-oriented processes. A BPEL process definition relates a number of *activities*. Activities are split into two categories: *basic activities* and *structured activities*. Basic activities correspond to atomic actions such as: *invoke*, invoking an operation on some web service; *receive*, waiting for a message from a partner; *reply*, replying to a partner; *assign*, assigning a value to a variable; *exit*, terminating the entire process instance; *empty*, doing nothing; and etc. Structured activities impose behavioural and execution constraints on a set of activities contained within them. These include: *sequence*, for defining an execution order; *flow*, for parallel routing; *switch*, for conditional routing; *pick*, for capturing a race between timing and message receipt events; *while*, for structured looping; and *scope*, for grouping activities into blocks to which event, fault and compensation handlers (see below) may be attached.

Event, *fault* and *compensation handlers* are another family of control flow constructs in BPEL. In particular, *event handlers* are the only construct in BPEL that allows to have multiple simultaneously active instances within a single process instance (initiated by a single case). An event handler is an *event-action rule* associated with a scope. An event handler is enabled when its associated scope is under execution and may execute concurrently with the main activity of the scope. When an occurrence of the event associated with an enabled event handler is registered (and this may be a message receipt or a timeout), the body of the handler is executed. The completion of the scope as a whole is delayed until all active event handlers have completed. *Fault* and *compensation handlers* are designed for exception handling and are not used further in this paper.

SPMs can easily capture control-flow patterns, such as *multi-merge* and *arbitrary cycles*, for which BPEL does not offer direct support [15]. It may also have the facility for spawning multiple *independent* instances of activities within the context of a single case, and so far there has been no solution for mapping such a process to a *single* BPEL process. Hence, translating a process like one of the above into BPEL is not trivial. Below, we discuss this in detail using as an example the mapping from BPMN to BPEL proposed by White [12].

Figure 1 depicts four SPMs described using BPMN. The first three SPMs in Fig. 1(a) to Fig. 1(c) show fundamental issues and limitations in the mapping proposed in [12]. The SPM in Fig. 1(d) involves a livelock and will be mentioned at the end of this section. Note that in BPMN *parallel* (*forking/joining*) *gateways* correspond to AND-Splits/Joins, and *data-based exclusive* (*decision/merge*) *gateways* correspond to XOR-Splits/Joins.

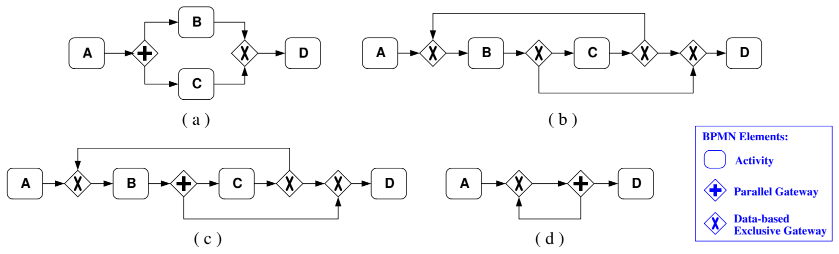

Fig. 1. Four SPMs (described using BPMN) which contain (a) a multiple merge, (b) arbitrary loops, (c) arbitrary loops with a facility spawning multiple independent instances of activities without synchronization, and (d) a livelock.

In Fig. 1(a), an XOR-Join following an upstream AND-Split captures a multi-merge pattern. In this process, activity D is executed twice: once when activity B completes and another time when activity C completes. In White's approach, a parallel gateway is always mapped to a BPEL "flow" activity and a data-based exclusive gateway to a "switch" activity [12]. This mapping assumes that for each AND-split there is a corresponding AND-join and for each XOR-split a corresponding XOR-join. However, in the scenario at hand, the outgoing branches of the AND-split lead to an XOR-Join, thus making White's mapping unapplicable.

In Fig. 1(b), there is a cycle with one entry point before activity B and two exit points – one after activity B, the other after activity C. This scenario cannot be mapped *directly* to a BPEL "while" activity as the "while" activity only captures structured cycles (i.e. loops with one entry point and one exit point). In [12], White considers only two types of cycles: *structured loops* and *interleaved loops*. Interleaved loops are a particular form of unstructured loops wherein two distinct loops can be identified which are not nested one inside the other. The basic idea to map such interleaved loops is to separate the original process into "one or more derived processes that are spawned from a main process and will also spawn or call each other". As a result, the original process will be mapped onto multiple BPEL processes rather than a single BPEL process. The synchronisation between the derived BPEL processes and the main process is achieved through message exchange[1]. While this is an interesting translation, it is not general enough: the scenario in Fig. 1(b) is neither a structured loop nor an interleaved loop (as it is not possible to distinguish two distinct loops on it), so its mapping is not covered by White's approach.

Fig. 1(c) illustrates yet another scenario not covered by White's approach. This model differs from the one in Fig. 1(b) in that there is an AND-Split (Fig. 1(c)) rather than an XOR-Split (Fig. 1(b)) between activities B and C. Since this AND-Split is located in a loop and also has another branch leading to

[1] A proposed extension to BPEL (`http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpelsubproc`) includes constructs for defining and invoking sub-processes. These constructs can be used to define multiple inter-related BPEL processes in a single module. However, there are no near-term plans of including these constructs in the BPEL standard. If these constructs were included, they could be used as an alternative to event handlers in our mapping.

activity D outside the loop, it provides a way for spawning multiple instances of D, all of which are independent of each other and no synchronisation is needed. As a new instance of activity D will be created each time the cycle is taken, the number of instances of D becomes unbounded. This captures the pattern of *multiple instances without synchronisation* [2]. Wohed et al. [15] proposes a solution for capturing this pattern in BPEL. The basic idea is to define another process containing activity D, and to invoke this "auxiliary" process multiple times thus spawning multiple instances of D. Again, the original process will be mapped onto multiple BPEL processes. In [12], White proposes a similar solution for mapping a subclass of parallel multiple-instance loops (without synchronisation) onto BPEL, which however does not cover the above scenario.

Sometimes, arbitrary cycles can be converted into structured cycles and these structured cycles can then be mapped directly onto BPEL "while" activities. However, not all non-structured cycles can be converted into structured ones when AND-splits and AND-joins are involved. An analysis of possible conversions and an identification of some situations where they are unapplicable can be found in [7,8]. For example, in Fig. 1(b) it is possible to unfold the arbitrary loops to structured ones, whereas in Fig. 1(c) the arbitrary cycles generate unbounded concurrency (i.e. they may spawn an unbounded number of concurrent instances of an activity) and do not have an equivalent structured form.

In the sequel, we present a technique to translate any SPM into a single BPEL process by exploiting the "event handler" construct in BPEL. The technique can be applied to any SPM so long as it does not involve a livelock (also called *divergence* in the concurrency theory literature) such as the one shown in Fig. 1(d). Livelocks can be detected using model-checking techniques and thus such undesirable SPMs could be excluded during a pre-processing step.

3 From Standard Process Models to BPEL

This section presents an initial approach for translating SPMs to BPEL. The translation focuses on control-flow perspective, and is conducted in three steps. We first generate so-called *precondition sets* for all activities in an SPM. Each precondition set is associated with an activity and encodes a possible way of enabling the activity. Next, all the precondition sets with their associated activities, are transformed into a set of *Event-Condition-Action* (ECA) rules. Finally, we translate this set of ECA rules into BPEL.

3.1 Translating Control-Flow Constructs into Precondition Sets

The term "precondition" is used to capture a conjunction of events and conditions that lead to the execution of an activity in a process. Thus, for each activity in an SPM, we can compute a precondition set that encapsulates all possible ways of reaching that activity. Fig. 2 shows an algorithm[2] for generating a set of precondition sets for all activities in an SPM. The algorithm is

[2] This is a variant of an algorithm designed in the context of a method for flexible execution of process-oriented applications [4].

```
AllPreCondSets($p$: Process):
  let $\{a_1, ..., a_n\}$ = Activities($p$) in
      return {PreCondSet($a_1$), ..., PreCondSet($a_n$)}

PreCondSet(x: Element):
   if IncomingTrans($x$) = $\emptyset$   /* initial element */
     return {ProcessInstantiation(Process($x$))}
   else let $\{t_1, ..., t_n\}$ = IncomingTrans($x$) in   /* non-initial elements */
          return PreCondSetTran($t_1$) $\cup$ ... $\cup$ PreCondSetTran($t_n$)

PreCondSetTran($t$: Transition)
   let $x$ = Source($t$)
      if ElementType($x$) = "Activity"
        return {Completion($x$)}
      else if ElementType($x$) $\in$ {"AND-Split","XOR-Split","OR-Split"}
            let $c$ = Guard(t),
               $\{prc_1, ..., prc_n\}$ = PreCondSet($x$) in
               return $\{c \wedge prc_1, ..., c \wedge prc_n\}$
      else if ElementType($x$) = "XOR-Join"
            let $\{t_1, ..., t_n\}$ = IncomingTrans($x$) in
               return PreCondSetTran($t_1$) $\cup$ ... $\cup$ PreCondSetTran($t_n$)
      else if ElementType($x$) = "AND-Join"
            let $\{t_1, ..., t_n\}$ = IncomingTrans($x$),
               $\{< prc_{1,1}, ..., prc_{1,n} >, ..., < prc_{m,1}, ..., prc_{m,n} >\}$ =
               PreCondSetTran($t_1$) $\times$ ... $\times$ PreCondSetTran($t_n$) in
               return $\{prc_{1,1} \wedge ... \wedge prc_{1,n}, ..., prc_{m,1} \wedge ... \wedge prc_{m,n}\}$
```

Fig. 2. Algorithm for deriving precondition sets from an SPM.

sketched using a functional programming notation. It defines three functions. The first one, namely AllPreCondSets, generates the above set of precondition sets for a process by relying on a second function named PreCondSet. This function takes as parameter a process element (i.e. an activity or a control node). If the element is an *initial* activity, i.e. it has no predecessors, the function returns a singleton set containing a process instantiation event, indicating that the corresponding activity will be executed when a new instance of the process must be started. Otherwise, function PreCondSet generates a precondition set for each of the non-initial elements in the process by relying on a third function named PreCondSetTran. This third function produces the same type of output as PreCondSet but takes as input a transition rather than an element in the process. Before moving on, we introduce the following notations used in the algorithm.

- Activities(p) is the set of activities in process p (defined as an SPM).
- IncomingTrans(x) is the set of transitions whose target is element x.
- Process(x) is the process to which element x belongs.
- ProcessInstantiation(p) is the event signaling to start an instance of process p.
- Source(t) is the source element of transition t.
- ElementType(x) is the type of element x (e.g. "Activity", "AND- Split", etc.).
- Completion(x) is the event signaling that activity x has completed.
- Guard(t) is the guard (i.e. boolean expression) on transition t.

The definition of PreCondSetTran operates based on the type of the source of the transition, which may be an activity or a control node. If the transition's source is an "Activity", a set is returned containing a single completion event for the activity. Intuitively, this means the transition in question may be taken when the activity has completed. Otherwise, if the source of the transition is a control node, the algorithm keeps working backwards through the process model, traversing other control nodes, until reaching activities. In the case of a transition originating from one of the "Split" nodes ("AND-Split", "XOR-Split", or "OR-Split"), which is generally labeled by a guard (or an implicit "true" guard if no guard is explicitly given), this condition is added as a conjunct to all the elements in the resulting precondition set. Finally, in the case of a transition originating from a "XOR-Join" (resp. a "AND-Join"), the function is recursively called for each of the transitions leading to this control node, and the resulting precondition sets are combined to capture the fact that when any (all) of these transitions is (are) taken, the corresponding XOR-Join (AND-Join) node may fire.

3.2 Translating Precondition Sets into ECA Rules

An ECA rule consists of three parts: *event*, which causes the rule to be triggered; *condition*, which is checked when the rule is triggered, and *action*, which is executed when the rule is triggered and its condition is true [13]. An ECA rule can be written in the form of `E[C]A`: `E` is a single event or a conjunction of single events (namely a *composite event*); `C` is a condition; `A` is a list of actions that can be executed in sequence (denoted as $a_1;a_2$), in parallel ($a_1||a_2$), in conditional branches (if-then-else), in loops (while), or in a combination of any of these block-structured constructs. If an ECA rule allows the use of single events only, it is called a *simple ECA rule*; otherwise, it is a *composite ECA rule*.

It is possible to translate a precondition into an ECA rule. To this end, we use two auxiliary functions GetEvent and GetCond, which extract respectively the events and conditions of a precondition. GetEvent takes as input a precondition *prc*, and gives as output a composite event equal to the conjunction of all the events appearing in *prc* (there is always at least one single event in *prc*). GetCond takes *prc* as input and gives as output a condition equal to the conjunction of all the conditions appearing in *prc* (it returns a value of TRUE if there is no condition in *prc*). A precondition *prc* for an activity a can be translated into the following ECA rule:

```
GetEvent(prc) [GetCond(prc)] {do a; invoke Completion(a)}
```

In the general case, this leads to a composite ECA rule. However, BPEL only supports simple ECA rules. To address this issue, when GetEvent(*prc*) is a composite event, say $e_1 \wedge ... \wedge e_n$, we translate the above rule into the following simple ECA rule:

```
e1 [TRUE] {receive e2 || ... || receive en;
           if GetCond(prc)
           then do a; invoke Completion(a)
           else empty}
```

The rule specifies that when occurrences for all events e_1 to e_n have been registered, the condition $\mathsf{GetCond}(prc)$ can be evaluated. If it evaluates to true, activity a is executed; otherwise, no action is performed.

By applying the above transformation to each precondition in a precondition set, we can translate a precondition set into a set of simple ECA rules. From there, we can generate a set of ECA rules for a given SPM by performing the union of the sets of ECA rules generated for each of the activities in the SPM. However, some of these rules may end up competing for the same event, which may lead to non-deterministic behaviour. For example, in the case of a Split node preceded by activity a_1 and followed by two activities a_2 and a_3, the precondition sets for a_2 and a_3 will both contain event $\mathsf{Completion}(a_1)$ and thus the resulting rules will compete for this same event. To avoid this problem, before transforming the precondition sets derived from an SPM into ECA rules, we rename the events shared by more than one precondition to eliminate any overlap between events. For example, the following set of precondition sets:

$\{\{\mathsf{ProcessInstantiation}(p)\},$
$\{\mathsf{Completion}(a_1)\wedge c_1\},$
$\{\mathsf{Completion}(a_1)\wedge c_2\},$
$\{\mathsf{Completion}(a_3)\},$
$\{\mathsf{Completion}(a_3),\ \mathsf{Completion}(a_1)\wedge c_3\},$
$\{\mathsf{Completion}(a_2),\ \mathsf{Completion}(a_4)\wedge c_4,\ \mathsf{Completion}(a_4)\wedge c_5\wedge\mathsf{Completion}(a_5)\}\}$

can be renamed to:

$\{\{\mathsf{ProcessInstantiation}(p)\},$
$\{\mathsf{Completion}(a_1)^{(1)}\wedge c_1\},$
$\{\mathsf{Completion}(a_1)^{(2)}\wedge c_2\},$
$\{\mathsf{Completion}(a_3)^{(1)}\},$
$\{\mathsf{Completion}(a_3)^{(2)},\ \mathsf{Completion}(a_1)^{(3)}\wedge c_3\},$
$\{\mathsf{Completion}(a_2),\ \mathsf{Completion}(a_4)^{(1)}\wedge c_4,\ \mathsf{Completion}(a_4)^{(2)}\wedge c_5\wedge\mathsf{Completion}(a_5)\}\}$

Because of this renaming process, we need to ensure that upon completion of an activity a, one occurrence of each of the completion events associated to a is produced. Coming back in the example above, instead of performing a single action "`invoke` $\mathsf{Completion}(a_1)$" following the execution of activity a_1, we perform the following actions:

`invoke` $\mathsf{Completion}(a_1)^{(1)}$ || `invoke` $\mathsf{Completion}(a_1)^{(2)}$ || `invoke` $\mathsf{Completion}(a_1)^{(3)}$

3.3 Translating ECA Rules into BPEL

A simple ECA rule $se\,[C]\,A$ can be realised by a BPEL event handler (`onEvent`) sketched in Fig. 3(a). As soon as an occurrence of event se is registered, the event handler starts with a *switch* activity in which condition C is evaluated. If C evaluates to true, the activity A is carried out; otherwise, nothing can be done. This event handler may be simplified if C is a boolean constant TRUE. In this case, the switch activity with its conditional branches (drawn in shaded boxes) can be omitted, and activity A is executed once the occurrence of event se is registered. In more detail, Fig. 3(b) sketches a BPEL event handler capturing a simple ECA rule which is transformed from a composite ECA rule as discussed in Sect. 3.2. Since the rule has a condition TRUE, the event handler executes the sequence of actions immediately upon registering the occurrence of event e_1. This sequence starts with a flow of receive activities waiting for occurrences of events e_2 to e_n, and a switch activity for conditional routings based on evaluation of the condition given by function GetCond(prc). Similarly to Fig. 3(a), if GetCond(prc) is a boolean constant TRUE, the switch activity can be omitted, and once the occurrences of events e_1 to e_n are registered, the event handler executes action a and the activity for invoking a single occurrence of event Completion(a) (when only a single occurrence of the event is needed).

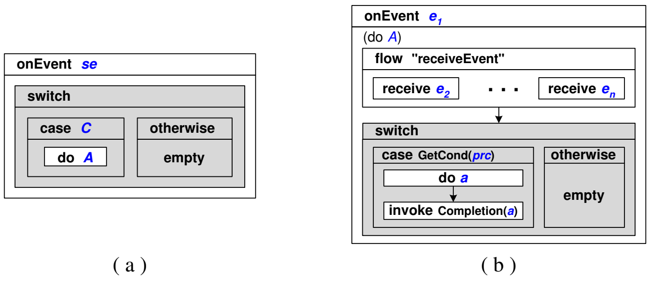

Fig. 3. Translating a simple ECA rule into a BPEL event handler.

Based on the above, we now translate the set of simple ECA rules derived from the original SPM into a BPEL process. We first introduce some notations. Given a process p, $\{a_1, ..., a_n\}$ is the set of activities in p, and the function InitialActivity(p) returns the initial activity of p. Let m+1 be the total number of ECA rules derived from process p, $\{se_1, ..., se_m\} \subseteq \{$Completion$(a_1), ...,$ Completion$(a_n)\}$ is the set of (single) events for triggering each of these ECA rules except the one associated with the initial activity. The ECA rule for execution of the initial activity is triggered upon occurrence of the process instantiation event (ProcessInstantiation(p)). The SPM of process p can be translated into the BPEL process sketched in Fig. 4. The main activity of this process is a sequence of three actions which corresponds to the ECA rule associated with the initial activity of p. Then each of the other (m) ECA rules are mapped onto totally m event handlers within the process. The whole process completes after its main activity and all active event handlers have completed.

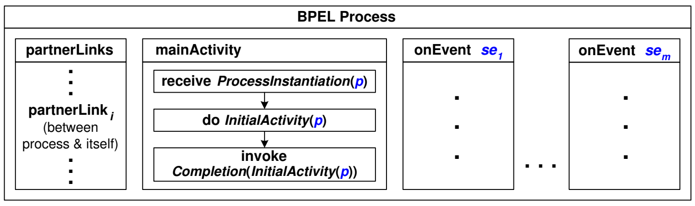
BPEL Process
partnerLinks
partnerLinki
(between process & itself)
mainActivity
receive ProcessInstantiation(p)
do InitialActivity(p)
invoke Completion(InitialActivity(p))
onEvent se1
onEvent sem

Fig. 4. A BPEL process derived from the set of ECA rules of an SPM.

The completion events $\mathsf{Completion}(a_1)$ to $\mathsf{Completion}(a_n)$ are produced by performing a BPEL invoke action via a *local* partner link between process p and itself. A local partner link which allows a process to send a message to itself, can be defined as:

```
<partnerLink name="local" partnerLinkType="localLT"
             myRole="localService"/>
</partnerLink>
```

where the corresponding partner link type can be defined as:

```
<partnerLinkType name="localLT">
     <role name="localService" portType="localPT"/>
</partnerLinkType>
```

In a BPEL invoke activity, one needs to specify, in addition to a partner link, a port type and an operation which are defined in a WSDL description. Accordingly, we define a single port type "`localPT`" and as many operations in this port type as there are completion events in the generated set of ECA rules. In the case of the example in Sect. 3.2, the operations over "`localPT`" for three completion events of activity a_1 can be defined as: "`completion_a1_1`", "`completion_a1_2`" and "`completion_a1_3`". These operations serve only to signal the completion of activities and do not carry any data. Their definition is thus trivial. For example, the production of event $\mathsf{Completion}(a_1)^{(1)}$ is captured in BPEL as follows:

```
<invoke partnerLink="local" portType="localPT"
        operation="completion_a1_1"/>
```

Likewise, completion events are consumed by event handlers and receive activities, referring to the local partner link, port type and the operations described above. For example, the event handler corresponding to event $\mathsf{Completion}(a_1)^{(1)}$ can be defined as follows:

```
<onEvent partnerLink="local" portType="localPT"
         operation="completion_a1_1"/>
```

We have implemented the above approach in a tool called SPM2BPEL, which supports automated translation from SPMs into BPEL. It is available under an open-source license at `http://www.bpm.fit.qut.edu.au/projects/babel/tools`.

4 Case Study

Consider the process for handling complaints shown in Fig. 5. It is described using BPMN. First the complaint is registered (activity *register*), then in parallel a questionnaire is sent to the complainant (*send questionnaire*) and the complaint is processed (*process complaint*). In the upper parallel path, the questionnaire is processed (*process questionnaire*) after it is returned from the complainant (*receive questionnaire*). In the lower parallel path, the complaint is evaluated (*evaluate*). Based on the evaluation result, the processing is either done or continues to activity *check processing.* If the check result is not ok, the complaint requires re-processing. After the complaint has been successfully processed, the

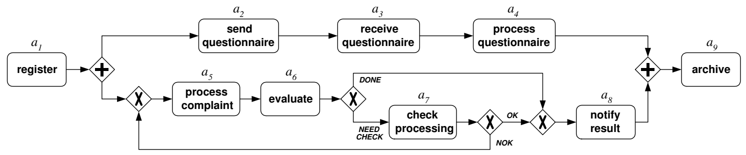

Fig. 5. A complaint handling process described using BPMN.

complainant is notified of the result. Finally, activity *archive* is executed. Note that the labels *DONE*, *NEED-CHECK*, *OK* and *NOK* on the outgoing transitions of each XOR-Split, are abstract representations of guards on these transitions.

Following the algorithm presented in Sect. 3, we now translate the above process into a BPEL process with event handlers. For simplicity, we assign each activity an activity identifier (placed above an activity rectangle in Fig. 5), and use these identifiers to refer to activities in the following translation.

Step 1: Generating Precondition Sets. Let p denote the process in Fig. 5, then $\mathsf{Activity}(p)=\{a_1, ..., a_9\}$. The precondition sets for each of these activities are:

$$\begin{aligned}
\mathsf{PreCondSet}(a_1) &= \{\mathsf{ProcessInstantiation}(p)\}\\
\mathsf{PreCondSet}(a_2) &= \{\mathsf{Completion}(a_1)\}\\
\mathsf{PreCondSet}(a_3) &= \{\mathsf{Completion}(a_2)\}\\
\mathsf{PreCondSet}(a_4) &= \{\mathsf{Completion}(a_3)\}\\
\mathsf{PreCondSet}(a_5) &= \{\mathsf{Completion}(a_1),\ \mathsf{Completion}(a_7)\wedge NOK\}\\
\mathsf{PreCondSet}(a_6) &= \{\mathsf{Completion}(a_5)\}\\
\mathsf{PreCondSet}(a_7) &= \{\mathsf{Completion}(a_6)\wedge \mathit{NEED\text{-}CHECK}\}\\
\mathsf{PreCondSet}(a_8) &= \{\mathsf{Completion}(a_6)\wedge DONE,\ \mathsf{Completion}(a_7)\wedge OK\}\\
\mathsf{PreCondSet}(a_9) &= \{\mathsf{Completion}(a_4)\wedge\mathsf{Completion}(a_8)\}
\end{aligned}$$

Step 2: Generating ECA Rules. The completion events for activities a_1, a_6 and a_7, each appears twice in the above precondition sets. Thus, it is necessary to rename these events. After the renaming process, all the precondition sets for process p can be translated into the set of simple ECA rules listed below, where "for a_i" is a shortened form of "for execution of activity a_i".

For a_1: $\mathsf{ProcessInstantiation}(p)[\mathsf{TRUE}]$
$\quad\{$do a_1; invoke $\mathsf{Completion}(a_1)^{(1)}$ || invoke $\mathsf{Completion}(a_1)^{(2)}\}$
For a_2: $\mathsf{Completion}(a_1)^{(1)}[\mathsf{TRUE}]\{$do a_2; invoke $\mathsf{Completion}(a_2)\}$
For a_3: $\mathsf{Completion}(a_2)[\mathsf{TRUE}]\{$do a_3; invoke $\mathsf{Completion}(a_3)\}$
For a_4: $\mathsf{Completion}(a_3)[\mathsf{TRUE}]\{$do a_4; invoke $\mathsf{Completion}(a_4)\}$
For a_5: $\mathsf{Completion}(a_1)^{(2)}[\mathsf{TRUE}]\{$do a_5; invoke $\mathsf{Completion}(a_5)\}$
$\quad\mathsf{Completion}(a_7)^{(1)}[NOK]\{$do a_5; invoke $\mathsf{Completion}(a_5)\}$
For a_6: $\mathsf{Completion}(a_5)[\mathsf{TRUE}]$
$\quad\{$do a_6; invoke $\mathsf{Completion}(a_6)^{(1)}$ || invoke $\mathsf{Completion}(a_6)^{(2)}\}$
For a_7: $\mathsf{Completion}(a_6)^{(1)}[\mathit{NEED\text{-}CHECK}]$
$\quad\{$do a_7; invoke $\mathsf{Completion}(a_7)^{(1)}$ || invoke $\mathsf{Completion}(a_7)^{(2)}\}$
For a_8: $\mathsf{Completion}(a_6)^{(2)}[DONE]\{$do a_8; invoke $\mathsf{Completion}(a_8)\}$
$\quad\mathsf{Completion}(a_7)^{(2)}[OK]\{$do a_8; invoke $\mathsf{Completion}(a_8)\}$
For a_9: $\mathsf{Completion}(a_4)[\mathsf{TRUE}]\{$receive $\mathsf{Completion}(a_8)$;
$\quad\quad$do a_9; invoke $\mathsf{Completion}(a_9)\}$

Step 3: Deriving the BPEL Process. The above ECA rules can be translated into a BPEL process of which the XML code is sketched in Fig. 6. The rule for execution of activity a_1 is mapped to the last sequence activity, i.e. the main activity of the process, and the rest of the rules are mapped to event handlers. All the events are identified in an abstract way. Intuitively, the arrival of a complaint from a client will initiate a new instance of the process, and thus can be treated as a process instantiation event. The production and consumption of each completion event can be defined in a similar way as that of event Completion$(a_1)^{(1)}$ described in Sect. 3.3. The receive activity waiting for the process instantiation event, is the "start activity" of the process and thus has the `createInstance` attribute set to `yes`. Also, we would like to elaborate all the activities in the original process to obtain a list of detailed code for the resulting BPEL process. For example, activity "register" (a_1) or "archive" (a_9) may be mapped to a BPEL assign activity for recording the relevant information into variables, and activity "sendQuestionnaire" (a_2) corresponds to an invoke activity for sending the questionnaire to the client. This elaboration procedure is however not the focus of our approach, and thus is not presented here. The interested reader may refer to the testing example of SPM2BPEL (on the tool's website) for a complete list of the BPEL code generated from the complaint handling process in Fig. 5.

```
<process name="complaintHandling">
  <partnerLinks>
    <partnerLink name="local" partnerLinkType="localLT" ... />
    ...
  </partnerLinks>
  <variables> ... </variables>
  <eventHandlers>
    <onEvent Completion(a1)(1)/>
      <sequence>
        <invoke name="sendQuestionnaire" ... />  <!--do a2-->
        <invoke Completion(a2)/>
      </sequence>
    </onEvent>
    ...
    <onEvent Completion(a4)/>
      <sequence>
        <receive Completion(a8)/>
        <assign name="archive"> ... </assign>  <!--do a9-->
        <invoke Completion(a9)/>
      </sequence>
    </onEvent>
  </eventHandlers>
  <sequence>
    <receive ProcessInstantiation(p) createInstance="yes"/>
    <assign name="register"> ... </assign>  <!--do a1-->
    <flow>
      <invoke Completion(a1)(1)/>
      <invoke Completion(a1)(2)/>
    </flow>
  </sequence>
</process>
```

Fig. 6. An abstract view of the BPEL code for complaint handling process in Fig. 5.

5 Improving the Translation Approach

The previous approach in Sect. 3 treats each activity of an SPM as a single unit for translation. This can be improved by taking advantage of structured activities defined in BPEL. For example, in the complaint handling process in Fig. 5, three activities a_2, a_3 and a_4 can be directly mapped onto a "sequence" activity. Hence, if we cluster them into one activity block as a single unit for translation, the complexity of translation can be reduced with less precondition sets, less ECA rules and less event handlers, and the resulting BPEL process will become more compact. However, it is not always the case that a number of activities clustered into an activity block can be directly mapped onto a structured activity in BPEL. Coming back in the example in Fig. 5, the process elements on the lower parallel path constitute an unstructured workflow which cannot be mapped directly onto a structured activity (e.g. a "while" activity).

To improve our approach further, we would like to transform unstructured activity blocks to structured ones which can then be mapped onto structured activities. Workflows that do not contain parallelism have similar semantics as elementary flow charts that are commonly used for procedural program specification. Work in the field of structured programming [11] has shown that any unstructured flow chart can be transformed to a structured one and from there one can generate code in structured programming languages including "sequence", "if-then-else", and "while" constructs. Based on this, we propose to cluster those connected process elements except AND-Splits and AND-Joins into an activity block. Since each activity block will later be treated as a single unit for translation, it cannot have more than one entry point nor more than one exit point. Below, we define *Clusterable Activity Blocks* based on the concept of *Weakly Connected Component* (from MathWorld `http://mathworld.wolfram.com/WeaklyConnectedComponent.html`).

Definition 1. *A* Weakly Connected Component (WCC) *is a maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there is an undirected path from u to v and a directed path from v to u.*

Definition 2. *A* Clusterable Activity Block (CAB) *is a WCC that has at most one entry point and one exit point in an SPM. It is made up of activities, control nodes except AND-Splits and AND-Joins, and transitions connecting these process elements, such that:* $\forall x \in \mathsf{Elements}(CAB)$,

- $\mathsf{ElementType}(x) \in \{\textit{Activity, XOR-Split, OR-Split, XOR-Join}\}$;
- *let* $T_i = \bigcup_{x \in \mathsf{Elements}(CAB)} \mathsf{IncomingTrans}(x)$, $|T_i \setminus \mathsf{Transitions}(CAB)| \leqslant 1$; *and*
- *let* $T_o = \bigcup_{x \in \mathsf{Elements}(CAB)} \mathsf{OutgoingTrans}(x)$, $|T_o \setminus \mathsf{Transitions}(CAB)| \leqslant 1$.

Before translating an SPM into BPEL, we pre-process the model by clustering all the original activities into CABs. We start with an arbitrary unclustered activity in the process, then move backwards and forwards from that activity without traversing AND-Splits/Joins (i.e. stop when reaching an AND-Split/Join), and finally cluster all the traversed elements and transitions into a single CAB. This procedure is repeated until no unclustered activities are left in the process.

Next, we replace the "Activity" elements with "CAB" elements in the previous algorithm defined in Sect. 3, and apply this updated algorithm to the above pre-processed model. Finally, we map each CAB onto BPEL activities. This may also include transformation from unstructured to structured workflow before the mapping. Note that in the worst case a CAB contains only a single activity.

Example. We apply the improved approach to the translation of the complaint handling process in Fig. 5. Fig. 7 depicts a pre-processed model for this process where the previous *nine* activities with *four* XOR-Splits/Joins are clustered into *four* CABs. CAB_1, which contains just the initial activity a_1, is the initial item, and CAB_4, which consists of only the final activity a_9, is the final item.

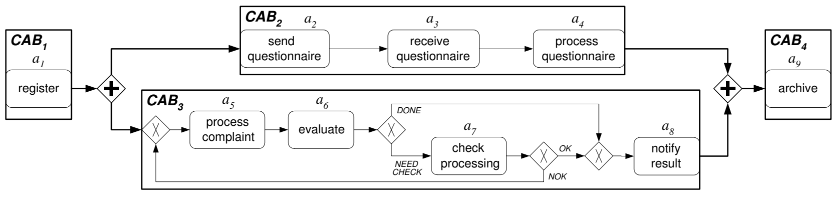

Fig. 7. Pre-processed complaint handling process in Fig. 5.

The precondition sets for each of the *four* CABs in Fig. 7 are:

$\mathsf{PreCondSet}(CAB_1) = \{\mathsf{ProcessInstantiation}(p)\}$
$\mathsf{PreCondSet}(CAB_2) = \{\mathsf{Completion}(CAB_1)\}$
$\mathsf{PreCondSet}(CAB_3) = \{\mathsf{Completion}(CAB_1)\}$
$\mathsf{PreCondSet}(CAB_4) = \{\mathsf{Completion}(CAB_2) \wedge \mathsf{Completion}(CAB_3)\}$

The set of simple ECA rules derived from these precondition sets are:

For CAB_1: $\mathsf{ProcessInstantiation}(p)[\mathsf{TRUE}]$
$\{$do CAB_1; invoke $\mathsf{Completion}(CAB_1)^{(1)}$ || invoke $\mathsf{Completion}(CAB_1)^{(2)}\}$
For CAB_2: $\mathsf{Completion}(CAB_1)^{(1)}[\mathsf{TRUE}]\{$do CAB_2; invoke $\mathsf{Completion}(CAB_2)\}$
For CAB_3: $\mathsf{Completion}(CAB_1)^{(2)}[\mathsf{TRUE}]\{$do CAB_3; invoke $\mathsf{Completion}(CAB_3)\}$
For CAB_4: $\mathsf{Completion}(CAB_2)[\mathsf{TRUE}]\{$receive $\mathsf{Completion}(CAB_3)$;
do CAB_4; invoke $\mathsf{Completion}(CAB_4)\}$

The above ECA rules, except the first one for execution of the initial item CAB_1, can be translated into *three* event handlers. As a comparison, our previous approach yields *ten* event handlers for the same process.

Finally, the improved approach requires an additional step of mapping CABs onto structured activities in BPEL. This does not apply to CAB_1 and CAB_4 as both contain a single activity. As mentioned before, CAB_2 can be directly mapped onto a sequence activity. CAB_3 exhibits an unstructured workflow which can be transformed into an equivalent structured form shown in Fig. 8. The transformation is done by introducing an auxiliary boolean variable (Q) to carry the evaluation result of the guard represented by *OK* ($\sim Q$ for *NOK*). Three new activities a_{10}, a_{11} and a_{12} are also added to assign appropriate values to Q. The resulting workflow in Fig. 8 has a structured loop which can be directly mapped onto a while activity. Each loop starts if Q has a value of $false$, otherwise the loop will be exit. The main activity of the loop is a sequence of a_5, a_6 and a conditional choice between two branches – one for the guard represented by *DONE*, the other for *NEED-CHECK*.

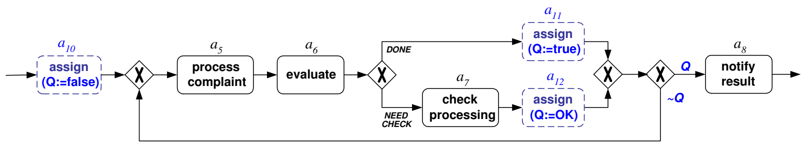

Fig. 8. A structured form of the original unstructured workflow in CAB_3.

Fig. 9 sketches the XML code of the abstract BPEL process for the complaint handling process shown in Fig. 5, which is generated under the above improved translation approach. It can be observed that within the event handlers corresponding to activity blocks CAB_2 CAB_3 are more structured BPEL activities. For example, CAB_3 is mapped to a complex structured activity where "sequence", "while" and "switch" constructs are nested within each other.

6 Conclusions

Capturing workflow patterns such as multi-merge, arbitrary cycles and multiple instances in BPEL is problematic. On the other hand, these patterns can be directly captured in standard process modelling notations (i.e. BPMN and UML AD). This mismatch hinders the definition of automated translations from process models to process implementations when using these standards. This paper has presented a technique to translate models captured in a core subset of BPMN or UML AD into BPEL. The technique exploits an interesting and often underused BPEL construct, namely "event handler".

To the best of our knowledge, this is the first attempt at tackling the above patterns in a systematic translation from BPMN or UML AD to BPEL. The proposal has been validated through the implementation of a tool (SPM2BPEL) that automatically translates Standard Process Models into BPEL code. The paper also sketched possible improvements to the technique by clustering activities into activity blocks that can be mapped onto BPEL structured activities, thereby reducing the number of event handlers in the resulting BPEL process.

Ongoing work aims at designing and implementing an algorithm for the improved translation technique. We then plan to extend the technique to cover other workflow patterns, e.g. deferred choice and cancellation. Since the deferred choice pattern captures a race condition between events, the translation algorithm presented in this paper, which excludes the notion of race condition, will need to be revisited. Investigating the expressive power of the BPEL "pick" and "fault handler" constructs, which allow one to capture race conditions and cancellation in structured settings, may provide a foundation for designing this extended translation.

```
<process name="complaintHandling">
  <partnerLinks>
     <partnerLink name="local" partnerLinkType="localLT" ... />
     ...
  </partnerLinks>
  <variables> ... </variables>
  <eventHandlers>
     <onEvent Completion($CAB_1$)$^{(1)}$/>
        <sequence>
           <sequence>  <!--do $CAB_2$-->
              <invoke name="sendQuestionnaire" ... />  <!--do $a_2$-->
              ...
           </sequence>
           <invoke Completion($CAB_2$)/>
        </sequence>
     </onEvent>
     <onEvent Completion($CAB_1$)$^{(2)}$/>
        <sequence>
           <sequence>  <!--do $CAB_3$-->
              <assign name="$a_{10}$"> ... </assign>
              <while>
                 <condition> NOK </condition>  <!-- ~OK/~Q/false-->
                 <sequence> ... </sequence>  <!--do $a_5$ & $a_6$-->
                 <switch>
                    <case>
                       <condition> DONE </condition>
                       <assign name="$a_{11}$"> ... </assign>
                    </case>
                    <otherwise>  <!--NEED-CHECK-->
                       <sequence> ... </sequence>  <!--do $a_7$ & $a_{12}$-->
                    </otherwise>
                 </switch>
              </while>
              <assign name="$a_8$"> ... </assign>
           </sequence>
           <invoke Completion($CAB_3$)/>
        </sequence>
     </onEvent>
     <onEvent Completion($CAB_2$)/>
        <sequence>
           <receive Completion($CAB_3$)/>
           <assign name="archive"> ... </assign>  <!--do $CAB_4$/$a_9$-->
           <invoke Completion($CAB_4$)/>
        </sequence>
     </onEvent>
  </eventHandlers>
  <sequence>
     <receive ProcessInstantiation($p$) createInstance="yes"/>
     <assign name="register"> ... </assign>  <!--do $CAB_1$/$a_1$-->
     <flow>
        <invoke Completion($CAB_1$)$^{(1)}$/>
        <invoke Completion($CAB_1$)$^{(2)}$/>
     </flow>
  </sequence>
</process>
```

Fig. 9. An abstract view of the BPEL code for complaint handling process in Fig. 5 as generated using the improved translation approach.

References

1. W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.
2. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
3. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0.* Working Draft. WS-BPEL TC OASIS, May 2005. URL: `http://www.oasis-open.org/committees/download.php/12791/`.
4. M. Dumas, T. Fjellheim, S. Milliner, and J. Vayssiére. Event-based coordination of process-oriented composite applications. In *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 236–251, Nancy, France, 2005. Springer-Verlag.
5. P. Harmon. Standardizing business process notation. URL: `http://www.bptrends.com`, November 2005.
6. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
7. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Proceedings of 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, London, UK, 2000. Springer-Verlag.
8. R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. In *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 268–284, Nancy, France, 2005. Springer-Verlag.
9. K. Mantell. From UML to BPEL. URL: `http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel`, September 2005.
10. OMG. *Unified Modeling Language: Superstructure.* UML Superstructure Specification v2.0, formal/05-07-04. OMG, August 2005. URL: `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.
11. G. Oulsnam. Unravelling unstructured programs. *Computer Journal*, 25(3):379–387, 1982.
12. S. A. White. *Business Process Modeling Notation (BPMN) Version 1.0.* Business Process Management Initiative, BPMI.org, May 2004.
13. J. Widom and S. Ceri, editors. *Active Database Systems : Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996.
14. M. H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *Computer Journal*, 20(1):45–50, 1977.
15. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web services composition languages: The case of BPEL4WS. In *Proceedings of 22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, 2003.