From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods

Jana Koehler Giuliano Tirenni
IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
email: {koe | tir}@zurich.ibm.com

Santhosh Kumaran
IBM T.J. Watson Research Center
PO BOX 218 RTE 134
Yorktown Heights, NY 10598, USA
email: sbk@us.ibm.com

Abstract

Today's business applications and their underlying process models are becoming more and more complicated, making the implementation of these processes an increasingly challenging task. On the one hand, tools and methods exist to describe the business processes. On the other hand, different tools and method exist to describe the IT artifacts implementing them. But a significant gap exists between the two. To overcome this gap, new methodologies are sought.

In this paper, we discuss a pattern-based modeling and mapping process. Starting from a business process model, which emphasizes the underlying structural process pattern and its associated requirements, we map this model into a corresponding IT model based on nondeterministic automata with state variables. Model checking techniques are used to automatically verify elementary requirements on a process such as the termination and reachability of states. Using an example involving coupled, repetitive activities we discuss the advantages of an iterative process of correcting and refining a model based on insights gained in the interleaved verification steps.

1 Introduction

Business process modeling and reengineering have been longstanding activities in many corporations in recent years. Most *internal* processes have been streamlined and optimized, whereas the *external* processes have only recently become the focus of business analysts and IT middleware providers. The full integration of all processes within a supply network is an upcoming challenge that has to be addressed by network partners and solution providers. The static integration of inter-enterprise processes as common in past years can no longer meet the new requirements of customer orientation, flexibility and dynamics of cooperation. A dynamic integration of processes becomes mandatory, which is obviously much harder to achieve.

A major prerequisite for dynamic integration is the *reliability* and *flexibility* of the processes involved. Although companies will work together and might even form what is known as a virtual enterprise, they will usually not reveal the details of their internal processes to each other. Furthermore, even integrated business processes are subject to change, i.e., a company commits itself to providing a process with a certain input and result, but still wants to retain the freedom to internally reorganize and optimize its processes as necessary. Consequently, traditional business process modeling techniques fail to describe loosely coupled business processes, the structure of which is opaque or completely hidden. New modeling methods are sought that allow the challenges of reliability and flexibility to be addressed.

The new paradigm of *web services* [3] allows companies to describe the external structure of their processes and how they can be invoked via XML-based messaging using WSDL, the web services description language. Standards such as RosettaNet [20] and ebXML [8] provide means to describe the choreography of message exchanges in an implementation-independent way, which is crucial to enabling two trading partners to communicate without either placing constraints on the other's implementation.

WSFL [17] and XLANG [22] are two XML-based choreography languages for explicitly describing service flows within a business process that may be composed of multiple web services. They assume an execution environment ("orchestration engine") for programs written in the language. WSDL is used with these languages to provide the externally visible description of the individual web services in the composition.

The new languages make the communication and interaction structure of a business process explicit, which is a prerequisite for the flexible composition of processes. Furthermore, languages such as WSFL and XLANG possess an operational semantics that enables the orchestration engine to directly execute the flow models specified in the language. This makes them very different from other business process modeling languages such as UML [9], which allow a business process to be described in detail, but the specification has then to be translated into an IT implementation by a human IT expert. Although the various views provided by UML on a business process strongly support this translation process, mismatches and communication misunderstanding can still happen. The flow models reduce the need to directly translate a business process model into a low-level programming language. Instead, a business process model is mapped into an IT flow model that can be regarded as a form of a high-level implementation language—the model is directly executable or can be automatically translated into another target language. As such, the IT flow model provides a useful abstraction for business process mapping.

Nevertheless, how the mapping from the business model to the IT flow model will proceed, is still an unexplored issue. The new approach has the potential of yielding executable business process specifications, but we cannot expect that flow models will be written by hand. Instead they have to be derived from the higher-level business process specification. By addressing this derivation process, we hope that the continuity of the design and implementation process can be further improved and that the gap between the business analyst's view and the IT specialist's view can be further reduced.

The key problem is the following. Given a business process model, for example in UML, and an IT model, for example in WSFL, how can one know that they are consistent with each other? In other words, how can one determine whether the IT model actually preserves the essential requirements of the business process model?

This is the question we explore in this paper, which is organized as follows. In Section 2, we discuss the representation of business process models and the need to make process requirements explicit. Section 3 explores the pattern-based mapping of business models into automata-based IT models and the translation of process requirements into logical formulas, which can then be automatically verified using model checking techniques. Section 4 describes an iterative process of model correction/refinement and subsequent model checking until all process requirements have been verified. In Section 5, we briefly review related work, and conclude with a summary and an outlook on future work in Section 6.

2 Business Process Requirements

The various approaches for business process modeling and the tool sets implementing them have numerous features in common. They all try to capture *which* business tasks are going to be automated, *where* the automating system is going to be deployed, *who* will use it, and *how* it will integrate with other systems. In a nutshell, we find the following typical elements in a business process modeling language [9]:

- The *organizational model* describes the roles and areas of responsibilities within an organization with respect to the activities of a business process. It presents a more static view of a process.
- The *control flow* describes the order of execution and the dependencies among the various activities.
- The *data flow* describes how the business entities (or artifacts) are manipulated by the various activities.
- *Use cases* describe the context of a business process and its externally visible behavior.
- *Collaboration diagrams* can further document how business agents and artifacts work together to perform a function.

All this information together provides an accurate semi-formal specification of the business process. In particular, the process requirements—*when* an activity executes, *how* often it will execute, and *under what conditions* it will end—can usually only be described informally in the form of use cases or textual descriptions in some natural language. But precisely this information is crucial for a business process to perform as a reliable partner within a larger, dynamic business environment.

In order to guarantee reliability, the process model must contain a specification of its requirements that includes the following properties:

- the requirement specification must be unambiguous,
- it must carry over to the IT model,
- it must be automatically verifiable in the IT model.

Only under these conditions will we be able to guarantee that an IT model implements a given business process model. Although a candidate for an unambiguous formal specification exists, namely mathematical logic, it is hard to imagine that a business analyst will use it to describe the process requirements. Therefore, "wrappings" into more friendly formalisms have recently been proposed, see for example [2].

Here, we explore another approach. We focus on typical *structural patterns* that occur frequently in business processes. We map them to typical automata structures that provide the semantics of the IT model and the orchestration engine executing these models. We define typical *properties* of these patterns that are of particular interest and that are subject to an automatic verification. We do not aim at having the business analyst do this work, but by providing typical patterns we hope to facilitate communication between business and IT expert. Furthermore, we believe that an IT model will not be a one-to-one mapping of a business model, but more a consistent refinement. The properties will provide additional information, for example the choice of a particular web service to implement a given business activity within the process. In summary, we make the requirements of the business process explicit—as only then can we verify whether subsequent refinements of the IT model will preserve these requirements.

In the following we discuss our approach with the help of an example introduced below.

3 Formalization and Verification of Process Requirements

A very common pattern occurring in business processes is that of *loosely coupled, repetitive* activities. The *Request for Quotation* (RFQ) process is a typical instance of this pattern: a request for quotation is sent out to various business partners; the returned quotes are received and collected, and finally the best quote is selected. In this example, sending and receiving quotes are typical repetitive activities. Although such a pattern is very frequent, it poses a real challenge to modern work flow management systems [18]. Figure 1 illustrates this pattern.

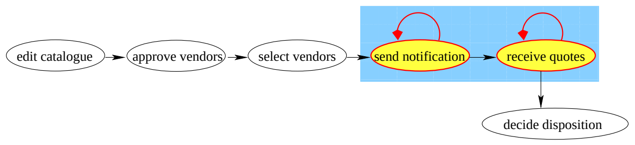

Figure 1. A pattern of loosely coupled, repetitive activities within the context of a business processes.

In the remainder of this section, we will start from an informal (in natural language) but more detailed description of this process. Then we map the business process model to a formal IT model based on nondeterministic automata. This step is facilitated by explicitly stating the business process pattern. Next, we translate the business process requirements into a formal language stating properties of the corresponding IT automata model.

Once we have a formal IT model plus the properties corresponding to the process requirements, we can automatically check whether these properties are satisfied in the model with the help of a *model checking* procedure [6, 1] that explores all possible traces of behavior described in the IT model. In case of any violations of properties or undesired behaviors, the model checking procedure will output a counterexample, which demonstrates the courses of wrong behavior. An analysis of this counterexample provides information that helps to correct and refine the IT model.

3.1 RFQ Process

The RFQ process starts when a purchasing agent sends out a request for quotation to a set of selected vendors. After having received a quote from each vendor, or after a deadline has elapsed, an evaluation process to select the best quote is begun. The business process is composed out of the following activities:

1. The *send notification* activity models the repetitive sending of the request for quotation from the purchasing agent to a preselected set of vendors.

2. The *receive quotes* activity occurs when the purchasing agent receives a quote from a vendor.

3. The *decide disposition* activity compares the various quotes with each other. It takes place after either a quote from each vendor has been received or a deadline has elapsed; at least one quote must have been received.

Identifying this process as an instance of the "coupled, repetitive activities" structural pattern is an important step during the business process modeling. It allows the requirements of the various activities to be made explicit:

- *Send* repeats a certain number of times until all selected vendors have been notified. In the process model (and of course in the corresponding IT model) we cannot foresee how many times this activity will be executed.

- *Receive* is also a repetitive activity, but it is definitely linked with *send*. Assuming that the orchestration engine provides us with some message correlation process, we will be able to check whether a quote received corresponds to a request sent. In this case, a corresponding event will occur and the quote is added to the collection of quotes answering the previous *send* activity. It is obvious that we do not expect more quotes to arrive than requests have been sent (those non-correlated messages are dealt with in a separate process), but there can be fewer if some vendors did not respond. It is therefore natural to associate not only a linking condition with this activity, but also a deadline, i.e., a timeout event will end the waiting for arriving quotes when the deadline set in the send activity has elapsed.

- *Decide* is not a repetitive activity in this model and takes place after send and receive have terminated. This can happen if either a timeout occurred or all expected quotes have arrived. An evaluation makes sense only if at least one quote has been received.

To each structural pattern on the *business process model* side, we associate one or more possible patterns on the *IT model* side based on nondeterministic communicating automata. We emphasize that the choice of an IT model is not unique, but that many possible control patterns may be applicable in order to implement a business process. We believe that providing pattern libraries extracted from best practices nevertheless has the potential in many cases to facilitate and even partially automate the difficult process of mapping a business model into an IT model.

3.2 Pattern-based Mapping of Models

For our example involving two loosely coupled, repetitive activities, we choose a model based on two linked counting automata as shown in Figure 2.[1]

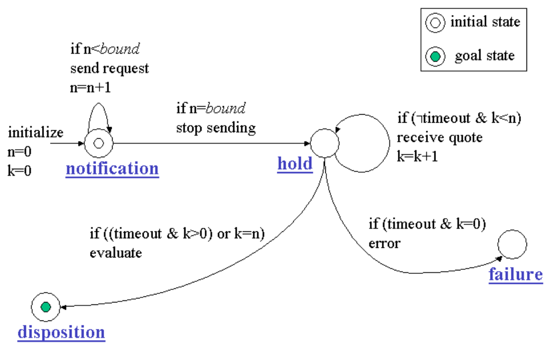

Figure 2. An initial automata model of two coupled, repetitive activities.

We use a general nondeterministic automata model with state variables and transition guards. A state in the automaton is an assignment of values to state variables. A state variable *activity* represents the business activity that is executed by some agent (human or system). The *activity* variable can take the possible values *initialize*, *send request*, *stop sending*, *receive quote*, *error*, *evaluate*. A state variable *stage* represents the stage of the business process in which an activity is executed. The *stage* variable can take the possible values *notification*, *hold*, *disposition*, and *failure*. The state variables n and k represent the number of sent and received quotes, respectively. They are shared by the two (sub-)automata modeling the *send* and *receive* activities and thereby allow these two automata to link to each other. A constant *bound*, which has an arbitrary, but fixed value represents the maximum (finite) number of requests that will be sent. All states with *stage=notification* are marked as the initial states of the process fragment we consider, all states with *stage=disposition* represent goal states. The state variable *timeout* is a Boolean variable that can change its value in a nondeterministic manner.[2]

The automaton interacts with the state variables in two ways:

1. Guards: A transition cannot occur unless a condition on one or more state variables holds.

2. Assignments: A transition can modify the value of one or more state variables.

A transition from one state to another occurs if and only if its guard condition is true. A transition leads to another state by modifying the value of one or more state variables. There are two more remarks we should make regarding this model: First, we do not explicitly model the occurrence of external events, e.g., as the output of some other process, but capture their possible occurrence by the nondeterminism in the automaton. Second, the symbolic representation we use allows us to speak about sets of states instead of single states. The circles in Figure 2 (and all subsequent figures) represent sets of states (more precisely, sequences of states) during which the *stage* variable does not change its value, but other state variables do change. For example, the process remains in *stage=notification* as long as *activity=send request* and the guard condition $n = bound$ are not satisfied. By *unfolding* the automaton, we obtain the *state graph* showing the possible transitions, cf. [1].

The *initialize* activity models the entry into the process fragment we wish to implement with our IT model. As we adopted the pattern of two linked counting automata, it is accompanied by the initialization of the two counter variables n and k. The IT model also contains an explicit *failure* state, which was not foreseen in the original business model, but which represents a refinement of it following from an analysis of the *hold* stage. The process remains in this stage as long as either all quotes have been received or the timeout occurred. Furthermore, the *evaluation* activity

[1]Alternative IT models will be discussed at the end of Section 4.

[2]In fact, the *stage* variable is even redundant in the model. We leave it only for the purpose of illustrating the different phases in the process. This hopefully makes it easier to understand the definition of the transition relation to follow.

only makes sense if at least one quote has been received. From this information, the guard conditions on the two possible nondeterministic transitions can be extracted[3]:

- If $\neg timeout \ \wedge \ k < n$ the process loops in the hold stage.
- If $(timeout \ \wedge \ k > 0) \vee k = n$ the process enters the disposition stage.

Inspecting these two conditions, one realizes that the important case of $timeout \ \wedge \ k = 0$ is not covered in the business model. The IT model refines the business model by adding a third transition into a failure state guarded by precisely this condition. The disjunction of the three possible guard conditions forms a logical tautology—a clear indication that all possible situations have now been covered. Furthermore, we see that only one of the transitions can be satisfied in any situation, i.e., how the *hold* stage is left is deterministically defined. The only nondeterminism in this example remains in the timeout event, which is not under the control of the process we consider.

The automaton in Figure 2 is supposed to represent a refinement of the original business process model obtained by mapping the business pattern of two coupled, repetitive activities into the IT pattern of two linked counters. To verify this claim, we need to ensure that the requirements placed on the process model carry over to the IT model. The *local* requirements on each activity have been formalized manually by introducing the guard conditions on the transitions. Although one can imagine that the formulation of guard conditions can be further supported by providing guard templates with the various patterns, designing an IT model will always remain a highly skilled activity. During the design process, an IT expert needs feedback regarding the *global* consistency of her/his model. This can be achieved by examining the possible traces of behavior implied by the model and verifying *global* properties of the model. Two types of properties are of particular importance: *reachability* and *liveness*, which we will discuss in the next section.

3.3 Global Requirements on Models

A *reachability* property states that a particular situation *can* sometimes be reached. For any IT model it is of particular interest to verify the reachability of the goal state(s) and to examine the sequences of activities through which the goal state(s) can be reached.

A *liveness* property expresses that, under certain conditions, a situation will *ultimately* occur. This formulates a much stronger condition than mere reachability. With a liveness property, we require that, independent of the system behavior, a particular situation can always be reached. *Termination* of a process is a liveness property. Whereas reachability only requires that there is a trace of behavior leading to the goal state, liveness requires that the goal state is reachable via any of the possible traces of behavior.

For the pattern of linked counters, termination is of particular interest and not at all a property, which is obvious to see. This means, similarly to associating business process models with certain structural patterns and mapping these to possible IT patterns, we associate certain business process requirements with properties of the IT model that are subject to automatic verification. By identifying the initial and goal state(s) of the IT model automaton, the corresponding termination condition, i.e., the *termination in one or more designated goal states*, can be derived automatically.

In the following, we will discuss the automatic verification of the termination property in more detail. In principle, we can base the verification on two techniques: First, one can identify a property in the process, which can be described by a bounded and strictly monotonically decreasing function. By conducting a *mathematical proof*, which shows that the function will ultimately reach this bound independent of what the trace of behavior is, we can establish the termination of the process. Second, we can use a *symbolic model checking procedure* to enumerate exhaustively the set of possible reachable states underlying the process model. If each trace of behavior ends in a designated goal state, the process will terminate. In this paper, we present our experiences with the second approach.

3.4 Verifying Termination via Model Checking

So far, we used automata as the basis of the operational models by which we specify the behavior of the processes and their implementing systems that we wish to investigate. This approach is not new and has been used successfully in the development of reactive systems [11] and in the VLSI design process [21]. The next step is to use a formalism with which we can formulate precisely the properties we wish to investigate further. An ideal candidate for this is the *Computation Tree Logic* (CTL) [5]—a temporal logic especially suited for behavioral descriptions of reactive systems with *discrete* time. CTL extends standard-first order predicate logic with temporal operators and path quantifiers that help to express statements about the possible traces of behavior in a system. In the following, we will review the main constructs in CTL that allow us to verify the termination of our IT model. The reader is referred to [6, 1] for more details.

If p denotes a logical formula representing a property satisfied in a state s, then $\mathsf{X}\,p$ ("*next p*") states a property satisfied in a successor state of s, $\mathsf{G}\,p$ ("*globally p*") states a

[3]For readers unfamiliar with the basic logical notation: $\neg$ stands for the negation of a formula, $\wedge$ represents logical conjunction (both conjuncts must evaluate to *true* in order for the formula to become *true*), $\vee$ stands for the non-exclusive OR, and $\rightarrow$ means logical implication.

property satisfied in *all* future states (without saying which states), $\mathsf{F}\,p$ ("*finally p*") states a property satisfied in *some* future state. Using these operators we can express properties about *one* possible trace of behavior of the process, i.e., one path of execution of the automata. As many different traces of behavior are possible, we need path quantifiers that allow us to express the tree aspect of the behavior. A formula $\mathsf{A}\,p$ ("*for all paths*") states that *all* the executions out of the current state satisfy the property p, whereas the formula $\mathsf{E}\,p$ ("*it exists some path*") states that there is at least *one* path of execution satisfying p.

By combining temporal operators and path quantifiers we can formulate the termination property of interest:

$$\mathsf{A}\,\mathsf{G}\,\big((\textit{initial state}) \rightarrow \mathsf{A}\,\mathsf{F}\,(\textit{goal state})\big),$$

which states that it is invariantly true on all paths that if the execution starts in the initial state (or a set of states satisfying a CTL formula), it will eventually reach the goal state(s) independent of the execution path taken.

Writing CTL statements requires experience, and understanding CTL statements written by others might require even more experience—the reason why we opt for the support of the modeling process by providing predefined structural patterns.

Given the automata model in Figure 2, we can automatically derive its corresponding termination property as a CTL formula:

$$\mathsf{A}\,\mathsf{G}\,\big((\textit{stage} = \textit{notification}) \rightarrow \mathsf{A}\,\mathsf{F}\,(\textit{stage} = \textit{disposition})\big).$$

A further advantage of using CTL is the availability of numerous model checking tools that allow us to verify properties of interest automatically. In our experiments, we used the NuSMV model checker [4], which is an improved implementation of the famous and widely used SMV model checker [19]. SMV performs symbolic model checking of CTL formulas on networks of automata with shared variables. In order to use NuSMV, the automata model has to be translated into the input language of SMV. This process can be automated when starting from a representation as in Figure 2, which is limited to predefined patterns. Otherwise, it will require skill in using model checking tools. This may become a future requirement of IT architects anyway, as is already the case in the area of VLSI design.

Without going into excessive detail, the SMV representation can be summarized as follows. We declare the state variables *stage, n, k, timeout,* and *activity* as variables in the SMV program. The *bound* constant has to be assigned a specific value as SMV cannot represent arbitrary, but fixed values—which is a clear drawback when using it for termination proofs, as we want to prove termination for arbitrary bounds on the counters n and k. One can work around this limitation by repeating the model checking process for all possible bounds, but this is somewhat clumsy. As an example of this, we show the model checking process for the case of *bound = 4*.[4]

```
MODULE main
VAR
 stage : {notification, hold, disposition,
          failure};
 n : 0..4;
 k : 0..4;
 time_out : boolean;
 activity : {send_request, stop_sending,
             receive_quote, error,
             evaluate, initialize};
```

Next, we assign initial values to the state variables if we do not want the model checker to assign a value nondeterministically:

```
ASSIGN
 init(stage) := notification;
 init(n) := 0;
 init(k) := 0;
 init(activity) := initialize;
```

The more complicated part of the program defines the value of a state variable in the next state depending on the value of other state variables evaluated in the current state. We use case statements to describe different possible conditions. This information can be extracted from the guarded transitions in the automata model. For each state variable, one case statement is needed. As SMV requires the transition relation to be *total*, the final (and default) case (marked with 1 standing for *true*) simply asserts that if none of the previous cases applies, the value of the state variable remains unchanged.[5]

```
next(stage):= case
 stage=notification & n<4
 : notification;
 stage=notification & n=4
 : hold;
 stage=hold & !time_out & k<n
 : hold;
 stage=hold & time_out & k=0
 : failure;
 stage=hold & (time_out & k>0 | k=n )
 : disposition;
 1 : stage;
esac;
```

[4]Given this limitation of SMV, we are therefore interested in alternative methods for termination checking or alternative representations of loops, see Section 6.

[5]We show the original SMV syntax with "!" standing for negation, "&" standing for conjunction, and "|" standing for disjunction. A case statement has the syntax *condition* : *new value*.

```
next(n):= case
 stage=notification &
 next(stage)=notification & n<4
 : n+1;
 1 : n;
esac;

next(k):= case
 stage=hold & next(stage)=hold & k<n
 : k+1;
 1 : k;
esac;

next(activity):= case
 stage=notification & next(stage)=notification
 : send_request;
 stage=notification & next(stage)=hold
 : stop_sending;
 stage=hold & next(stage)=hold
 : receive_quote;
 stage=hold & next(stage)=failure
 : error;
 stage=hold & next(stage)=disposition
 : evaluate;
 1 : activity;
esac;
```

Finally, the property we want to verify is added as a specification to the SMV program.

```
SPEC
AG(stage=notification
    -> AF(stage=disposition))
```

When feeding this SMV program into the NuSMV model checker, we obtain a counterexample, showing that the program does not always terminate in the desired goal states, but gets locked in a state satisfying *stage = failure*. The counterexample shows an execution path by listing the sequence of states forming this execution. In each state only those variables are shown whose value changes.

```
-- specification
AG(stage=notification
   -> AF(stage=disposition)) is false
-- as demonstrated by the following
   execution sequence

State 1.1:
stage = notification
n = 0
k = 0
time_out = 0
activity = initialize

State 1.2:
n = 1
activity = send_request

State 1.3:
n = 2

State 1.4:
n = 3

State 1.5:
n = 4

State 1.6:
stage = hold
time_out = 1
activity = stop_sending

-- loop starts here --
State 1.7:
stage = failure
time_out = 0
activity = error
```

Simply removing the failure stage does not eliminate the deadlock. In our example business model, no failure situation was mentioned. When removing from the IT model in Figure 2 the failure stage and the transition leading into it, NuSMV will still detect a deadlock in the *hold* stage, which cannot be left in case a timeout occurs *before* any quotes were received. This means, the model checker discovers the incomplete specification of the transition relation even in the business model we discussed above.

```
-- loop starts here --
State 1.6:
stage = hold
time_out = 1
activity = stop_sending
```

The counterexample provides interesting insight into the IT model. It tells an IT architect that if a timeout occurs immediately after reaching the *hold* stage, then the *failure* stage is entered and the process remains there. There is a loop indicating that the *failure* stage is entered infinitely often, which is triggered by the default transition in the case statement specifying the values of the *stage* variable. This means, none of the other specified transitions can fire given the path of execution shown in the counterexample and thus, the default transition is taken infinitely often.

Counterexamples may suggest refinements and corrections of the model in order to meet the required behavior. In the next section, we discuss possible changes to our model.

4 Refining the IT Model

In the next design phase, we enter a process of iterative changes of the IT model and subsequent verification of the termination property. Recalling our discussion from the beginning of the example, it was perhaps not a good idea to introduce an explicit *failure* situation into the model. The business model did not mention such a situation and maybe the business analyst would rather assume that the RFQ process is repeated in this case. Therefore, a reasonable correction of the IT model would be to redirect the transition back into the *notification* stage if $timeout \wedge k = 0$, i.e., to restart the entire RFQ process in the case of a failure (perhaps after changing the conditions in the RFQ, a detail not discussed further here). The corrected model is depicted in Figure 3.

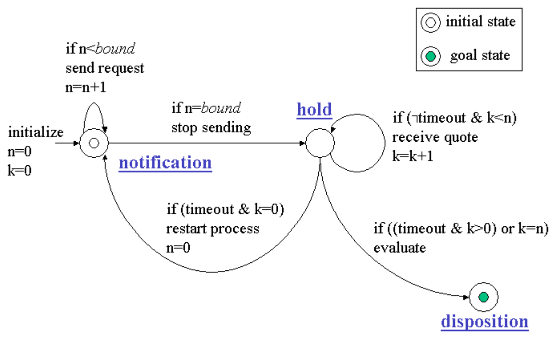

Figure 3. Corrected model with a restart loop.

Now, if a timeout occurs and no quotes have been received yet, a transition is fired, which restarts the notification process and resets the counter n back to zero. If we verify the termination of this model, we detect another loop—this time not caused by locking the process in a non-goal state, but by an infinite transition between two subsequent states.

```
-- specification
AG(stage=notification
   -> AF(stage=disposition)) is false
-- as demonstrated by the following
   execution sequence

State 1.1:
stage = notification
n = 0
k = 0
time_out = 0
activity = initialize

-- loop starts here --
State 1.2:
n = 1
activity = send_request

State 1.3:
n = 2

State 1.4:
n = 3

State 1.5:
n = 4

State 1.6:
stage = hold
time_out = 1
activity = stop_sending

State 1.7:
stage = notification
n = 0
time_out = 0
activity = restart_process

State 1.8:
n = 1
activity = send_request
```

The model checker provides a counterexample demonstrating that the termination property is false. In fact, one possible behavior of our model is a loop starting from a state satisfying *stage= notification*, arriving in *stage = hold*, going immediately back to *stage = notification* again, and continuing this loop indefinitely. The loop is caused by the fact that immediately after entering a state satisfying *stage=hold*, a timeout occurs, which fires the *restart process* activity. This means that in case of a broken communication pipeline with the vendors, this process could be restarted infinitely often. No quotes will arrive, but a timeout will always eventually occur. Such a situation is not impossible, although perhaps rare. Termination is not guaranteed in our refined model. If we assume that it is not infinitely often possible that no quotes arrive, but instead that a timeout occurs, we would expect that termination is guaranteed. This assumption can be added to the model as a *fairness constraint* making explicit our confidence in the communication pipeline. A fairness constraint expresses that, under certain conditions, an event or property will occur (or fail to occur) infinitely often. In our case, we want to add to the model our conviction that the timeout event will fail to occur infinitely often when entering the *hold* stage. Fairness constraints cannot be directly expressed in CTL, but require an extension of the language. However, many model checking tools allow one to *assert* certain fairness properties to hold in the model. This causes the model checker to consider only *fair* paths, i.e., paths satisfying the fairness constraint. It does not mean that the fairness constraint can be verified, which is a much trickier issue. The fairness constraint in our case reads:

$$\text{FAIRNESS } \neg(stage = hold \rightarrow timeout).$$

Under this fairness assumption, the model checker confirms that our automata will always terminate in the goal state.

The fairness constraint can also trigger a further refinement of the IT model, namely an introduction of a bound on the restart loop as shown in Figure 4. A new variable I is introduced into the model, which counts how many times the restart loop is iterated. If the loop is iterated max times and still no quotes have been received, the process terminates in a state satisfying *stage =failure*, which we reintroduced into the model. In all other cases, it will reach a goal state satisfying *stage = disposition*. We can formally verify the termination by reformulating the corresponding CTL formula and specifying *failure* and *disposition* as the two legal goal states of the process.

$$\mathsf{A\,G}\,\big((stage = notification) \\ \quad \rightarrow \mathsf{A\,F}\,(stage = disposition \;\vee\; stage = failure)\big).$$

This specification can now be verified even if the fairness constraint is dropped again from the model.

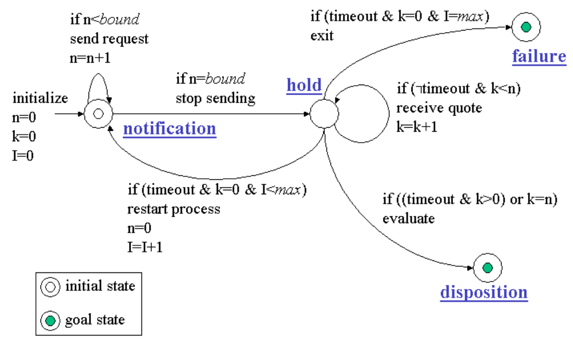

Figure 4. Final model with a bound on the restart loop.

The use of formal verification techniques allowed us to express the termination condition in our model and to verify it. In the case of a failing verification attempt, we were able to correct the model based on the information provided in the counterexample. The approach also helped us identify the assumptions according to which termination is guaranteed. Such assumptions often remain implicit in the model and lead to implementation failures quite frequently if a business process is changing and the IT model has to be adapted according to the change.

Finally, we want to show an alternative IT model that could be used to implement the RFQ process. The model we considered so far introduces two separate *notification* and *hold* stages in the process, i.e., first, all requests for quotation have to be sent out, then the quotes are received. In many situations, we would rather have a more flexible process design, in which the *send request* and *receive quote* activities can be interleaved with each other. This can be achieved either by merging the *notification* and *hold* stages or by allowing a nondeterministic transition between both stages during the RFQ process. We briefly discuss the second variant below.

Figure 5 shows a possible starting point for an IT model with interleaved sending and receiving activities. We note that two loops can occur between sending and receiving quotes. The first loop is caused by interrupting the *notification* stage in order to receive quotes and then using the *resume send* activity to return to the *notification* stage. The second loop is caused by the timeout event in the case that no quotes have been received and uses the activity *restart process* to repeat the RFQ process. Guaranteeing the termination of such a more complex, but also more realistic process model becomes an even more important task and can again be supported by the use of formal verification methods. In this example, true nondeterminism in the transitions can occur, which reflects that an agent can rather freely decide how to perform the RFQ process.

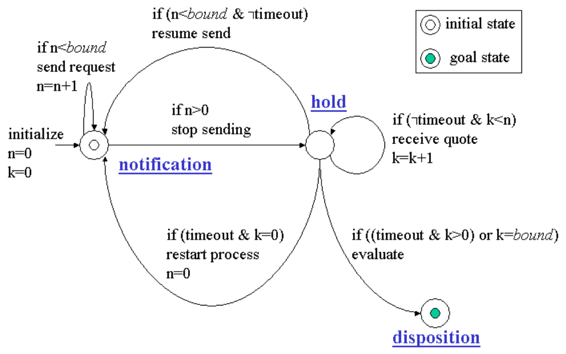

Figure 5. Alternative IT model with interleaved sending and receiving activities.

Termination of the process modeled in Figure 5 can be verified under the following fairness assumption:

$$\mathsf{FAIRNESS}\;\;\neg\big(stage = hold \rightarrow (timeout \;\vee\; k < bound)\big),$$

which expresses that it is not infinitely often true that if being in a state satisfying *stage = hold* a timeout occurs or not all quotes have been received.

5 Related Work

The formal specification of system and process models and the automatic verification of their properties is a long-standing area of research. Practical and widespread applications can be found in the area of reactive systems [11] and VLSI design [21]. Model checking has become a very popular method to verify the properties of finite-state concurrent systems [6, 1]. Only recently have these methods attracted increasing interest in the area of business process automation and integration. Various research approaches can be observed.

Several approaches try to make formal methods more amenable to business analysts. For example, [2] proposes a new interface language for SMV that combines an object-oriented representation with an action language. In [12], the language Promela is used to model business processes, and the process requirements are specified in the temporal logic LTL. SPIN, another model checking tool, is used to verify LTL specifications in Promela models. A library of elementary process patterns is provided, out of which business processes can be composed. The patterns are simpler than the one we discussed here and focus on the ordering of activities. For example, the authors distinguish between an activity that *can* follow another activity and an activity that *must* follow it. The activities themselves can be named and ordering constraints between them can be specified, but the conditions under which an activity executes cannot be made explicit. The verification focuses on reachability properties, i.e., by asking whether a certain state can never be reached in the process, SPIN will output counterexamples showing sequences of activities reaching this state if they exist. These sequences are then investigated further and may lead to a refinement of the model.

The need to equip languages such as UML with a non-ambiguous semantics before model checking techniques become applicable is discussed in [7], whereas [10] discusses the use of standardized ontologies as a semantic foundation of business process models.

The few proposals published so far seem to initiate a discussion of the suitability of the different modeling formalisms. So far, we have been able to identify an approach based on Petri nets [14] and one based on graph grammars [15]. An action-oriented representation of business processes based on the language ConGolog is presented in [16], and the applicability of the *Language Action* approach to e-commerce is discussed in [13].

In our own work, we have investigated the usability of automata with state variables, a formalism that appears to be especially attractive due to the availability of model checking tools. Given the limited experience with formal methods in business modeling, none of the approaches appears to be superior compared to the others. We agree with the authors of [14] that the activity-centered view of business process models and the state-based view adopted in model checking of concurrent systems pose a particular challenge, which needs to be further explored.

6 Summary and Outlook

We discussed a pattern-based approach to the mapping problem in business process automation and integration. Starting from a business process model given in some common modeling language, we emphasized the need to explicitly state the requirements of the process and its underlying structural pattern. For example, we showed how a process based on the structural pattern of *coupled, repetitive activities* is mapped into an IT model based on linked counting nondeterministic automata with state variables. Based on the example, we discussed how the automatic verification of process requirements—in our case the termination of the process in a designated set of goal states–can support an iterative approach of correcting and refining the IT model and making explicit assumptions that otherwise often remain hidden in the model.

We used symbolic model checking as a general approach to the verification of finite-state concurrent systems. Although model checking is a computationally hard problem, we argue that the complexity of business processes will *not* prevent the use of this technique. First, symbolic representation methods allow us to talk about entire sets of states instead of only individual states. Second, business process models appear to be bounded to a reasonable size; even if they do become large, they often inhibit structural information that permits a further decomposition of the process into subprocesses.

Current work is devoted to analyzing various structural patterns of business process models and their corresponding IT models. Typical examples are *branching* and *merging* of activities and the *interruption* and *interleaving* of activities. We are especially interested in verifying the termination of processes in designated end states. Our example based on two counters showed a clear limitation of the SMV model checking tool, which did not allow us to specify an arbitrary, but fixed bound on a state variable. Instead, the verification process had to be repeated for all possible bounds. We are currently developing alternative representations of loops that avoid this limitation.

References

[1] B. Berard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[2] B. Bloom. Seeing by owl-light: Symbolic model checking of business application requirements. submitted for publication, 2002.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. The web services description language WSDL. http://www-4.ibm.com/ software/ solutions/ webservices/ resources.html, 2001.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Proceed-*

ings Eleventh Conference on Computer-Aided Verification (CAV'99), number 1633 in LNCS, pages 495–499. Springer, 1999.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1996.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[7] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan, 2000.

[8] EbXML. ebXML specification. http://www.ebxml.org, 2001.

[9] H. Eriksson and M. Penker. *Business Modeling with UML: Business Patterns at Work*. Wiley Computer Publishing, 2000.

[10] J. Gordijn, H. Akkermans, and H. van Vliet. What's in an electronic business model? In *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, number 1937 in LNCS, pages 257–273. Springer, 2000.

[11] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw -Hill, 1998.

[12] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In *Theoretical and Practical Aspects of SPIN Model Checking*, number 1680 in LNCS, pages 92–107. Springer, 1999.

[13] P. Jayaweera, P. Johannesson, and P. Wohed. From business model to process pattern in e-commerce. In *Proceedings of the 6th International Workshop on the Language-Action Perspective on Communication Modelling*, pages 88–108, 2001.

[14] E. Kindler and T. Vesper. A temporal logic for events and states. In *Application and Theory of Petri Nets*, number 1420 in LNCS, pages 365–384. Springer, 1998.

[15] C. Klauck and H.-J. Mueller. Formal business process engineering based on graph grammars. *International Journal on Production Economics*, 50:129–140, 1997.

[16] M. Koubarakis and D. Plexousakis. A formal model for business process modeling and design. In *Conference on Advanced Information Systems Engineering*, pages 142–156, 2000.

[17] F. Leymann. WSFL: The web services flow language. http://www-4.ibm.com/ software/ solutions/ webservices/ pdf/ WSFL.pdf, 2000.

[18] F. Leymann and D. Roller. *Production Workflow*. Prentice Hall, 2000.

[19] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] RosettaNet. Rosettanet - a lingua franca for e-business. http://www.rosettanet.org, 2001.

[21] L. Stok et al. BooleDozer: Logic synthesis for ASICS. *IBM Journal of Research and Development*, 40(4):407–429, 1996.

[22] S. Tatte. XLANG: Web services for business process design. http://www.gotodotnet.com, 2000.