Verifying Business Processes using SPIN

Wil Janssen[1] Radu Mateescu[2] Sjouke Mauw[2,3]
Jan Springintveld[2,4]

[1]Telematics Institute,
P.O. Box 589, NL–7500 AN Enschede, The Netherlands

[2]CWI, Department of Software Engineering,
P.O. Box 94079, NL–1090 GB Amsterdam, The Netherlands

[3]Eindhoven University of Technology, Department of Mathematics and Computing Science,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands

W.Janssen@telin.nl, mateescu@cwi.nl, sjouke@win.tue.nl, spring@cwi.nl

Abstract

We present an application of the Spin model-checker in the business process reengineering framework of Testbed. Business processes are described by end-users in a graphical language with a causality-based semantics, called Amber. The Amber language contains various constructs describing actions, causality relations, disabling, and hierarchical composition into blocks (possibly) with interaction. Data entities are modeled as variables that are handled by the business processes. We present a validation methodology for business processes using model-checking techniques. In this approach, an Amber specification is automatically translated into a state machine description in Promela, which is the input language of the Spin model-checker. The correctness properties, concerning both the behavioural aspects and the data entities used in the specification, are checked on the resulting Promela program using Spin. A prototype verification toolset has been developed and successfully applied to various examples inspired from industrial Amber specifications.

1 Introduction

When modeling processes one is faced with the following dilemma: high level, informal models can be easily constructed and give insight, while rigourous, detailed models are needed if formal analysis is needed. Detailed, formal models, however, are often hard to understand and complex to construct.

One way of solving the dilemma is chosing for either of the two: use an informal model to give insight and to help structure thoughts, *or* invest in building a formal, complete and accurate model that can be analysed (automatically). A second way of solving the dilemma is to try to bring the informal and formal world more close to each other, for example by using a graphical formalism that allows for specifications that are easy to understand but are still supported by a formal semantics. Based on that semantics automated analysis can be developed. Statecharts and the Statemate environment [1] are a good example of such an approach for reactive systems specifications, based on structured automata. Other examples are SDL and MSC [2, 3, 4], often used in the telecommunications industry.

In the Testbed Project [5, 6] a systematic approach is developed to handle change of *business processes*. To support this approach, an environment is built to model and analyse business processes. The primary users of the environment are *business architects* which are not trained in formal modelling or computer science. Therefore, the modeling language should be highly intuitive. But, in order to allow for the analysis of complex models, the language should be supported by a

[4]Current affiliation is: Philips Research Laboratories Eindhoven, springtv@natlab.research.philips.com

formal syntax and semantics. Such properties are not often combined in languages for specifying business processes [7].

The Testbed language, called AMBER (for *Architectural Modeling Box for Enterprise Redesign*), is a graphical specification language. Both the behaviour in a business process as well as the performers of the process and the data used are modeled. Behaviour is specified as actions with their enabling relations. Also, behaviour can be structured in phases, and also using interactions, similar to synchronisation as in CSP or Lotos. Analysis is possible for both functional properties as well as quantitative (performance) properties. Quantitative analysis is discussed elsewhere [8].

This paper discusses how automated functional analysis in the Testbed setting can be realised. We do so by using the language Promela and the tool Spin [9, 10, 11] to perform the analysis. AMBER models are translated to Promela on the basis of an operational semantics. Properties are specified in Linear Time Temporal Logic for the moment, though work is under way to allow for more intuitive property specifications as well: LTL is not understandable for business architects.

We developed a translation from AMBER to Promela that covers almost the complete language. AMBER allows for the specification of infinite state systems, which cannot be verified by Spin. Such models had to be precluded. For finite state models different types of properties, such as precedence, consequence and exclusion, can be verified. The approach has been validated using non-trivial examples, which could be tackled easily. The limits of Spin have not yet been reached. It must be stated, however, that we have only used data in business models to a very limited extent.

The paper is organized as follows. Section 2 contains an informal presentation of the AMBER language used for specifying business processes. Section 3 describes the methodology used in the Testbed project for validating AMBER specifications. Section 4 illustrates the application of this methodology to an example inspired from a real-life AMBER specification. Finally, Section 5 gives some concluding remarks and directions for further work.

2 The AMBER language

In this section we give an informal explanation of the syntax and semantics of an AMBER model. Currently, the AMBER language is being applied to real world case studies, using the AMBER tool set, which is called Testbed Studio. The AMBER language (both syntax and semantics) has not yet reached its final shape. Feedback from the practical case studies will influence the further development of the language. In this section we present the basic constructs of the AMBER language in its current shape.

2.1 Actions and causality

AMBER is a graphical language for the specification of business processes. Such a specification describes which actions are involved in the business process and the causal relation between these actions.

Graphically, an action is represented by a circle which contains the name of the action. The fact that there is a causal relation between two actions is expressed by means of an arrow connecting the actions.

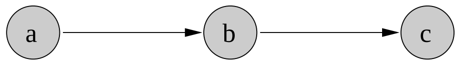


Figure 1: A simple AMBER model with three actions.

Figure 1 describes a process consisting of actions a, b and c. Action b may only be executed after action a and action c after action b. We can also speak of this causal relation in terms of enabledness. We say that execution of a enables action b. An action may have at most one incoming arrow and at most one outgoing arrow. An action without incoming arrow is enabled initially.

Please note the difference between such a causality relation and a sequential composition relation. The former specifies restrictions on the occurrence of actions, while the latter would specify the control flow of a system. The example in Figure 2 clarifies the difference.

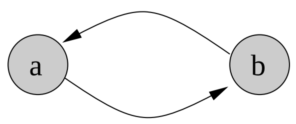


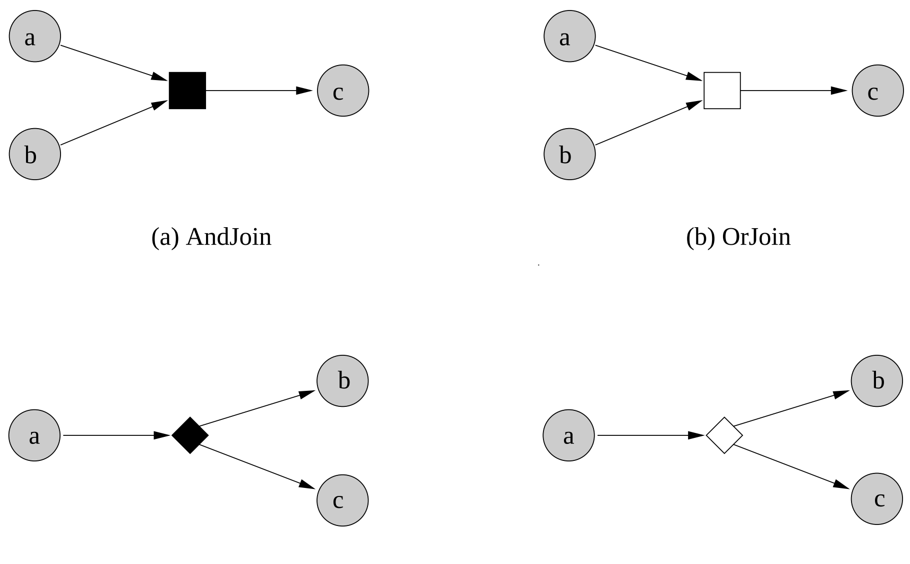
Figure 2: A deadlocking AMBER model.

In AMBER this cyclic graph does not describe a process that loops between actions a and b, as would be the case when interpreting the arrows as sequential connectors. It merely expresses that action b should be preceded by action a, and vice versa. Clearly, no process in which a or b occur can satisfy this cyclic dependency. This models a deadlocking situation.

Nevertheless, the AMBER language does have a way to express loops. We will discuss this feature later. Unless in such a loop, an action can only be executed once, regardless how often it is enabled. After execution, an action outside a loop can never be executed again.

2.2 Splits and Joins

Apart from expressing the causal relation between pairs of actions, AMBER also allows for more complex causal structures. Examples are shown in Figure 3. Figure 3(a) shows an AndJoin node, represented by a filled box. It means that in order for action c to become enabled, both a and b must be executed.



(a) AndJoin (b) OrJoin

(c) AndSplit (d) OrSplit

Figure 3: Four functional nodes: (a) AndJoin (b) OrJoin (c) AndSplit (d) OrSplit.

If the box is not filled, as in Figure 3(b), the node is called OrJoin. This means that execution of a or b (or both) suffices to enable c.

In the same way we can define an AndSplit node (denoted by a filled diamond shape) and an OrSplit node (denoted by an open diamond shape). The AndSplit in Figure 3(c) denotes that after execution of action a, both b and c are enabled. The OrSplit from Figure 3(d) implies that after execution of action a either b or c is enabled. This choice between b and c is made non-deterministically.

A Join node must have at least one incoming arrow and exactly one outgoing arrow. A Split node must have exactly one incoming arrow and at least one outgoing arrow.

We will call Join and Split nodes functional nodes. Using these functional nodes, complex causal relations can be defined, such as in Figure 4.

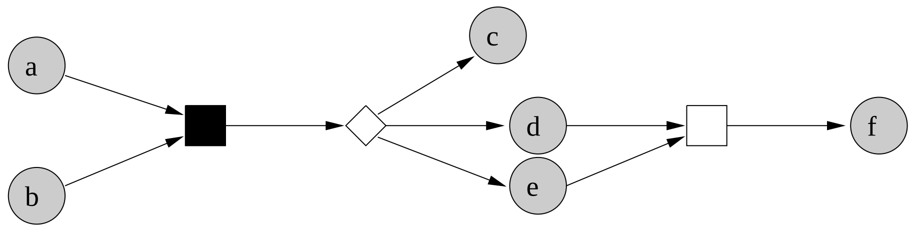


Figure 4: Composing functional nodes.

This example expresses that after execution of both a and b a choice is made between three continuations. If d or e is chosen, f is enabled, while after choosing c there are no enabled actions left.

Only actions have observable behaviour.

2.3 Loops

As explained before, we need additional machinery to express loops in AMBER. A typical example of such a loop is in Figure 5. An execution sequence described by this model starts with a, and is followed by an arbitrary (possibly infinite) number of repetitions of the sequence of two actions bc. Finally, if the loop is repeated a finite number, after the last b, action d is enabled.

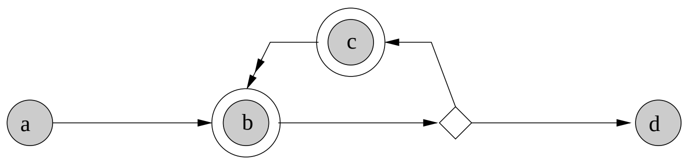


Figure 5: Expressing a loop.

A double arrow is used here for closing the loop, rather than a single arrow which would yield an inconsistent specification, as explained before. It is allowed that more than one double headed arrow enters an action node. The action nodes within the loop are outlined in order to express that they may be executed more than once. We will call this a new occurrence of the action.

Two more complex examples of the use of loops are given in Figure 6.

The first example shows two partly overlapping loops. There is no restriction on the coupling of loops. The second example shows that it is possible to specify a process with an infinite state space. The only difference with the standard loop example from Figure 5 is that the OrSplit is replaced by an AndSplit. The AndSplit makes that for each cycle through the loop, a new occurrence of both b and c is enabled. However, it is not necessary that this occurrence of c executes before the next cycle through the loop. Therefore, while the previous occurrence of c is still enabled, a new occurrence of c can be enabled. In this way, any finite number of c's may become enabled, before execution.

These examples show that in order to obtain easy to understand and finite state systems, some restrictions on the use of double headed arrows should apply. We will not discuss these restrictions in more detail.

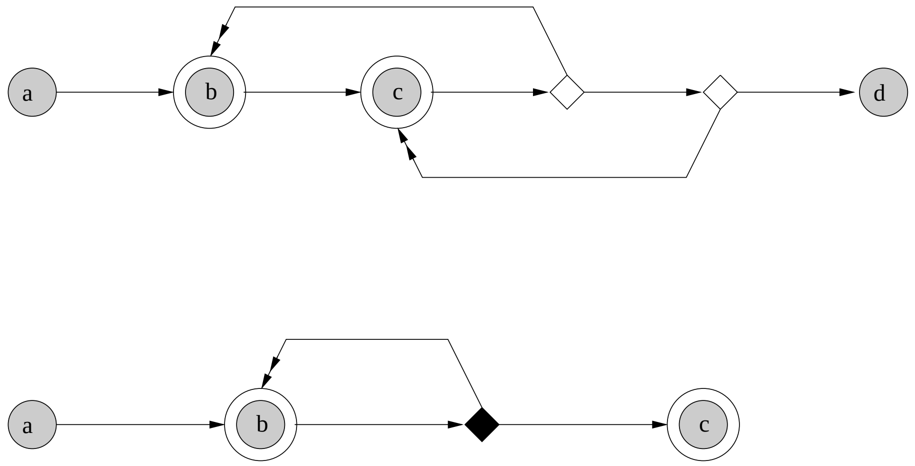


Figure 6: Two examples of loops.

2.4 Blocks

The Amber language has a modularization construct for structuring specifications. A number of actions can be grouped into a block, which is represented by a rectangle with rounded corners, as in Figure 7. This example describes three blocks, named x, y and z, which have several connections to each other.

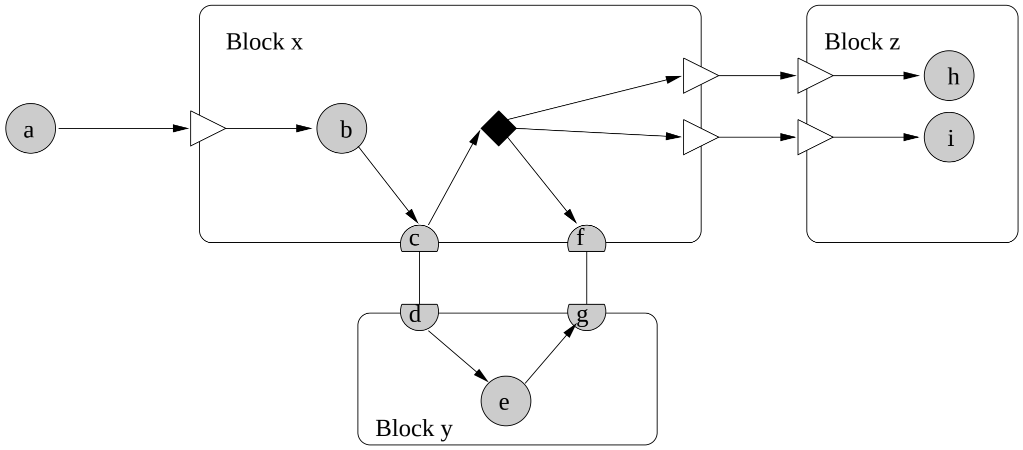


Figure 7: A specification with three blocks.

There are two ways to connect blocks to each other. First, there can be a causal relation from a node in one block to a node in another block. The arrow representing this causality is not allowed to simply cross the border of the blocks. It has to connect via the exit and entry points of the blocks. These exits and entries are represented by triangles. Entries and exits are syntactic elements that have no observable behaviour. The example in Figure 7 shows a causal relation between actions a and b through the entry of block x.

The second way of connecting blocks is by using interactions, which is a way to synchronize actions between blocks. An interaction is represented by a half circle, such as node c in Figure 7. Two or more interactions can be linked together by an interaction relation. There is, e.g., an interaction relation between interactions c and d. This means that c and d have to occur simultaneously. An interaction relation is enabled if all its interactions are enabled. In all other respects, interactions behave like normal actions. An interaction relation between more than two interactions is represented graphically by a forked line.

Blocks can be nested and can have any number of entries, exits and interactions.

2.5 Data

Finally, we mention that AMBER has a notion of data. At the moment, the actual data language has not yet been selected. For the sake of simplicity, we have imported the PROMELA language as our data language.

A simple example of the combined use of AMBER and PROMELA is given in Figure 8. It describes a loop which is executed exactly ten times.

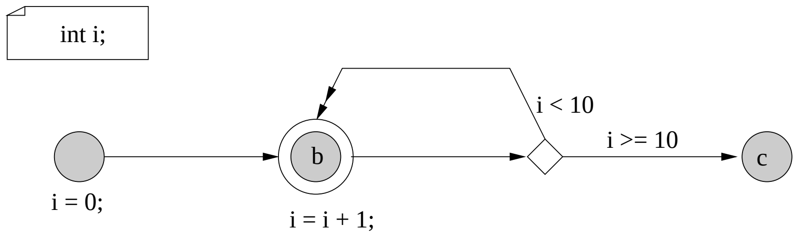


Figure 8: A loop which is executed ten times.

It shows that the AMBER language is extended at three places with data. First, we need a preamble for the declaration of data types and variables. In the example we only need the declaration of variable i. Second, we allow to associate program fragments to actions. The program fragment is assumed to be executed every time that its action is executed. In the example, an initialization statement is associated to action a and an increment statement to action b. Since it is in a loop, the increment statement will be repeated as often as action b executes. And thirdly, we allow to attribute arrows with conditions. After evaluation, such a condition determines whether the enabling arrow can be taken.

2.6 Other features

We have given an overview of most features of the AMBER language. Only two features have not been discussed yet, namely, disabling arrows and optional (inter)actions. An (inter)action is made optional by dashing its border. It simply means that it may, or may not, execute. Think of an activity like *pay invoice*: although a customer should pay the invoice, he can refrain from doing so. Such an activity is typically modeled as an optional action.

An enabling arrow is made into a disabling arrow by adding a slash through the arrow. If action a is connected to action b via a disabling arrow, it means that execution of a *prevents* the execution of b. We will not treat these features in more detail.

3 Verification using SPIN

In order to ensure the reliability of business processes described in the AMBER language, formal verification methods are needed. A part of the Testbed project is concerned with the functional analysis of business processes by means of model-checking techniques. In this section we give a detailed presentation of the approach used in Testbed for functional analysis of AMBER specifications, which is based upon the SPIN model-checker and its input language PROMELA.

3.1 Methodology

The approach adopted in the Testbed project for validating AMBER specifications is illustrated in Figure 9. Objects are denoted by oval shapes and transformations by rectangles. Two functionalities are offered: simulation of an AMBER specification (left track in the figure) and verification of temporal properties of an AMBER specification (right track in the figure).

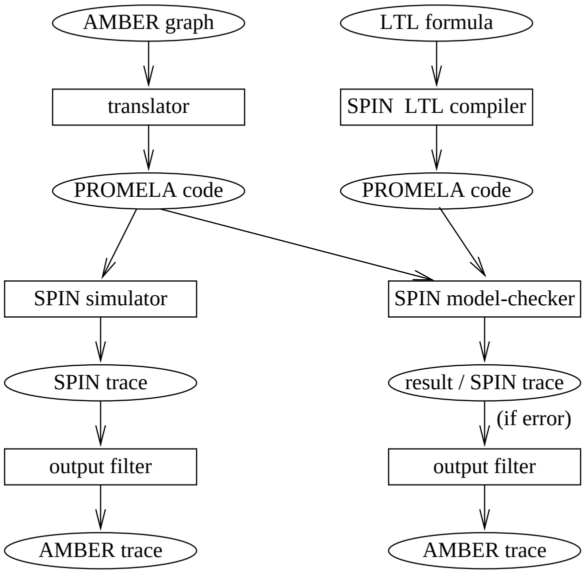


Figure 9: Overview of the validation methodology.

The AMBER specification to be analysed is automatically translated into a PROMELA program modeling the execution of actions in the AMBER model. We will give more details about this translation in Section 3.2. The PROMELA model obtained may serve as input for both simulation and verification using the SPIN tools.

To perform simulation of the AMBER specification, the SPIN simulator is used on the corresponding PROMELA model to generate random execution traces of actions contained in the AMBER model. An additional filter is applied to these traces in order to remove irrelevant information and to add some AMBER specific details.

To perform verification, the temporal properties of the AMBER specification are described as formulas in LTL (Linear Temporal Logic). These properties may concern both the execution of actions and the values of data variables in the AMBER model. We will precise in Section 3.3 the way in which these properties are specified. The temporal logic formulas are translated in PROMELA using the SPIN LTL compiler and are subsequently verified on the PROMELA model of the AMBER specification using the SPIN model-checker. Besides indicating whether a formula is satisfied or not by the model, SPIN also generates diagnostic traces whenever a formula does not hold. Using an additional filter, these traces are translated into a form that is easier to relate to the original AMBER specification.

3.2 Translation from AMBER to PROMELA

In order to translate AMBER specifications into PROMELA models, we must interpret the AMBER language using an interleaving semantics rather than a causality-based semantics. This operational interleaving semantics is defined as a *state automaton*. In this state automaton all nodes in the AMBER model correspond to transitions: actions and (combined) interactions, as well as functional nodes define state transitions. Thus the executions of individual actions, functional nodes, and interaction relations in an AMBER specification $\mathcal{A}$ are atomic events.

States are determined by the enabledness of nodes and the values of the data variables. The *initial state* is characterized by the set of events enabled initially (i.e., the actions having no incoming arrows) and by the initial values of the data variables of $\mathcal{A}$.

The *transition relation* of $\mathcal{A}$ describes in which way $\mathcal{A}$ evolves from one state to another. A transition between two states s_1 and s_2 is performed by executing a single event a, chosen non-deterministically from the set of events enabled in s_1. Also the values of the data variables associated with a are updated. Each transition is labelled with the event a that has determined it. An event may correspond to the execution of several nodes in the graph, in the case of an interaction relation.

Such a state automaton can be easily translated to Promela. Other translations than via an automaton have been tried, but resulted in extermely poor performance of Spin.

In the following, we sketch the translation and we illustrate it by means of an example. It is worth noticing that, since the efficiency was our primary concern, this translation does not strive for the most elegant Promela modelisation, but rather for reducing the state space as much as possible.

Let $\mathcal{A}$ be a correct Amber model. The state of the automaton is the cartesian product of the data variables and a set of variables per node i:

- A boolean state variable `triggered_`i, which is `true` if a predecessor of i executes and "triggers" i via an enabling arrow.
- For every AndJoin i there is a counter variable `in_counter_`i, which keeps track of the number of incoming enabling arrows of i that have been traversed. The node i can execute when it is triggered and all of its enabling predecessors have been executed, i.e., the value of `in_counter_`i equals the number of enabling arrows leading to i.

In the presence of disabling arrows, additional boolean variables and counters are used, in a similar way, to encode the states of the nodes having incoming disabling arrows. We do not treat these encodings in more detail here. Also, OrSplit nodes that are not on a loop require special attention to ensure they are executed only once.

The Promela model of $\mathcal{A}$ consists roughly of two parts:

1. A *preamble* containing declarations and initializations of the data variables contained in $\mathcal{A}$ and of additional variables representing the states of $\mathcal{A}$.
2. A *process* modeling the transition relation of the automaton. The process consists of a single, non-terminating `do-od` loop, the body of which is a large case distinction having one branch for each event (i.e., node or interaction relation) in $\mathcal{A}$. Each branch is guarded by a boolean expression encoding the enabledness condition of the corresponding event. If more than one branch is enabled, Spin chooses one non-deterministically. If there is no branch enabled at all, Spin will detect this by means of a timeout, causing the termination of the process. The condition is followed by the state update, corresponding to the transition taken.

In case of an OrSplit, a branch following the split is chosen non-deterministically. For each interaction relation in $\mathcal{A}$, the corresponding branch is obtained by the combination of all Promela fragments for the individual interactions in that interaction relation.

To illustrate the translation described above, we give below (a simplified version of) the Promela code generated by the Testbed Studio translator from the Amber model shown in Figure 8 (in which the OrSplit node has been noted s).

```
int i;                                        /* data variables       */

bool triggered_a = true;                      /* boolean flags for    */
bool triggered_b = false;                     /* nodes in the graph   */
bool triggered_s = false;
bool triggered_c = false;

active proctype AMBER_simulator ()            /* main process         */
{
  do                                          /* simulation loop      */
```

```
11      :: triggered_a ->
12           printf("Execute Action a\n");       /* trace message        */
13           triggered_a = false;                /* effect on source     */
14           i = 0;                              /* effect on data       */
15           triggered_b = true;                 /* effect on successors */
16      :: triggered_b ->
17           printf("Execute Action b\n");       /* trace message        */
18           triggered_b = false;                /* effect on source     */
19           i = i + 1;                          /* effect on data       */
20           triggered_s = true                  /* effect on successors */
21      :: triggered_s ->
22           printf("Execute OrSplit s\n");      /* trace message        */
23           triggered_s = false;                /* effect on source     */
24           if                                  /* effect on successors */
26             :: i < 10  -> triggered_b = true;
25             :: i >= 10 -> triggered_c = true
27           fi
28      :: triggered_c ->
29           printf("Execute Action c\n");       /* trace message        */
30           triggered_c = false;                /* effect on source     */
31           printf ("Value of i: %d\n", i);     /* effect on data       */
32    od
33  }
```

The preamble contains the declaration of the counter variable i (line 1) and the definitions of the state variables associated to the four nodes in the graph (lines 3–6). The process contains a `do-od` loop (lines 10–32) with four branches, one for each node in the graph. The code for the action nodes a, b, and c contains the effects on source, on data, and on successors previously described in the translation (except for node c, which has no successors). The code for the OrSplit node s has a non-deterministic effect on successors, triggering only one node between b and c.

It is worth noticing that the Promela code shown above can be further optimized, e.g., by encapsulating every deterministic sequence of instructions in a `d_step` construct, which allows to reduce the state space explored by Spin. Several optimizations of this kind are actually carried out by the Testbed Studio translator, but for the sake of clarity we did not describe them here.

For simulation purposes, each branch in the `do-od` loop contains also an additional `printf` statement witnessing the execution of the corresponding node. Applied to the Promela program above, the Spin simulator produces the following execution sequence (obtained after filtering):

```
Execute Action a  /* start         */
Execute Action b  /* 1st iteration  */
Execute OrSplit s
...
Execute Action b  /* 10th iteration */
Execute OrSplit s
Execute Action c  /* stop           */
Value of i: 10
```

This indicates that, after executing action a, the loop is traversed exactly ten times before executing action c. The execution of c produces no real effect on data, but only outputs the current value of the i variable.

3.3 Specification of temporal properties

The temporal properties of Amber specifications are expressed using Ltl [12], which is the property specification formalism accepted as input by Spin. Detailed descriptions of Ltl can be found in [12] or [9]. For the sake of completeness, we give here only a brief outline of the logic, mainly insisting on the way in which Ltl formulas are related to Amber specifications.

LTL formulas, noted `f`, are built from atomic proposition symbols `p` (denoting boolean predicates) and the constants `true` and `false`, combined using boolean connectives and/or temporal operators. Boolean conjunction, disjunction, negation, implication, and equivalence are denoted by `f1 && f2`, `f1 || f2`, `!f`, `f1 -> f2`, and `f1 <-> f2`, respectively. The formulas `<> f` and `[] f` denote the *eventually* and *always* temporal operators, meaning that `f` will be satisfied by some state (all states) in the future. The formula `f1 U f2` denotes the *strong until* temporal operator, stating that `f2` will certainly hold in the future, and `f1` will continuously hold until then.

The correctness requirements of an AMBER specification $\mathcal{A}$ may combine two kinds of temporal properties:

behavioural properties, concerning the execution of the actions contained in $\mathcal{A}$. This kind of properties are expressed by means of special boolean variables witnessing the execution of the nodes referred to in the temporal property. For each such node i, a boolean variable `executed_`*i* is automatically declared in the preamble of the PROMELA model generated from $\mathcal{A}$ (see Section 3.2). The variable is initialized to `false` (in the preamble) and set to `true` (as an additional effect of i) when the node i is executed. These variables can be used as atomic propositions in the LTL formulas.

data-based properties, concerning the evolution of the data variables defined in $\mathcal{A}$. This kind of properties are expressed by means of atomic propositions defined in the preamble of the PROMELA model and denoting predicates over data variables. For each atomic proposition `p` referred to in the LTL formula, there must be a definition `#define p` *exp*, where *exp* denotes a PROMELA expression of type `bool`. These definitions must be provided by the user together with the temporal formula; they will be expanded when the PROMELA model of $\mathcal{A}$ and the LTL formula are processed by SPIN.

We illustrate the expression of both behavioural and data-based temporal properties on the AMBER example shown in Figure 8, for which the corresponding PROMELA model has been given in Section 3.2.

A simple liveness property of the system is that the action c will be eventually executed (i.e., the loop will be eventually exited). This can be expressed by the LTL formula below:

```
<> executed_c
```

where the variable `executed_c` must be appropriately defined and updated, by adding the line

```
bool executed_c = false;
```

in the preamble of the PROMELA model (lines 3–6), and the line

```
executed_c = true;
```

in the simulation loop branch corresponding to c (lines 28–31). The developed tools automatically take care of this.

A simple safety property of the system is that the variable i will never exceed the value 10. This can be expressed by the following LTL formula:

```
[] p
```

where the atomic proposition `p` must be appropriately defined, by adding the line

```
#define p (i <= 10)
```

in the preamble of the PROMELA model.

Using the SPIN model-checker, we can verify that both properties are satisfied by the model.

4 Application

In order to illustrate the methodology presented in the previous section, we present here the verification of a more involved example of business processes described in AMBER. We first give the AMBER specification, next we express the desired correctness properties, and finally we show the verification results obtained.

4.1 The AMBER specification

We consider the AMBER specification illustrated in Figure 10. This AMBER model describes the interaction between a process GARAGE, modelling the repairing of a car after an accident has occurred, and a process PRO-FIT, handling the evaluation of the claim issued by a customer.

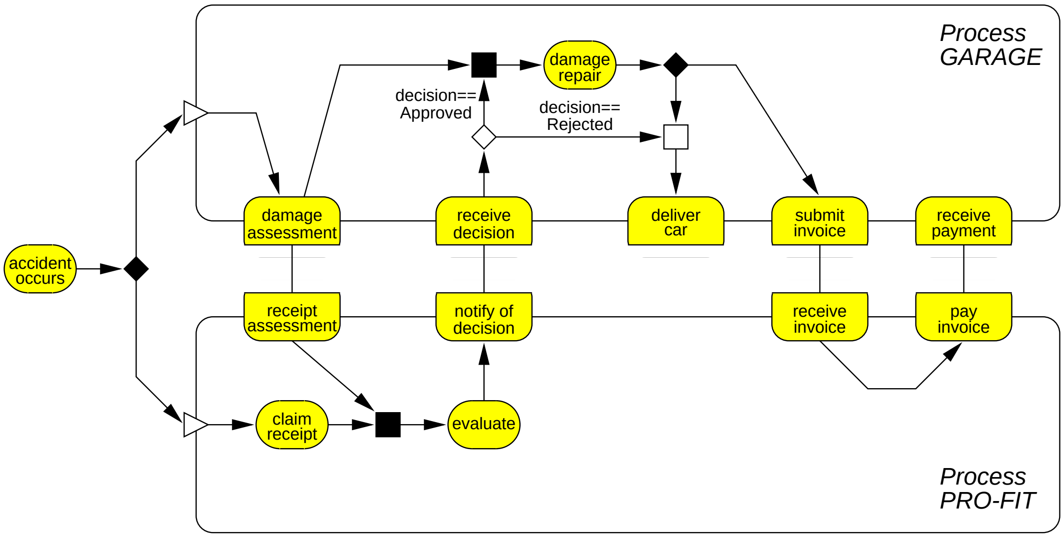


Figure 10: An example of AMBER specification.

The AMBER specification also contains several data entities, which are modelled in PROMELA using the following types:

```
mtype = {
   Whole, Broken, Repaired,          /* car status          */
   None, Approved, Rejected,         /* decision status     */
   Idle, UnderConsideration, Informed /* customer status     */
};

typedef Customer {                   /* customer attributes */
   byte customerGroup;
   mtype status;
   bool claimReceived;
}
```

where `byte` and `bool` are predefined PROMELA types. The states of the different data entities are encoded as values of an enumerated type, defined using the `mtype` PROMELA construct. Each customer has associated three informations: his group (identified by a number), his status (which may be `Idle`, `UnderConsideration` or `Informed`), and a boolean value indicating whether his claim has been received or not by the process PRO-FIT. The data entities used in the specification are given below.

- A file recording information about customers, modelled by an array variable `customerFile`:

  ```
  Customer customerFile[5];
  ```

- A customer, modelled by a variable `customerId`:

  ```
  byte customerId;
  ```

- A car, modelled by a variable `car`:

```
mtype car = Whole;
```

- A decision taken by the process PRO-FIT, modelled by a variable `decision`:

```
mtype decision = None;
```

- A damage assessed by the process GARAGE, modelled by a variable `damage`:

```
int damage;
```

The data variables above are modified by the execution of certain nodes in the AMBER model given in Figure 10. For each of these nodes, the effects performed on data are modelled by the fragments of PROMELA code indicated below.

- Accident occurs:

```
customerFile[0].customerGroup = 1;     /* customer file initialisation */
customerFile[0].status = Idle;
customerFile[0].claimReceived = false;
...
customerFile[4].customerGroup = 5;
customerFile[4].status = Idle;
customerFile[4].claimReceived = false;

car = Broken;                          /* car status                   */

if                                     /* customer selection           */
   :: true -> customerId = 0;
   ...
   :: true -> customerId = 4;
fi;
```

- Damage assessment:

```
if
   :: true -> damage = 1000;
   ...
   :: true -> damage = 5000;
   :: true -> damage = 6000;
fi;
```

- Receipt assessment:

```
customerFile[customerId].status = UnderConsideration;
```

- Claim receipt:

```
customerFile[customerId].claimReceived = true;
```

- Evaluate:

```
if
   :: damage <  customerFile[customerId].customerGroup * 1000 ->
        decision = Approved;
   :: damage >= customerFile[customerId].customerGroup * 1000 ->
        decision = Rejected;
fi;
```

- Notify of decision:

```
customerFile[customerId].status = Informed;
```

- Damage repair:

```
car = Repaired;
```

The assignment of random values to `customerId` and `damage` is modelled by means of the non-deterministic PROMELA construct `if-fi`. A claim is approved (and subsequently treated by the process GARAGE) if the value of the damage is below a certain threshold, determined by the customer group.

4.2 The correctness properties

We give here the correctness properties of the AMBER specification described in Section 4.1. To express more conveniently certain safety properties, besides the LTL operators mentioned in Section 3.3, we will also use the *weak until* operator `W`, defined as `f1 W f2` = `[] f1 || (f1 U f2)`, which states that `f1` continuously holds until `f2` is satisfied (which may never happen, in this case `f1` being always satisfied).

To express behavioural properties, we will use in the LTL formulas several propositions denoting the execution of nodes in the AMBER model. These are modelled by means of special boolean variables `executed`i, associated to each node i used in the formulas (see Section 3.3). For clarity, we will use for these variables the names of their corresponding nodes: the propositions `damage_repair`, `deliver_car`, and `submit_invoice` will denote the execution of the nodes labelled by "damage repair," "deliver car," and "submit invoice," respectively.

To express data-based properties, we will use in the LTL formulas several propositions over the data variables of the AMBER specification. These propositions are defined as follows:

```
#define claim_approved   (decision == Approved)
#define claim_below_6000 (damage <= 6000)
#define customer_4       (customerId == 4)
#define claim_rejected   (decision == Rejected)
```

Some typical temporal properties are described below.

Property 1. *Is the car repaired only when the claim is approved?*

This question can be answered by checking the following LTL formula:

```
[] (!damage_repair W claim_approved)
```

specifying that the car cannot be repaired unless the claim is approved.

Property 2. *Will every claim below 6000 be approved for customer 4?*

This reduces to the verification of the LTL formula below:

```
[] ((claim_below_6000 && customer_4) -> <> claim_approved)
```

stating that every claim less than 6000 issued for customer 4 will be eventually approved.

Property 3. *Is the car always repaired when delivered?*

To answer this, we check the following LTL formula:

```
[] (!deliver_car W damage_repair)
```

expressing that it is impossible to reach a car delivery before performing a damage repair.

Property 4. *Can the car be repaired if the claim is rejected?*

We answer this question by checking the LTL formula below:

```
[] (claim_rejected -> [] !damage_repair)
```

specifying that after a claim has been rejected, the car will never be repaired.

Property 5. *Can the garage submit an invoice even if the claim is rejected?*

This can be translated into the following LTL formula:

```
[] (claim_rejected -> [] !submit_invoice)
```

stating that every time a claim is rejected, the garage will never submit an invoice.

4.3 The verification results

The five correctness properties given in Section 4.2 have been verified using the Testbed Studio toolset on the AMBER specification described in Section 4.1.

The PROMELA model generated by the Testbed translator has about 300 lines. The verification results of the temporal properties on this model are summarized in Table 1. For each property, the table gives its result on the model, the number of states explored by SPIN, and the memory and time required for the verification. All experiments have been performed on a Silicon Graphics workstation with 64 Mbytes of memory.

Property	Result	Nb. states	Memory (Mb)	Time (sec.)
1	true	1,797	2.542	35
2	false	969	2.542	17
3	false	37	2.542	4
4	true	1,240	2.542	23
5	true	1,240	2.542	23

Table 1: Verification results.

Properties 1, 4, and 5 are true on the model. This confirms the intended behaviour of the AMBER specification, meaning that the answers on the corresponding questions are “yes,” “no,” and “no”, respectively. Properties 2 and 3 are false, but this also confirms the intended semantics of the AMBER model, meaning that the answers to the corresponding questions are “no.” For property 2, SPIN produces an error trace leading to the rejection of a claim with value 5000 for customer 4 (this is indeed the behaviour imposed by the claim evaluation). For property 3, the error scenario leads to a car delivery without any damage repair (this happens indeed when the claim is rejected).

5 Conclusion and future work

The main conclusion that can be drawn from this work is that it is feasible to use model checking techniques for analyzing AMBER specifications.

We have succeeded in providing a translation from AMBER to PROMELA which preserves the intended semantics for the subset of finite state AMBER specifications. The restriction to finite state specifications is motivated by the fact that techniques for model checking of infinite state systems are still in the stage of academic research.

By means of examples we have shown how properties of AMBER specifications expressed in natural language can be translated into LTL, the property language of SPIN.

Tools have been developed, which prove feasibility of our methodology. These tools are currently being integrated in the AMBER toolset, called Testbed Studio. Examples show that for the model checking of industrially relevant AMBER specifications time and memory requirements stay within reasonable limits. We expect that in practice the size of the data attributes and items used in the AMBER specification will be most critical.

From doing this work we have learnt that it is hardly impossible to develop a language and its tools without having a formal semantics. Ideally, syntax, tools and semantics should be developed in parallel because they influence each other.

Nevertheless, in the course of the work, we have provided a formal semantics for AMBER. This semantics lines up very neatly with the translation to PROMELA. Drawback of this approach is that it is a very operational semantics, not in the least fully abstract.

Our experience with the SPIN tools was positive, but some comments apply.

- We had to restrict ourselves to a sublanguage of PROMELA which is very basic. Our first approach to modelling AMBER in PROMELA made heavy use of the PROMELA process model. It appeared to be much more efficient to encode all behaviour into one single PROMELA process.
- The use of non-trivial data structures in PROMELA to encode the state automaton, such as arrays and structs resulted in significantly higher memory requirements.
- In some cases properties can be expressed more easily in a branching time logic than in the linear time logic provided by SPIN.
- SPIN only supports weak fairness, whereas in AMBER often strong fairness is assumed.

Although this work has showed feasibility of the approach, there are several points of interest that require attention in the near future.

First of all, the AMBER language has not yet reached its final shape. In particular, a data language still has to be selected and integrated with AMBER. A translation from this data language to PROMELA has to be provided.

An interesting question that remains is how to specify verification properties. Such properties should be specified by (skilled) business architects. For them (temporal) logic is too difficult and too abstract. Currently, we are investigating two different directions. The first direction is to use simple, unstructured AMBER models as property specifications. This means that such specification have to be interpreted slightly differently: an action a preceding an action b can be interpreted as *if a occurs, then b should follow it,* or *if b occurs, it should be preceded by a*, or even as *a and b occur both, and a precedes b.*

A second way of making property specification accessible is to define a set of parameterised patterns that occur frequently when verifying business properties. Such properties can be presented in a "drag and drop" style to the user.

For both directions one has to choose between a translation to PROMELA never claims via LTL, or directly.

Closely related to this is the question of what fairness assumptions should be considered and how they could be expressed in LTL.

The definition of sufficient conditions to ensure that a given AMBER graph describes a finite state system is also of practical interest.

Finally, the interaction between the model checker and the simulation tool under development should be made clear in such a way that the error scenarios produced by SPIN can be interactively replayed in the Testbed simulation tool.

Acknowledgements

This paper results from the Testbed project, a 120 man-year research initiative that focuses on a virtual test environment for business processes. The project develops knowledge, methods and

(software) tools to support business process (re-)design, in particular in the financial sector where project results have been applied in numerous real-life business cases.
The project builds on an architectural modelling technique developed, and further scientifically supported by the Architecture Group of the University of Twente (Prof. Chris A. Vissers). The Testbed consortium consists of ABP, the Dutch Tax Department, ING Group, IBM and the Telematics Institute (The Netherlands) and co-operates with several Dutch universities and research institutes. The project is financially supported by the Dutch Ministry of Economic Affairs. We appreciate to acknowledge all Testbed contributors.

We would also like to acknowledge several individual people for their support during different phases of the project. We thank Rob Gerth for his help in initializing the project and for his support with the SPIN tools. Thanks are also due to Petra van der Stappen and Peter Fennema for implementing the model-checking features in the Testbed Studio toolset.

References

[1] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and A. Shtull-Trauring, STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16 (1990), 4, p. 403-414.

[2] K. Turner, editor. *Using Formal Description Techniques.* John Wiley & Sons, 1993.

[3] Braek, R., SDL Basics. *Computer Networks and ISDN Systems*, 28 (1996), p. 1585-1602.

[4] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, 1997.

[5] H.M. Franken. A virtual test environment for business processes, *ACM Bulletin of Special interest group on supporting group work*, April, Vol. 18, N. 1, pp. 63-67, 1997.

[6] H. Franken and W. Janssen. Get a grip on changing business processes - results from the Testbed project. *Knowledge and Process Management*, Volume 5, Number 3, 1998 (forthcoming). John Wiley & Sons Ltd.

[7] W. Janssen, H. Jonkers and J. Verhoosel. What makes business processes special? An evaluation framework for modelling languages and tools in Business Process Redesign, in Siau, Wand and Parsons (eds.), *Proc. 2nd CAiSE/IFIP 8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Barcelona, Spain, 1997. (Available as http://www.telin.nl/publicaties/1997/caise97_final.doc)

[8] H. Jonkers, W. Janssen, A. Verschut en E. Wierstra, A unified framework for design and performance analysis of distributed systems, to appear in *Proc. 1998 IEEE Int. Computer Performance and Dependability Symposium* (IPDS'98), Durham, NC, USA, Sept. 1998.

[9] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[10] G.J. Holzmann. Design and Validation of Protocols: A Tutorial. *Computer Networks and ISDN Systems*, 25(9):981-1017, 1993.

[11] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.

[12] A. Pnueli. The Temporal Logic of Programs. *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science*, Providence, R.I., pp. 46-57, 1977.