Formal analysis of BPMN models using Event-B

Jeremy W. Bryans[1] and Wei Wei[2]

[1] School of Computing Science, Newcastle University, United Kingdom
Jeremy.Bryans@ncl.ac.uk
[2] SAP Research CEC Darmstadt, SAP AG, Bleichstr. 8, 64283 Darmstadt, Germany
wei01.wei@sap.com

Abstract. The use of business process models has gone far beyond documentation purposes. In the development of business applications, they can play the role of an artifact on which high level properties can be verified and design errors can be revealed in an effort to reduce overhead at later software development and diagnosis stages. This paper demonstrates how formal verification may add value to the specification, design and development of business process models in an industrial setting. The analysis of these models is achieved via an algorithmic translation from the de-facto standard business process modeling language BPMN to Event-B, a widely used formal language supported by the Rodin platform which offers a range of simulation and verification technologies.

1 Introduction

Complex, large-scale business information systems are critical to the successful operation of many businesses, and SAP is a leading provider of such systems. Business process modeling has become increasingly important to the development of enterprise software applications [12]. Nowadays, business applications are usually built by integrating a broad range of highly configurable software components and services, which can be rapidly tailored to satisfy different and constantly changing business needs. Business process models are used to describe such integration scenarios and their work flows, facilitating an intuitive common understanding of the business logic between customers and developers. In addition to their use as documentation, business process models can also be simulated, analyzed and verified to reveal design errors at an early stage in software development. This promises to enhance the efficiency of reaching high-quality software solutions and can save substantial implementation and diagnosis costs which would otherwise be incurred at later development phases.

We wish to use formal methods to improve the quality of business process models within a software design process, and also aim to reduce the extra burden that formal methods induce on designers and developers. Within the context of the DEPLOY project[3], we choose the Event-B modeling formalism [1] and the Rodin platform [2] in our pursuit of these goals. The choice is also encouraged

[3] www.deploy-project.eu

by our past successful experiences of using Event-B for describing and analyzing business applications [5, 6]. Event-B offers many indispensable features for analyzing business process models such as the ability to model data. The Rodin platform is empowered by a large number of plug-ins providing various analysis capabilities like specialized provers, model checking, and simulation.

This paper examines our recent work on the formal analysis of business process models using Event-B and Rodin, and discusses the impact of the analysis results on software design and development. We also investigate the potential to largely automate these analyses in order to pave the way for future industrial deployment. We designed an algorithmic translation from BPMN, the de-facto standard business process modeling language, to Event-B. The translation covers most of the commonly used BPMN features, also including features newly introduced in the proposed draft of the second version of the language [15]. We also make the Event-B translation structurally faithful to the original BPMN model, which not only improves readability, but also enhances provability and analyzability.

Outline. In Section 2 we briefly introduce BPMN and Event-B, and in Section 3 sketch our translation from BPMN to Event-B. Sections 4 and 5 describe two case studies to illustrate how formal analysis is performed on the Event-B translations of BPMN models, and also discuss the possibility of automating these analysis procedures. Related work is discussed in Section 6, before we conclude in Section 7 with a discussion of our next steps. Due to space constraints, we have moved some of the Event-B code and discussion into appendices.

2 Background

BPMN. We introduce the Business Process Modeling Notation (BPNM) elements we use in this paper. We only show syntactic compositions here. The semantics of syntactic elements will be discussed later when we explain how they are translated.

A typical BPMN model consists of one or more *pools*, each representing a collaboration partner (such as `FACTORY` and `WORKER` in Figure 1). Each pool usually contains a top process. A *process* contains flow objects and the connections between them. Flow objects include *events*, *gateways* and *activities*. Events either throw or catch triggers and are represented as circles containing a marker indicating the kind of trigger. Gateways converge or diverge control flows and are represented as diamonds. An *activity* can be either an atomic *task* or a composite *sub-process* that contains an inner process. An activity can be a loop. Activities are graphically represented as rounded rectangles. A process may contain data items as process instance attributes. There are also data stores that are process-independent and globally accessible.

Two pools communicate with each other mainly by exchanging messages, which may carry data fields. Message flows are represented as directed dotted lines connecting two pools. BPMN does not dictate how the message exchange

mechanism works. In this paper, we assume that messages may not be lost, duplicated, or altered but may arrive in any order. Furthermore, BPMN provides the concept of correlation to identify the proper recipient of a message.

An important concept of BPMN is activity compensation that usually happens when the effect of an activity is no longer desired and needs to be reversed. We will discuss compensation in greater length in Section 3.6. A complete description of all BPMN features can be found in [15].

Event-B and Rodin. An Event-B model consists of *contexts* and *machines*. The contexts describe the static elements of the model, whereas the machines specify the dynamic behavior of the model. Each machine may contain *variables* that model persistent state data, *invariants* that restrict the valid content of variables, and guarded *events* that describe functionality of the machine in terms of actions defined over the state variables. Typically, a model consists of a chain of Event-B machines, each of which (apart from the first) is linked to its predecessor by a refinement relation expressed in terms of a gluing invariant between the two machines. In a refinement relation, we refer to the successor machine and its components as *concrete* and the predecessor and its components as *abstract*. A concrete event refines an abstract event if the guards of the concrete event imply the guards of the abstract event and the abstract actions simulate the concrete ones with respect to the gluing invariant. Machines and refinement steps give rise to proof obligations that ensure internal consistency of individual machines (e.g. well-definedness and feasibility of events) and behavior preservation across refinement steps. A typical Event-B model has an extremely simple initial machine, with detail added in a controlled way through refinement steps. These steps are usually small to reduce the size and complexity of the generated proof obligations and the associated burden on the automatic provers. We make substantial use of refinement in our translation from BPMN to Event-B. A detailed account of the Event-B language can be found in [1].

Rodin is an open, extensible toolset for modelling and verification of Event-B models. A model editing interface is provided for constructing Event-B machines and refinement steps. Proof obligations are automatically generated and discharged (as far as possible automatically) by proof tools built into the platform. In the event of an obligation not being automatically discharged, an interface for manual proof guidance can used.

3 Translating BPMN to Event-B

BPMN is specified using natural and graphic languages, and comes with no rigorous semantics defined. Therefore, there are a lot of ambiguities in BPMN that had to be clarified when we designed the translation into Event-B. These clarifications are according to the specific needs of our use cases, so by no means do they offer the only proper solutions – other semantic variants can be chosen.

The translation covers most of the commonly used BPMN features including comprehensive modeling of control flows, data modeling, compensation, message

based communication, error and exception handling, sub-processes, looping and multi-instance activities. The uncovered BPMN features are most notably choreography and conversations as well as some types of flow objects, including call activities, transactions, conditional events and complex gateways. Some of these missing features are rarely used in practice and add significant complexity to the model. Other missing features such as transactions have very vague descriptions in the official BPMN specification and are difficult to interpret.

Our translation was guided by three principles. First, the Event-B translation should be **structurally faithful** to the original BPMN model so that anyone with knowledge of the original model can easily understand the translation. Also, any analysis result that we may obtain from the Event-B translation can be easily mapped back to the original model. Second, the translation should be designed to improve **provability**, i.e. it should result in the automatic discharge of as many proof obligations as possible. Finally, we are interested in verifying properties for systems that allow **multiple instances** of same processes.

We are unable to give a detailed description of how each BPMN element is translated. We therefore select a few important BPMN features and explain the intuition of their translation.

3.1 The structure of the translation

We take the model in Figure 1 as an example to show how its Event-B translation is structured (Figure 2). This model describes the management of shift work within a factory: A worker assigned to a shift becomes unavailable, and the manager has to find a replacement from the pool of available workers. The status of each worker is maintained in a database. In this scenario, an attempt is made to automatically choose a replacement. A request is sent to an available worker, who has a fixed length of time to reply. If he accepts, he is assigned to the shift and the database is updated. Otherwise, the process may be repeated up to a maximum of five times. If, after five attempts, a replacement has not been found, a manager steps in to allocate a worker to the shift directly.

The contexts in the Event-B translation contain common definitions such as process life cycle states as well as abstract constants and carrier sets that represent process instances, message instances, data types, and so on. The translations of processes and their communication are gradually added to a series of refining machines: The machine at the first level contains nothing but the control flow information of the `Factory` process. In particular, it has neither data information nor the internal detail of the sub-process `schedule`. The machine at the second level preserves or refines all information in the first machine, and adds also the data flow information of `Factory`. Details of `schedule` and `WORKER` are added similarly into later refinements. In the end, the communication between the two top processes is added into the last machine.

The above structure preserves the hierarchical structure of the original model through refinements: the information of a sub-process (e.g., `schedule`) is always added into machines at higher refinement levels than that of the container process (e.g., `FACTORY`). Our structure also achieves separation of concerns, which is very

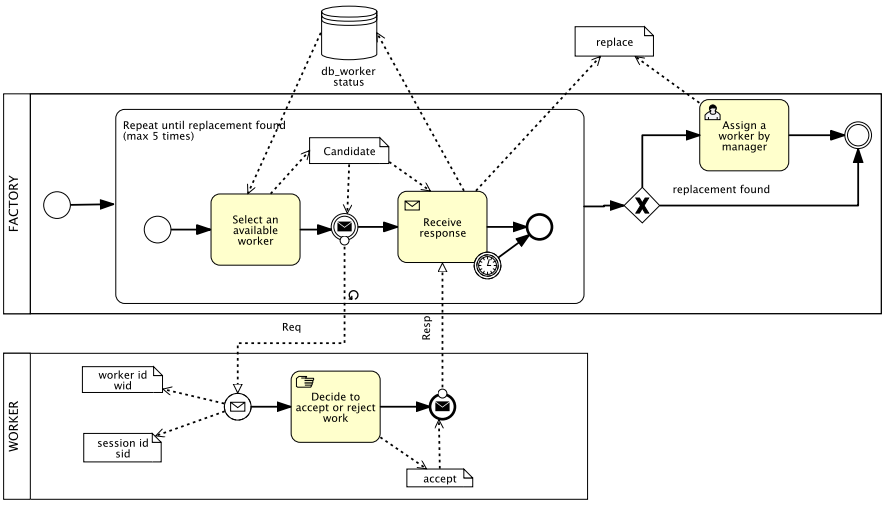

Fig. 1. The shift worker scheduling model.

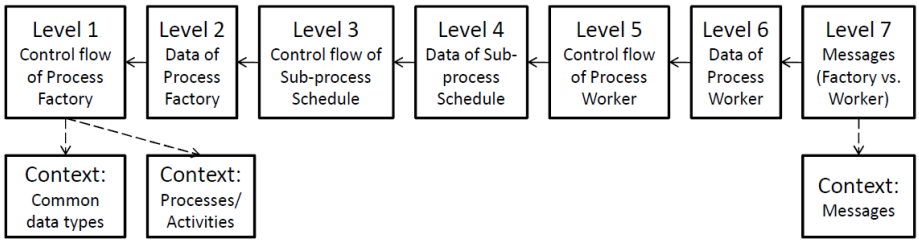

Fig. 2. The structure of the Event-B translation of Figure 1.

beneficial for automated provers: A property about the control flow of process `Factory` can be expressed and proved at the first refinement level since it needs no information from later levels. This means a smaller hypothesis space for automated provers to search.

3.2 Processes

We allow multiple instances of a process. We use an abstract carrier set to represent all possible instances of each process (e.g. `PROC_FACTORY_INSTANCES`) in contexts. The machines contain variables recording existing process instances (e.g. `instances_Factory`); recording the life cycle state of each existing instance (e.g. `state_Factory`) and, in case of a sub-process, recording the parent of each

sub-process instance (e.g. `parent_inner_schedule`); and recording which activity instance (outer instance) results in the creation of a sub-process instance (e.g. `outer_inner_schedule`).

Control flow. Our interpretation of sequential and parallel executions of flow objects uses tokens. For each sequence flow, we define a function that maps each process instance to the number of tokens in this particular process instance. Tokens are initialized when a new process instance is created by a start event: all outgoing sequence flows from the start event receive a certain number of tokens (usually 1), and all other flows receive no tokens. Each flow object is guarded by a condition stipulating how many tokens it needs to start execution.

Control flow convergence. With a few exceptions like join gateways, a flow object with multiple incoming flows needs only one of the incoming flows to carry enough tokens to start execution. In this case, we use as many Event-B events to represent the flow object as the number of incoming flows: each event describes the situation in which the tokens on the corresponding incoming flow are consumed. This is because otherwise we must express disjunctive choices and updates of tokens in the guard and action of the Event-B event representing the flow object. Automated provers often struggle to deal with disjunctions because they lead to case splitting and a potential explosion in the size of the proof tree. Appendix A shows how control flow convergence is translated. On the contrary, a join gateway requires all incoming flows to carry enough tokens to start execution. Then, it is enough to have one Event-B event to represent the gateway, which consumes tokens from all incoming flows.

Data. There are three kinds of data: process attributes, data stores, and activity inputs/outputs. For each process attribute, we define a function that maps each process instance to the runtime value of the attribute in that particular instance. A data store is globally accessible and does not belong to a particular process. Therefore, unlike process attributes, the data structure representing the data store involves no process information. Finally, activities may have input and output parameters. BPMN allows activities to have multiple sets of inputs or outputs. However in our translation we stipulate that any flow object or sub-process has at most one input set and one output set. We also do not explicitly represent inputs and outputs, since the runtime values of inputs/outputs are decided by process attributes or data stores.

3.3 Events triggers

An event either throws or catches a certain kind of triggers. The BPMN specification provides no information of trigger structures and how triggers are processed, stored, and discarded. In our understanding, each kind of triggers has its specific processing mechanism. For instance, the trigger of a message receiving event occurs when a desired message becomes available, and it persists until the message is consumed. On the contrary, the trigger of a conditional event occurs

when a certain condition is fulfilled. However, if the conditional event is not ready to be triggered, e.g, it has no incoming tokens, then the trigger immediately disappears. Based on the above discussion, we have no explicit and unified representation for all kinds of triggers. Instead, we model the trigger behavior implicitly in their executional contexts.

3.4 Messages

Message buffers are implemented simply as sets since message order information is absent. For each type of message, we introduce two variables to record (1) the set of already sent messages of the type and (2) the set of messages still in the buffer (i.e., not yet received). Note that the buffer is shared by all process instances that may send or receive this type of messages. Sending a message is simply to add the message into both the buffer and the set of already sent messages, while receiving a message is to remove it from the buffer. Message fields are defined as functions that map each message instance to the concrete value of the corresponding field in that message.

Some message fields may contain correlation information that identifies the intended receiver which contains matching correlation information. In the model in Figure 1, session identifiers (`sid`) are used as correlation information. Each response message contains an `sid` field, which can be received only by a process instance with a matching `sid` as its process attribute. A request message is used to create a new `WORKER` instance. Therefore, a new request message should contain a new `sid`. Appendix B illustrates how correlation-based message exchanges are translated.

3.5 Sub-Processes

In BPMN, a sub-process can be either collapsed or expanded, with the internal structure of the sub-process either hidden or revealed respectively. These two appearances find their analogies in the refinement hierarchy of the Event-B translation: The sub-process is first specified without internal detail when the control flow of its containing process is added. The internal detail of the sub-process is specified at later refinement levels.

For a looping sub-process, each execution creates a single *outer* instance, which acts as a container for multiple `inner` instances. The execution of an inner instance corresponds to a single loop iteration. We show in Appendix C the translation of the "collapsed view" of the loop sub-process in the `FACTORY` process in Figure 1.

The translation of the "expanded view" is shown below. At this level we add the outer instance attribute `loop counter`, and also introduce an auxiliary variable `next` to control the creation of the next inner instance. In our example, the loop condition is tested before each iteration, and therefore we initialize `next` to *false* to enforce the checking of the loop condition before any inner instance is created. Note that in the following code we leave out all guards and actions inherited from abstract events.

MACHINE Level_03_Sub_schedule_CF
VARIABLES
……
at_outer_schedule_loopcounter
au_outer_schedule_next
……
EVENTS
Event $act\_Factory\_schedule\_activate \mathrel{\widehat{=}}$
refines $act\_Factory\_schedule\_activate$
any
pid
$child$
where
… : ……
then
… : ……
act5 : $au\_outer\_schedule\_next(child) := FALSE$
act6 : $at\_outer\_schedule\_loopcounter(child) := 0$
end
Event $act\_Factory\_schedule\_complete \mathrel{\widehat{=}}$
refines $act\_Factory\_schedule\_complete$
any
pid
$child$
$inners$
where
… : ……
grd6 : $inners = dom(outer\_inner\_schedule \rhd \{child\})$
grd7 : $ran(inners \lhd state\_inner\_schedule) \subseteq \{completed\}$
grd8 : $at\_outer\_schedule\_loopcounter(child) \geq max\_retry$
then
act1 : $state\_outer\_schedule(child) := completed$
act2 : $tk\_Factory\_schedule\_gate(pid) := tk\_Factory\_schedule\_gate(pid) + 1$
end
Event $act\_Factory\_schedule\_next \mathrel{\widehat{=}}$
any
pid
$child$
$inners$
where
… : ……
grd6 : $inners = dom(outer\_inner\_schedule \rhd \{child\})$
grd7 : $ran(inners \lhd state\_inner\_schedule) \subseteq \{completed\}$
grd8 : $at\_outer\_schedule\_loopcounter(child) < max\_retry$
then
act1 : $au\_outer\_schedule\_next(child) := TRUE$
end
Event $evt\_schedule\_start \mathrel{\widehat{=}}$
any
pid
$parent$
$outer$
where
… : ……
grd8 : $au\_outer\_schedule\_next(outer) = TRUE$
then
… : ……
act10 : $au\_outer\_schedule\_next(outer) := FALSE$
act11 : $at\_outer\_schedule\_loopcounter(outer) := at\_outer\_schedule\_loopcounter(outer) + 1$
end
END

3.6 Compensation

Compensation starts with the execution of a compensation throwing event. Each throw event has a scope, and only activities within this scope can be compensated. An activity is within the scope of a compensation throw event if (1) the activity is contained in the same process as the event; or (2) the event is contained in a compensation event sub-process of the process that contains the activity.

Usually, a compensation throw event contains a reference to the activity to be compensated. However, it is left open in the official BPMN document whether all completed instances of the activity inside the scope will be compensated, or only the last instance is to be compensated. In our translation, all completed instances are compensated. An activity can be compensated only after being completed. If a compensation trigger is thrown when an activity instance is still active, the compensation handler of the activity instance is not triggered and, in this translation, will never be triggered unless another compensation trigger is thrown again in the future.

The following code shows how the `shipping` activity in Figure 3 is compensated. `au_Retailer_shipcomp_insts` records the activity instances which need to be compensated, and `au_Retailer_shipcomp_sync` is used to wait for the completion of the involved compensations before passing tokens to outgoing flows.

MACHINE Level_04_Retailer_Data
VARIABLES
  .....
  au_Retailer_shipcomp_sync
  au_Retailer_shipcomp_insts
  .....
EVENTS
Event $evt\_Retailer\_shipcomp\_activate \mathrel{\widehat=}$
extends $evt\_Retailer\_shipcomp\_activate$
  any
    pid
    to_comp
  where
    grd1 : $pid \in instances\_Retailer$
    grd2 : $state\_Retailer(pid) = active$
    grd3 : $tk\_Retailer\_gate\_shipcomp(pid) > 0$
    grd4 : $au\_Retailer\_shipcomp\_sync(pid) = FALSE$
    grd5 : $to\_comp \subseteq instances\_ship$
    grd6 : $to\_comp = dom(parent\_ship \rhd \{pid\}) \cap dom(state\_ship \rhd \{completed\})$
  then
    act1 : $tk\_Retailer\_gate\_shipcomp(pid) := tk\_Retailer\_gate\_shipcomp(pid) - 1$
    act2 : $au\_Retailer\_shipcomp\_sync(pid) := TRUE$
    act3 : $au\_Retailer\_shipcomp\_insts(pid) := to\_comp$
  end
Event $evt\_Retailer\_shipcomp\_complete \mathrel{\widehat=}$
extends $evt\_Retailer\_shipcomp\_complete$
  any
    pid
  where
    grd1 : $pid \in instances\_Retailer$
    grd2 : $state\_Retailer(pid) = active$
    grd3 : $au\_Retailer\_shipcomp\_sync(pid) = TRUE$
    grd4 : $ran(au\_Retailer\_shipcomp\_insts(pid) \lhd state\_ship) \subseteq \{compensated\}$
  then
    act1 : $au\_Retailer\_shipcomp\_sync(pid) := FALSE$
    act2 : $tk\_Retailer\_shipcomp\_chargecomp(pid) := tk\_Retailer\_shipcomp\_chargecomp(pid) + 1$
    act3 : $au\_Retailer\_shipcomp\_insts(pid) := \varnothing$
  end
Event $act\_Retailer\_shipcomp \mathrel{\widehat=}$
refines $act\_Retailer\_shipcomp$
  any
    pid
    $child$
  where
    grd1 : $pid \in instances\_Retailer$
    grd2 : $child \in instances\_ship$
    grd3 : $state\_ship(child) = completed$
    grd4 : $parent\_ship(child) = pid$
    grd5 : $child \in au\_Retailer\_shipcomp\_insts(pid)$
  then
    act1 : $state\_ship(child) := compensated$

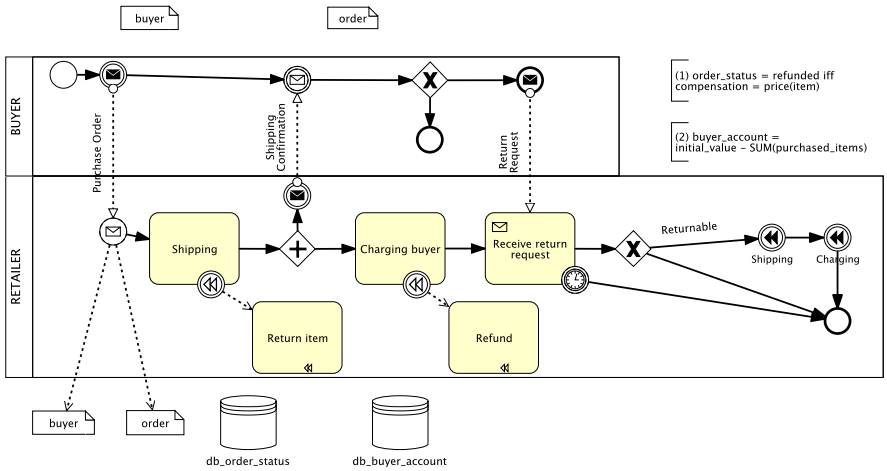

Fig. 3. A BPMN model for an online retailer.

act2 : $db\_order\_status(at\_Retailer\_order(pid)) := returned$

end

END

4 Consistency of business processes

We can use the Rodin toolset to examine the generated Event-B models for properties such as deadlock and livelock. In this section we show how we may gain further confidence in the correctness of the BPMN model by stating and proving application-level properties as invariants within the Event-B model. We use the online retailer model in Figure 3 as an example. The BPMN contains two extra annotations in the top right corner. These are extra application-level consistency conditions on the BPMN model. We anticipate these conditions to be defined by the developer and treated by the implementor as further constraints on the model. We show how we take account of them within the Event-B translation.

The online retailer model starts with the buyer, at which point a new instance of the process is generated. The buyer sends a purchase order to the retailer, which contains order and buyer information. The retailer ships the requested item, and the buyer is then charged. If, within a specified time period, the buyer asks to return the item, and the retailer chooses to accept the return, then both the shipping and charging activities must be compensated – shipping by the return of the item and charging by sending a refund to the buyer. The consistency of the information maintained about the order status and the buyer account must be maintained by this process.

The BPMN compensation event passes control to an associated compensating activity (Return item and Refund in our example.) The purpose of the

compensating activity is to "undo" an earlier part of the workflow. A precise specification of the behaviour of this activity is usually left to a later stage in the development process.

The text annotations we investigate here, such as (1) and (2) in Figure 3, give the BPMN developer the opportunity to provide a more precise specification of required properties of this subsequent development. Text annotation (1) in states that the order status is refunded if and only if the compensation paid is equal to the price of the item. Translating annotation (1) extends the Event-B refinement hierarchy with a new machine containing a new variable *compensation* and an additional invariant. The variable records the compensation paid in each instance of the retailer process. The consistency invariant introduced is formalized as

$$\begin{array}{l}\forall pid \cdot pid \in instances\_Retailer \Rightarrow \\ \quad (db\_order\_status(at\_Retailer\_order(pid)) = refunded \Leftrightarrow \\ \quad (compensation(pid) = price(at\_Retailer\_order(pid))))\end{array}$$

where the order is marked as *refunded* only when the compensation paid is equal to the price of the goods ordered. The event generated from the refund activity is also extended to record the compensation paid on that order. The new event is shown below with `act4` as the additional action.

Event $act\_Retailer\_chargecomp \mathrel{\widehat{=}}$
extends $act\_Retailer\_chargecomp$
any
pid
$child$
where
grd1 : $pid \in instances\_Retailer$
grd2 : $child \in instances\_charge$
grd3 : $state\_charge(child) = completed$
grd4 : $parent\_charge(child) = pid$
grd5 : $child \in au\_Retailer\_chargecomp\_insts(pid)$
then
act1 : $state\_charge(child) := compensated$
act2 : $db\_buyer\_account(at\_Retailer\_buyer(pid)) := db\_buyer\_account(at\_Retailer\_buyer(pid)) + price(at\_Retailer\_order(pid))$
act3 : $db\_order\_status(at\_Retailer\_order(pid)) := refunded$
act4 : $compensation(pid) := price(at\_Retailer\_order(pid))$
end

The second annotation in Figure 3 is a property over all instances of processes. The value in the account of any buyer should be the initial value of the account less any purchased items. Translating annotation (2) again adds a new machine to the model, which includes the invariant

$$\begin{array}{l}\forall b \cdot (b \in BUYERS \Rightarrow \\ \quad (db\_buyer\_account(b) = initial\_buyer\_account(b) - \\ \quad Sum(ran(dom(at\_Buyer\_buyer \rhd \{b\}) \lhd at\_Buyer\_order) \\ \qquad \cap \\ \qquad dom(db\_order\_status \rhd \{charged, returned\}))))\end{array}$$

in which the clause $ran(dom(at\_Buyer\_buyer \rhd \{b\}) \lhd at\_Buyer\_order)$ identifies all orders placed by buyer b. This is restricted to orders with status *charged*

or *returned* by the clause $dom(db\_order\_status \rhd \{charged, returned\})$. Order status *returned* identifies those orders which have been returned but not yet refunded, and therefore still need to be included in our invariant.

Proofs. The first property results in 28 proof obligations, of which 16 are automatically discharged. The other proof obligations require expert human intervention. The second property is considerably more complex and therefore results in 582 proof obligations, of which 300 are automatically discharged. The proving of both properties requires the discovery and use of auxiliary invariants as lemmas. For the second property, a total number of 88 additional invariants are added. Currently, we need to manually discover these lemmas. However, we observe that 30 lemmas express relations between token quantities on different sequence flows, e.g., if the incoming flow of `Charging buyer` has tokens then the incoming flow of `Shipping` cannot have tokens. Such information can be obtained by an automated static analysis on the control flow of the model. Therefore, it is possible to automatically discover these 30 lemmas. In future work we will also investigate the possibility of discovering other kinds of lemmas. Furthermore, we observe a highly repeated pattern in the proofs that involves case splitting to distinguish process instances. Such patterns can be implemented as proof strategies customized for proving a certain class of invariants.

5 Enhancement of processes models using patterns

When a property is violated by a model, it is possible that the model contains undesired behavior which can be removed by further constraining the model via refinement steps. We may directly perform such steps on the Event-B translation of the model in order to verify whether such refinement steps are valid, before making changes to the original model. Moreover, refinements in Event-B can be done automatically using patterns.

Event-B patterns [3, 10, 11, 7] are a means of expressing reusable modeling structures and managing effort by promoting proof re-use. In this example, we use the type of pattern presented in [10], which provides a controlled way of extending an Event-B development with a pre-validated refinement step. Since the refinement step between the abstract and the concrete pattern machines has been proved in advance, any application of the pattern results in a new, fully-proved, refinement step. The approach is automated as a plug-in for the Rodin platform ([9]). In the example we present below, a pattern is used to correct a previously discovered omission in a specification.

We use the shift worker scheduling process in Figure 1 as an example. The process depicted in Figure 1 contains a timing-related fault[4], which can lead to an inconsistency in the data maintained by the business process. It is caused by the use of the timeout at the point where worker responses are received. It arises when a request is sent to a potential worker but no reply is received within the

[4] Note that the shift worker scheduling process is a simplified version of a BPMN workflow proposal, and not a part of any real world system.

allotted time. Another request is therefore sent to another available candidate. He may accept and be assigned to the shift, after which an accept message is received from the first worker. Now the first worker thinks he is the replacement, but in fact the second has been chosen. We discovered this error using the Rodin model checker *ProB*: we added an invariant expressing the property that at most one worker accepts the shift at any given time. ProB found an erroneous execution within a short time.

When translated into Event-B, the flaw present in the described scenario can be corrected using the timed error recovery pattern, shown in Figure 4 and presented in full in [6]. It is designed to be applied to any model in which late messages are not properly processed. When applied, a further refinement level is added to the Event-B development. This new level contains the error recovery behavior which ensures adequate processing of any late messages.

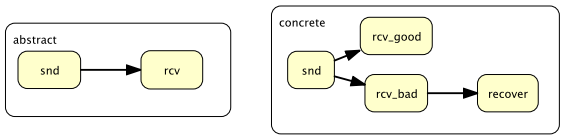

Fig. 4. Structure of the timed error recovery pattern.

The concrete machine in the pattern separates normal and recovery behavior by distinguishing the receipt of messages before and after the deadline and handling these two cases separately. Late responses are followed with a compensation event, which may be further refined depending on the way in which recovery is implemented.

Applying the pattern requires the identification of the activities in the workflow where the timer is set and the (on time or late) replies are received. These activities are then matched with the sending and receiving events in the pattern abstract machine (snd and rcv in the abstract machine in Figure 4). The pattern variables must also be matched to the appropriate variables within the development.

In the Shift Worker Scheduling model in Figure 1, the timer is set at the task `Select an available worker`. The `Receive response` action is the point at which messages are received. The application of the pattern introduces a new event corresponding to rcv_bad (the arrival of late replies) and given below.

Event $act\_schedule\_response\_late \mathrel{\widehat{=}}$
  any
    pat_m
  where
    grd1 : $pat\_m \in q\_rcv$
    grd2 : $tt(pat\_m) < now$
  then
    act1 : $q\_rcv := q\_rcv \setminus \{pat\_m\}$

act2 : $q\_comp := q\_comp \cup \{pat\_m\}$
act3 : $timercvd := timercvd \cup \{pat\_m \mapsto now\}$
end

In this event, pat_m is the message and q_rcv and q_comp are the messages queued for reception and compensation respectively. The second guard requires that the current time (now) is later that the target arrival time of the message ($tt(pat\_m)$). On arrival, the message moves to the queue for compensation and the time at which it is received is recorded.

The recover event refines the rcv event. As well as retaining all the functionality of rcv, it places the compensated message in the database of consistent messages. The precise nature of the compensation activity required will vary according to the particular activity it is compensating, so the recover event acts as a placeholder for a fuller description of compensation within the workflow, which may be added (perhaps by the application of a more specific pattern) in further refinements.

The ability to automatically add pre-validated refinement steps to generated Event-B models can be used to support BPMN development. In our example, the refinement step made to the Event-B translation can be re-constructed in the original model by introducing a parallel thread to detect and react to late messages. Such reconstruction can be achieved either through a reverse translation procedure from Event-B back to BPMN, or by building up a repository of BPMN refinement patterns corresponding to Event-B patterns. We will explore both possibilities in future work.

6 Related Work

Unlike our work presented in this paper, existing work in this area is largely concentrated on an examination of BPMN control flow, and does not consider data modeling. Most of them also consider only a small fraction of the BPMN language, and put many restrictions on models that can be analyzed. [8] uses Petri nets to formalize and analyze BPMN control flows while abstracting from data information. The approach requires 1-safeness (i.e., having at most one token on any sequence flow) in order to analyze exception handling for sub-processes. On the contrary, we wish to be able to model multiple processes instances. [17] also uses Petri nets to treat transactions and compensations, and also limits his treatment to pure control-flow aspects of models. [20] focuses on the control flow aspects of BPMN in a mapping to the formal workflow language YAWL. [18] formalizes a subset of BPMN in CSP but does not consider features such as compensations and correlation. It is also unclear how data is modeled. The recent work to be published in [4] gives a precise and well-structured semantics of BPMN using Abstract State Machines. This is, to our knowledge, the largest coverage of BPMN besides ours. In [16] a subset of BPMN is translated into the process algebra COWS in order to exploit the stochastic extensions to perform quantitative reasoning on BPMN processes. In [19] the authors present a relative timed semantics for BPMN using CSP. This approach concentrates on the

correctness of control flow. In [13] the authors outline a framework for the verification of business processes suggesting the use of TLA+ as well as Petri nets. In [14] the authors capture a number of BPMN Service Interaction Patterns as UML models.

7 Conclusions and further work

We have presented our work on the formal analysis of business process models through a translation into Event-B. The translation can be fully automated and covers a large set of BPMN features. In particular, we consider the modeling of both control flow and data flow. We showed how properties can be verified by the help of automated provers in the Rodin platform, and also showed how Event-B patterns can be used to support the correction of design errors.

Subsequent work on this topic will be driven by our long-term goal: to allow the BPMN developer to benefit from the improved analytic power of formal methods (and in particular Event-B) while adding minimal extra complexity to the design process. As an initial step, we expect to implement the presented translation as a plug-in to the Rodin toolkit.

The two proofs of possibility presented in Sections 4 and 5 point to two different enhancements to the BPMN development method which we could aim to support. The first, of adding annotations to BPMN, will require a definition of the annotation language and a formalization and implementation of the rules to translate these annotations to Event-B. The second, of using patterns to transform the generated Event-B models, would benefit from the definition of the inverse translation from Event-B to BPMN. Note that this is not the same as a *general* Event-B to BPMN translation, as we would be able to impose relatively strong conditions on the structure (and indeed syntax) of source Event-B models in this translation. We could also develop a library of BPMN transformations together with their Event B patterns, and offer developers a choice from this library in response to identified problems.

Both these approaches suffer from the high number of proof obligations which must be manually discharged. We will therefore work on the automatic discovery of auxiliary lemmas to assist in the proof task. Furthermore, we plan to design and implement various proof strategies tailored for specific classes of proof obligations in order to increase the number of automatically discharged proofs.

Finally, we expect to explore the use of model-checking as a means of providing rapid feedback to the developer on the reason for a failed proof. The challenge here is to provide feedback in a way meaningful to the developer.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments and to Deploy project colleagues for their continuing fruitful collaboration. This work has been supported by the EC FP7 Integrated Project *Deploy* and the EPSRC grant *TrAmS* (EP/E035329/1).

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. volume 4260 of *LNCS*, pages 588–605. Springer, 2006.
3. E. Ball and M. Butler. Event-b patterns for specifying fault-tolerance in multi-agent interaction. volume 5454 of *LNCS*, pages 104–129. Springer, 2009.
4. E. Börger and O. Sörensen. *Handbook of database technology*, chapter BPMN Core Modeling Concepts. Inheritance-Based Execution Semantics. Springer, 2010. To appear.
5. J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth. Formal modelling and analysis of business information applications with fault tolerant middleware. In *Proc. of ICECCS 2009*, pages 68–77. IEEE Computer Society, 2009.
6. J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for modelling time and consistency in business information systems. In *Proc. of IECCS 2010*, pages 105–114. IEEE Computer Society, 2010.
7. D. Cansell, D. Méry, and J. Rehm. Time constraint patterns for Event B development. volume 4355 of *LNCS*, pages 140–154. Springer, 2007.
8. R. M. Dijkman, M. Dumas, and C Ouyang. Formal semantics and analysis of BPMN process models using Petri nets. Available at `http://eprints.qut.edu.au/7115/01/7115.pdf`.
9. A. Fürst. Design patterns in Event-B and their tool support. Master's thesis, ETH Zürich, 2009.
10. T. S. Hoang, A. Fürst, and J.-R. Abrial. Event-B Patterns and Their Tool Support. In *Proc. of SEFM 2009*, pages 210–219. IEEE Computer Society, 2009.
11. A. Iliasov. *Design Components*. PhD thesis, Newcastle University, 2008.
12. S. Kätker and S. Patig. Model-driven development of service-oriented business application systems. In *Wirtschaftsinformatik (1)*, volume 246 of *books@ocg.at*, pages 171–180. Österreichische Computer Gesellschaft, 2009.
13. C. Masalagiu, W.-N. Chin, Ş. Andrei, and V Alaiba. A rigorous methodology for specification and verification of business processes. *Formal Aspects of Computing*, 21(5):495–510, 2009.
14. O. Nicolae, M Cosulschi, A Giurca, and G. Wagner. Towards a BPMN semantics using UML models. In *Business Process Management Workshops*, volume 17 of *LNBIP*, pages 585–596. Springer, 2009.
15. OMG. Business process model and notation (BPMN), FTF beta 1 for version 2.0. Available at `http://www.omg.org/spec/BPMN/2.0/Beta1/PDF/`.
16. D. Prandi, P. Quaglia, and N Zannone. Formal analysis of BPMN via a translation into COWS. In *Proc. of COORDINATION 2008*, volume 5052 of *LNCS*, pages 249–263. Springer, 2008.
17. T. Takemura. Formal semantics and verification of BPMN transaction and compensation. In *Proc. of APSCC 2008*, pages 284–290. IEEE, 2008.
18. P. Y. H. Wong and J. Gibbons. A process semantics for BPMN. In *Proc. of ICFEM 2008*, volume 5256 of *LNCS*, pages 355–374. Springer, 2008.
19. P. Y. H. Wong and J. Gibbons. A relative timed semantics for BPMN. *Electronic Notes in Theoretical Computer Science*, 229(2):59–75, 2009.
20. J.-H. Ye, S.-X. Sun, L. Wen, and W. Song. Transformation of BPMN to YAWL. In *CSSE (2)*, pages 354–359. IEEE Computer Society, 2008.

A Translation of control flows

The following shows how the end event[5] in the `FACTORY` process in Figure 1 are translated. This end event has two incoming flows. Every `tk_Factory_xx_xx` is a token function that maps each `Factory` instance to the number of tokens on the respective flow. The end event can be executed only if the containing `Factory` instance is still active. This is reflected by the guard `grd2` in each Event-B event.

MACHINE Level_01_Factory_CF
SEES Data_Types, Processes
VARIABLES
.....
`tk_Factory_schedule_gate`
`tk_Factory_gate_assign`
`tk_Factory_gate_end`
`tk_Factory_assign_end`
.....
INVARIANTS
... : .....
inv4 : $tk\_Factory\_schedule\_gate \in instances\_Factory \rightarrow \mathbb{N}$
... : .....
EVENTS
Event $evt\_Factory\_end\_in1 \mathrel{\widehat{=}}$
any
pid
where
grd1 : $pid \in instances\_Factory$
grd2 : $state\_Factory(pid) = active$
grd3 : $tk\_Factory\_assign\_end(pid) > 0$
then
act1 : $tk\_Factory\_assign\_end(pid) := tk\_Factory\_assign\_end(pid) - 1$
end
Event $evt\_Factory\_end\_in2 \mathrel{\widehat{=}}$
any
pid
where
grd1 : $pid \in instances\_Factory$
grd2 : $state\_Factory(pid) = active$
grd3 : $tk\_Factory\_gate\_end(pid) > 0$
then
act1 : $tk\_Factory\_gate\_end(pid) := tk\_Factory\_gate\_end(pid) - 1$
end
... ...
END

B Translation of messages

The following code shows how message exchanges are translated for the model in Figure 1.

MACHINE Level_07_Messages_Factory_Worker
REFINES Level_06_Worker_Data
SEES Data_Types, Processes, Messages
VARIABLES
.....
`sent_req`
`buf_req`
`fld_req_sid`
.....
INVARIANTS
inv1 : $sent\_req \subseteq REQ\_MESSAGES$

[5] An end event is a sink for tokens, i.e., it only consumes tokens without generating any. Reaching an end event does not necessarily imply the completion of process execution.

```
    inv2 :  buf_req ⊆ sent_req
    inv3 :  fld_req_sid ∈ sent_req → SESSION_IDS
    ... :  .....
EVENTS
Event  evt_schedule_request ≙
refines  evt_schedule_request
    any
        msg
        ...
    where
        ... :  .....
        grd4 :  msg ∈ REQ_MESSAGES
        grd5 :  msg ∉ sent_req
        ... :  .....
    then
        ... :  .....
        act3 :  fld_req_sid(msg) := at_Factory_sid(parent)
        act5 :  buf_req := buf_req ∪ {msg}
        act6 :  sent_req := sent_req ∪ {msg}
    end
Event  act_schedule_response_complete ≙
refines  act_schedule_response_complete
    any
        .....
        msg
    where
        ... :  .....
        grd10 :  msg ∈ buf_res
        grd11 :  fld_res_sid(msg) = at_Factory_sid(parent)
        ... :  .....
    then
        ... :  .....
        act5 :  buf_res := buf_res \ {msg}
    end
Event  evt_Worker_start ≙
refines  evt_Worker_start
    any
        pid
        msg
    where
        ... :  .....
        grd5 :  msg ∈ buf_req
        grd6 :  fld_req_sid(msg) ∉ ran(at_Worker_sid)
    then
        ... :  .....
        act5 :  at_Worker_sid(pid) := fld_req_sid(msg)
        act7 :  at_Worker_accept(pid) := FALSE
        act8 :  buf_req := buf_req \ {msg}
    end
END
```

C Translation of loop sub-processes

The code below shows how the "collapsed view" of the loop sub-process in the `FACTORY` process in Figure 1 is translated. Two separate Event-B events represent respectively the starting and the completion of the sub-process. At this level the completion of an outer instance is non-deterministic.

```
MACHINE  Level_01_Factory_CF
EVENTS
Event  act_Factory_schedule_activate ≙
    any
        pid
        child
    where
        grd1 :  pid ∈ instances_Factory
        grd2 :  state_Factory(pid) = active
        grd3 :  tk_Factory_start_schedule(pid) > 0
        grd4 :  child ∈ ACT_SCHEDULE_OUTER_INSTANCES
```

grd5 : $child \notin instances\_outer\_schedule$

then

act1 : $tk\_Factory\_start\_schedule(pid) := tk\_Factory\_start\_schedule(pid) - 1$

act2 : $instances\_outer\_schedule := instances\_outer\_schedule \cup \{child\}$

act3 : $state\_outer\_schedule(child) := active$

act4 : $parent\_outer\_schedule(child) := pid$

end

Event $act\_Factory\_schedule\_complete \mathrel{\widehat{=}}$

any

pid

$child$

where

grd1 : $pid \in instances\_Factory$

grd2 : $state\_Factory(pid) = active$

grd3 : $child \in instances\_outer\_schedule$

grd4 : $state\_outer\_schedule(child) = active$

grd5 : $parent\_outer\_schedule(child) = pid$

then

act1 : $state\_outer\_schedule(child) := completed$

act2 : $tk\_Factory\_schedule\_gate(pid) := tk\_Factory\_schedule\_gate(pid) + 1$

end

END