Model Checking for Managers

Wil Janssen[1], Radu Mateescu[2], Sjouke Mauw[3,4], Peter Fennema[1] and Petra van der Stappen[1]

[1]Telematica Instituut, P.O. Box 589, NL-7500 AN Enschede, the Netherlands
{Janssen,Stappen,Fennema}@telin.nl
[2]INRIA Rhone-Alpes, Montbonnot Saint-Martin, France
Radu.Mateescu@inrialpes.fr
[3]Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, NL-5600 MB Eindhoven, the Netherlands
sjouke@win.tue.nl
[4]CWI, P.O. Box 94079, NL-1090 GB Amsterdam, the Netherlands

Abstract. Model checking is traditionally applied to computer system design. It has proven to be a valuable technique. However, it requires detailed specifications of systems and requirements, and is therefore not very accessible. In this paper we show how model checking can be applied in the context of business modeling and analysis by people that are not trained in formal techniques. Spin is used as the model checker underlying a graphical modeling language, and requirements are specified using business requirements patterns, which are translated to LTL. We illustrate our approach using a business model of an insurance company.

1 Introduction

In the last few years model checking has proven to be a valuable tool in the development of correct systems. Applications range from software controlling storm surge barriers [Kars96], through space craft controllers [HaLP98] to integrated circuits. Tools like Spin [Hol97], SMV [SMV99] and CADP [CADP99,Gara98] have outgrown their infancy and are becoming professional tools that are applicable to real-life problems.

Model checking requires a number of steps. A correct abstraction from the problem must be defined in the input language of the model checker. Often this requires a translation from the problem domain to the concepts used in the model checker (such as message passing systems, process algebra or automata). This model must be validated in order to ensure that no mistakes are introduced by the abstraction. Thereafter, the correctness requirements must be formulated in the corresponding requirements language, such as never claims or temporal logic formulae. This again requires an abstraction from the informal requirements in the application domain.

Finally, the specification can be checked for satisfaction of the requirements. If the requirements are not satisfied, both the requirement specification as well as the system model must be checked: either the system does not satisfy the (informal) requirement, the requirement is not defined correctly or the model is an incorrect

abstraction of the system. To do so, the counter example must be translated back to the application domain.

All in all, this makes model checking complex and cumbersome: designing systems is not easy, developing specifications is a complex task and defining the right correctness requirements must be done carefully. Model checking is an activity for skilled computer scientists and engineers, isn't it?

Well, to a certain extent this is true. However, we argue that when given the appropriate tools and methods model checking can be made accessible to a large audience. Even for people that are not trained in formal techniques, model checking can be a valuable tool for developing correct systems. In the Testbed project [FrJa98] we have developed tools and methods for business process modeling and analysis, aiming at business analysts. Business analysts usually have a background in business administration and little or no knowledge of computer science. Testbed employs a graphical modeling language that closely corresponds to the concepts relevant to business modeling (activities, actors, co-operation, responsibilities, duration and so on). The tool Testbed Studio allows for easy modeling of business processes and provides a number of means of analysis, for both quantitative as well as functional properties (completion time, workloads, critical path, data flow, process type, multistep simulation and so on).

In [JMMS98] we showed that model checking can be applied in the context of Testbed and business modeling. On the basis of an operational semantics a translation from our business modeling language to Promela was defined. Model checking proved to help in validating and verifying business models. However, this was still performed by formalists outside the tool Testbed Studio. Model checking by managers requires a different approach.

Spin is "under the hood" of one of the analysis tools in Testbed Studio. Requirements are defined using a number of predefined patterns: traces of activities, combined occurrence, precedence and consequence. These requirements are translated to Linear Time Temporal Logic (LTL). The business process model is translated to Promela, using an operational semantics. The Promela model and LTL specification are checked using Spin and the outcome is visualized in the graphical environment of Testbed Studio.

Of course, such simplicity comes at a cost: using a fixed number of patterns restricts the expressivity to a large extent. Moreover, no data modeling language has been employed yet, allowing for full state space exploration, even for large models. Still, we carefully selected the patterns in the tool on the basis of a large number of practical applications. The coverage of questions that can be answered is substantial.

By complementing model checking with other means of analysis, especially for quantitative properties [JJVW98] the limitations of model checking in our set up, and the restriction to non-quantitative properties in general, are overcome.

The rest of this paper is organized as follows. We first discuss our graphical modeling language on the basis of an example. We then introduce the patterns used for functional analysis and their translation into LTL. These are illustrated using an

example. We conclude with a number of remarks on the implementation and our findings in using Spin in this context.

2 Functional analysis in Testbed

The Testbed project develops a systematic approach for business process engineering, particularly aimed at processes in the financial service sector [FrJa98]. A main objective is to give *insight* into the structure of business processes and the relations between them. This insight can be obtained by making *business process models* that clearly and precisely represent the *essence* of the business organisation. These models should encompass different levels of organisational detail, thus allowing the analyst to find bottlenecks and to assess the consequences of proposed changes for the customers and the organisation itself; see also, e.g. [JaEJ96].

Business modelling languages may be deployed for many different purposes. Not only do they supply a sound foundation for communicating and discussing business process designs, they may be used as well for, e.g.,

- *analysis* of business processes, that is, assessment of qualities and properties of business process designs, either in quantitative or qualitative terms;
- *export to implementation platforms*, such as workflow management and enterprise resource planning systems;
- *job design*, that is, designing detailed job specifications and generating job instructions.

Every specific purpose of a business modelling language brings about its own specific demands on the language. We first explain our business modeling language and illustrate its use with an example.

2.1 The Testbed modeling language AMBER

The core of the business modelling language contains concepts that enable basic reasoning about business processes. AMBER recognises three aspect domains:

- the *actor* domain, which allows for describing the resources deployed for carrying out business processes;
- the *behaviour* domain, which allows for describing what happens in a business process;
- the *item* domain, which allows for describing the items handled in business processes.

Here we restrict the discussion to the behaviour domain, as this is the relevant part for model checking. A detailed overview of the language can be found in [EJO+99].

The basic concept in the behaviour domain is the *action*. It models a unit of activity in business processes. An action can only happen when its *enabling condition* is satisfied. These conditions are formulated in terms of other actions having occurred yet, or not. Actions that are performed by more than one actor in co-operation are

called *interactions*. The contribution of an actor to an interaction is represented by a (stretched) semi-circle, e.g. *submit claim* and *receive claim* in figure 2.

Apart from the actions and their properties, causal or temporal relations between actions are important elements of a behaviour model. Figure 1 gives an overview of the main relations:

- The simplest situation is a single causality relation, which means that a certain action can only start when another (preceding) action has finished.
- We have two types of *splits*:
  1. an *or*-split, which means that after completion of the preceding action one of a number of possible actions is chosen.
  2. an *and*-split, which means that after completion of the preceding action several other actions can take place *in parallel*.
- Finally, we have two types of *joins*: an *or*-join (disjunction) indicates that *at least one* of the preceding actions must have been completed before an action can start, while an *and*-join (conjunction) indicates that *all* preceding actions must have been completed.

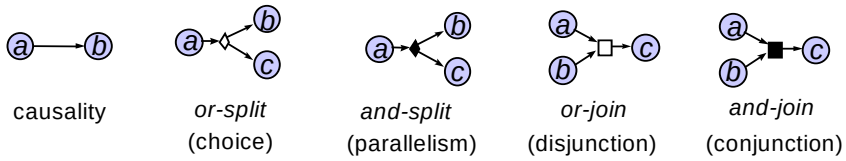

Fig. 1. (Inter-) action relations.

The use of these relations is illustrated in the behavior model below.

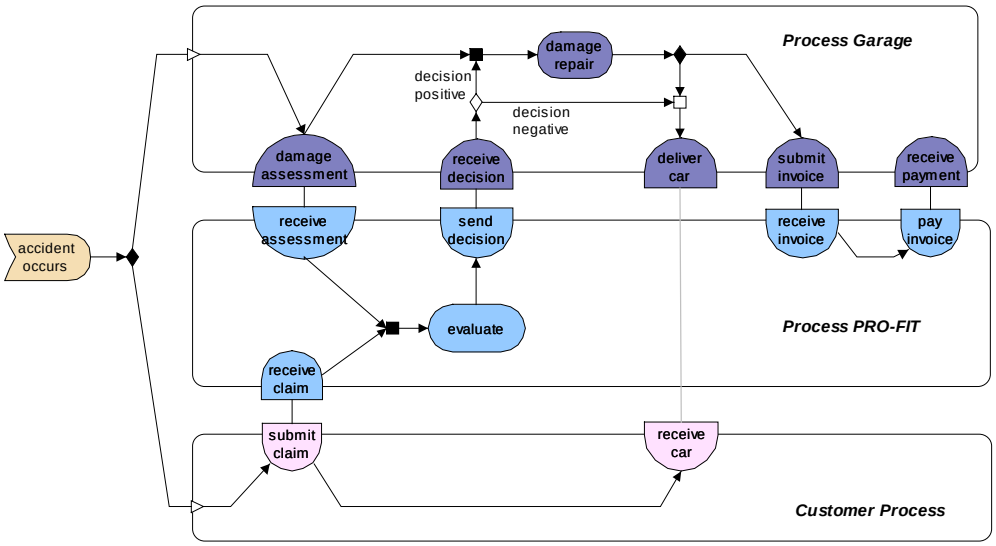

Fig. 2. The claim submission process

In figure 2 we model the claim submission process of an insurance company called *PRO-FIT*. After an accident occurs, the customer submits a claim to PRO-FIT. When the damage assessment has been submitted as well, the claim can be evaluated, leading either to a positive or a negative decision. In case of a positive decision the car is repaired and delivered, and the invoice is paid.

Data (the item domain) is not treated in this paper, as it is not yet taken into account in the functional analysis.

2.2 Analyzing AMBER models

Business models as described above function as a blueprint for the actual implementation. Procedures for people are derived from them and they may even function as a specification of the workflow implementations. Many business modeling environments have export possibilities to workflow. Therefore, the correctness of the business model becomes crucial to the company.

"Correctness" is a difficult property. It has quantitative elements (there are sufficient resources to do the work, completion time is according to the critical success factors of the organization and so on) as well as functional properties. Functional properties often concern the control flow in the process: every accepted claim should be accompanied by an assessment, no claim can ever be rejected as well as accepted, and so on. If models are small, simulation can give sufficient insight in the model to validate them. However, if processes –and thus their models- grow larger it becomes more difficult to check them. Model checking then becomes an interesting idea.

Unfortunately, the proper use of model checking techniques is not easy. It should be done at the abstraction level of the user: he or she should be able to define both the business models as well as the corresponding correctness requirements in the way he or she thinks. For Testbed this means models must be AMBER models, and requirements should be stated in (almost) natural language expressing occurrence of actions and their relations.

In [JMMS98] we showed how to translate AMBER models to Spin on the basis of a state machine description of the model. This resulted in highly compact Promela specifications of business models. The correctness requirements, however, were still formulated in LTL. Experiments, even with trained formalists, have shown that the specification of LTL queries was more a source of mistakes than the actual business models themselves. For practical applications in our context LTL is unacceptable. Therefore, we aimed at developing an easy to use and dedicated way to express requirements on AMBER models.

3 Enhancing usability with patterns

Hence, in this case we need an easy-to-use link to Spin, tailored to applying Spin to business models. We built such a link that offers the user a choice between a number

of selected query types. The user can instantiate a query by setting a number of parameters. We call these query generators *patterns*.

3.1 Identifying patterns

Our approach to identifying patterns is a pragmatic one, starting from a business perspective. We studied a large number of business cases, for which process models were drawn. For each case relevant questions that might be answered using model checking were listed. The questions were divided into categories, after the complicating factors contained in the question. Each question was judged on solvability by model checking.

When looking for patterns, we found four types of questions that occur often. These are relatively simple questions. More complicated questions often concern the relation between two of these questions. The four patterns are:

1. Sequences of activities, e.g. "the sequence of activities *submit claim*, *receive payment* can (or can never) occur";
2. Consequences of activities, e.g. "every submitted claim will lead to *receive payment* and *receive car*";
3. Combined occurrence or exclusion, e.g. "*evaluate* and *pay invoice* always occur together";
4. Required precedence of activities, e.g. "*receive car* requires that *repair car* has happened".

These patterns are illustrated using a larger example in section 5.

We present our four patterns in table form. The tables contain both text fields and variable fields. A question is derived from the pattern by filling in the variable fields. Variable fields for (sets of / lists of) actions may be filled with any action from the current model (represented in bold and italicized). Other variable fields offer a choice between some predefined values. This choice is indicated in the table by a column with one italicized value in each entry. The first value listed is the chosen default. The meaning of the patterns is explained and examples are given. Two choices that often appear are that between an, each, and all, and that between ever, never, and always.

The word "an" acts as existential quantifier: the query *an action leads to…* should be read as *Does an action exist that leads to …*. Less trivial is the difference between *each* and *all*. If we ask whether *each* action leads to *y*, we mean to ask whether each individual x from X leads to y. If we ask if *all* actions lead to *y*, we mean to ask whether all actions $x \in X$ together lead to y. Thus if all $x \in X$ are executed, will y be executed?

The word "ever" can be viewed as existential quantifier: "property p ever" should be interpreted as: there is a run for which the property p holds. "Never" refers to the opposite: there is no run in which the property holds. The word "always" stands for the phrase "in all possible runs". For Spin this means to check the property (not F) in the case of never, and to check F in the case of always:

$$\begin{aligned} \text{never } F &= \neg\exists \text{ path } p.\ p \models F \\ &= \forall \text{ paths } p.\ p \models \neg F \end{aligned}$$

	=	$\neg F$ is true in Spin
ever F	=	$\exists$ path p. $p \models F$
	=	not ($\forall$ paths p. $p \models \neg F$)
	=	$\neg F$ is false in Spin
always F	=	$\forall$ paths p. $p \models F$
	=	F is true in Spin

Pattern 1: Tracing

The series of actions	***[a_1, a_2, ..., a_n]***	occurs	*ever*
			always
			never

Queries derived from pattern 1 check whether actions a_1, a_2, through a_n occur in this order in the model. They need not occur consecutively: other actions may occur in between. This pattern is typically used to check a scenario. The pattern may also be used to check necessity of certain actions, for example by checking whether an action lies on the path that runs from customer to customer.

Pattern 2: Consequence

Each	action(s) from	***X***	lead(s) to	*an*	action(s) from	***Y***
All				*all*		
An						

This pattern is typically used to make sure that certain actions are executed, for example: after a decision concerning a damage claim, both the treasury department and the customer should be informed. It may also be used to check one property for two alternative paths, to see, for example, if both achieve the desired result. The query "each action from {expert judgement, standard judgement} leads to an action from {draw up rejection, draw up policy}" is an example.

Pattern 3: Combined occurrence

All actions from	***X***	*occur together always*
		occur together ever
		occur together never
		exclude one another always

The difference between "excluding one another" and "not occurring together" may need some clarification. The option "All actions of set *X* occur together never" should be interpreted as "There is no run in which all actions of set *X* occur together". The option "All actions of set *X* exclude one another always" should be interpreted as pair-wise exclusion, that is "In each run, if one action of *X* occurs, the remaining actions

in *X* do not occur". This difference is quite subtle, and not easy to explain to non-expert users. Explicit methodological guidance is a prerequisite.

Model checking is often used to rule out hazards and this pattern can be used for this purpose. An example is a complex process which includes a decision. The company wants to rule out that two different decisions are taken for one and the same case, which might happen because of splitting up the process or because of overlapping decision rules. We check whether a policy application can end up with two employees, who make different decisions using the query "all actions from {draw up rejection, draw up policy} exclude one another always".

The pattern can also be used to ensure the coupling of certain actions, like the case that both the treasury department and the customer are always informed.

Pattern 4: Precedence

Each	action(s) from	***Y***	require(s)	*an*	action(s) from	***X***
An				*all*		
All						

Pattern 4 ensures that all requirements for an activity to take place are fulfilled. A typical example is the fact that a customer should have a policy and have paid his contribution, before he can claim. The insurance company wants to make sure that if the customer does claim, without having paid his contribution, it is impossible for the claim to be settled anyway. This pattern can also be used to answer questions like what actions cause a certain customer contact and what functions are needed to create a certain product.

Of course, these patterns do not cover everything. One of the analysis questions asked for most is counting: "how often do the customer and PRO-FIT interact?". "How often is a claim checked for completeness?" Such patterns are difficult to implement without adapting the specification rigorously. Moreover, they require multiple analysis runs (can it occur once, then check if it can occur twice; if so, check for three et cetera). As yet, we have found no way to do this in an elegant way.

Furthermore, we would like to have a way to check for "model inclusion": is this AMBER model implemented by this process? We come back to this in the section on future work.

Dwyer et al. [DAC98] have worked on patterns for use in software development. Their basic elements are recognizable in our patterns:

- *Occurrence patterns*: the choice between ever, never, and always.
- *Ordering patterns*: precedence and response appearing in our patterns 4 and 2 respectively.
- *Compound patterns*: applying a pattern to more than one action at a time.

4 Implementation

In order to link a graphical tool such as Testbed Studio to Spin, including translation of queries, a number of steps must be taken in the tool. Besides that, good methodological support for the users is needed as well: a good tool without a carefully defined method still does not help; it just increases chaos instead of analyzing it. The methodological part, however, is beyond the scope of this paper.

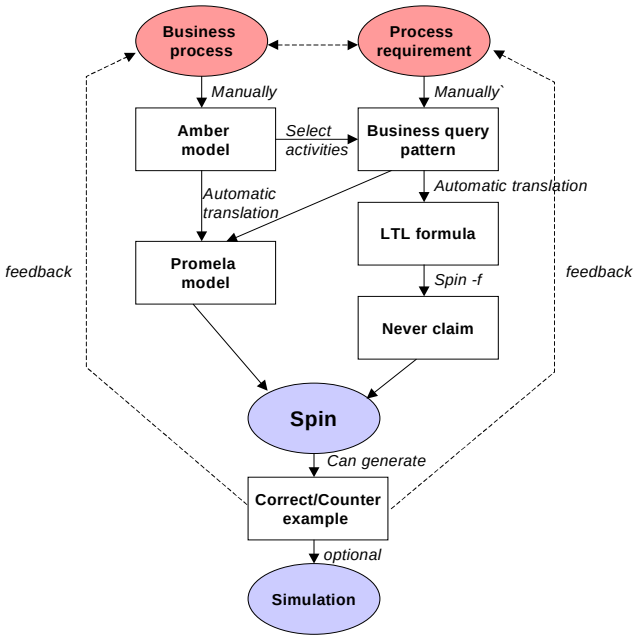

Fig. 3. The steps performed in Testbed Studio.

The approach taken in Testbed is shown in figure 3. Starting points are a business process plus the requirement that the business analyst would like to verify. The process is modeled using AMBER in Studio, and the requirement is defined in terms of the activities in the model. The model is then translated to Promela using the approach discussed in [JMMS98]. The translation uses the business query pattern to know what activities the user is interested in: the translation only generates activity occurrence information for those variables, in order to minimize the state space.
The business query itself is translated to LTL, and thereafter converted to a never-claim using the Spin LTL translator. The Promela specification and the never-claim together are then checked using Spin. If a trail is generated the outcome is translated back to Testbed Studio and visualized in the tool. The same information is also given to the simulator in Testbed Studio to simulate it, if the user wants to do so.

Not all AMBER models can be tackled by Spin. AMBER models can be non-finite state due to loops with unbounded parallelism. Models can be checked for finite-stateness before translating them to Promela (see [JMMS98]).

4.1 User interface

Model checking is only one of the analysis forms offered by Testbed Studio. The user can choose a pattern from a pull-down menu, as shown (in Dutch) in figure 4.

When a pattern is selected, the user can fill in the pattern parameters. Choices between predefined values, like between ever, never, and always, are offered in a pull-down menu. Actions and interactions are incorporated in the query by selecting them in the model, and then clicking the arrow below the input field.

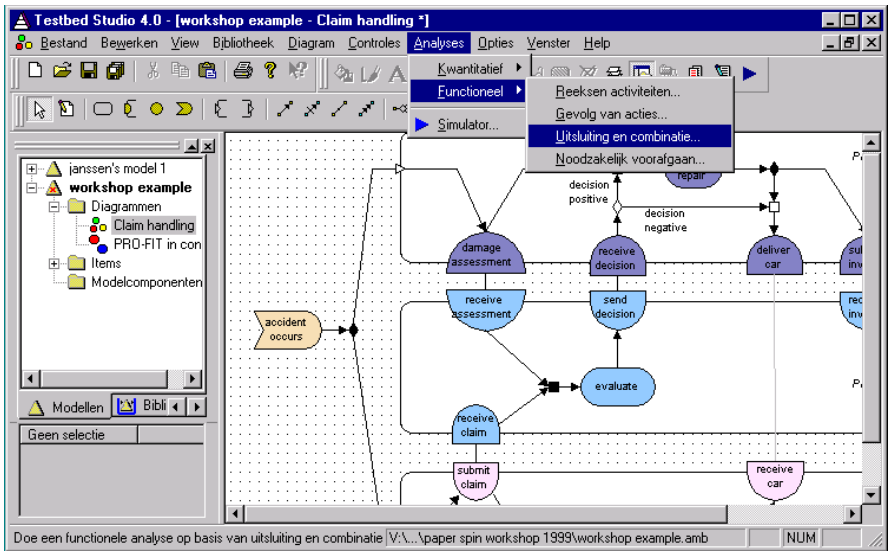

Fig. 4. Model checking in Testbed Studio (the current version of the tool is in Dutch).

The result of the query is returned in a new pop-up window. If an example is available, it is shown in the model. Testbed Studio's simulator can run the example. In figure 5 an example is shown of the question "does the series of activities *accident occurs, evaluate, pay invoice* ever occur?" Spin finds an example thereof, by using the LTL formula that states that the series is *impossible*, for which this positive example is a counter example. This trace can then be played in the Studio simulator.

Simulation can be done both using multisteps (maximal progress) or in an interleaving fashion. Interleaving is used for playback of Spin trails. By simulating the outcome of the analysis the user gets insight in why the model does not satisfy the requirement. Such a counterexample is very illustrative. In our approach we do not only give counterexamples, but also positive examples. For example, if the question is whether or not the sequence *accident occurs* followed by *pay invoice* can occur, we can show a positive example thereof. The reason for this is the fact that we can use the

query stating that the sequence can *never* occur. For this query, any counterexample is a positive example to the user.

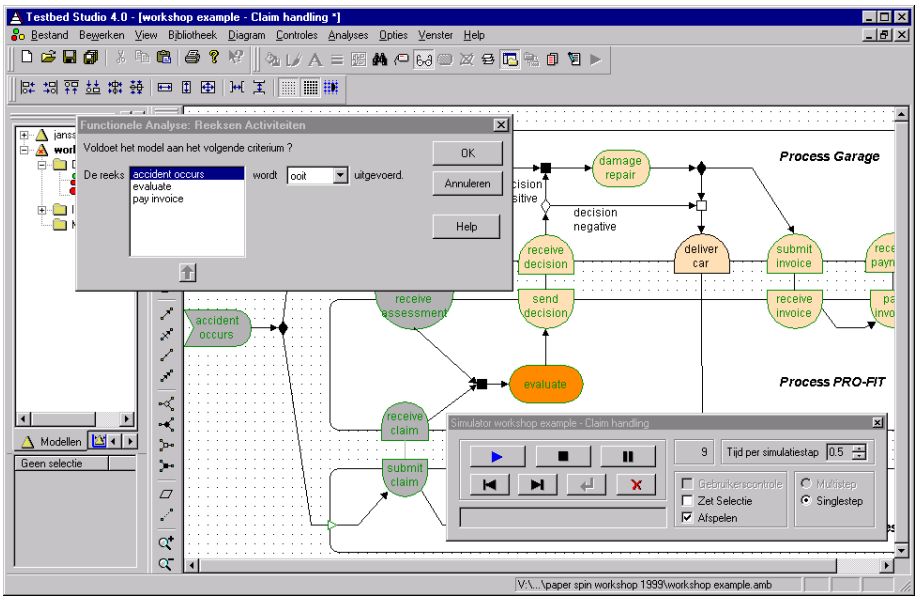

Fig. 5. Testbed Studio's functional analysis results.

5 Business query patterns illustrated

In the section on our business modeling language we introduced the insurance company PRO-FIT. Figure 2 showed the claim handling process. We now illustrate a number of the patterns above using this example.

Example of tracing pattern
Properties that the damage handling process clearly should satisfy are that a customer engaged in an accident always receives a fixed car in the end, and that the car should always be fixed before return. These properties can be checked by the queries "The series of actions [accident occurs, damage repair, receive car] occurs always" and "The series of actions [deliver car, damage repair] occurs never".

Testing the query "The series of actions [accident occurs, damage repair, receive car] occurs always" results in a negative answer. A counterexample is the result, visualizing the case of a negative decision. The query "The series of actions [deliver car, damage repair] occurs never" results in a positive answer, but without an example.

Example of consequence pattern
The option “each” can be used to check one property for two alternative paths, to see, for example, if both achieve the desired result. We might check the two possible decisions in the model with the query “each action from {receive decision} leads to an action from {deliver car}”. The query results in a positive answer.

Example of combined occurrence pattern
We extend the model to demonstrate pattern 3 and assume that in case of a rejection by the insurer, the garage offers the customer to repair the car and charge the customer.

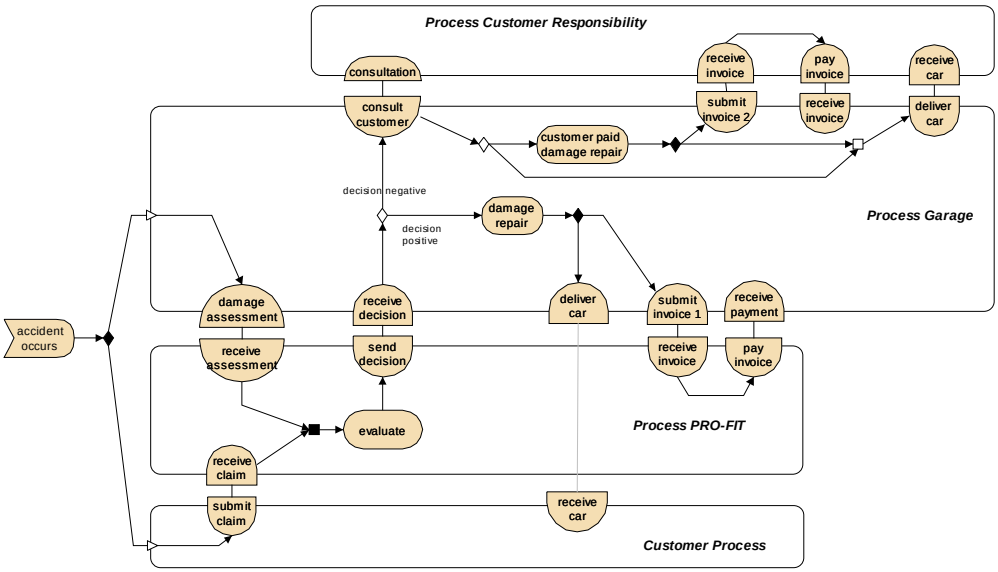

Fig. 6. PRO-FIT example extended.

We would like to make sure that the garage does not charge both the insurer and the customer for the same repair. The garage turns out to be reliable: the query “all actions from {submit invoice1, submit invoice2} exclude one another always” gives a positive result. In this case we could have asked the query “all actions from { submit invoice1, submit invoice2} occur together ever”. Although the answer is the opposite, it offers the same knowledge.

6 Expressing patterns in LTL

We translated the queries derived from our patterns into LTL. In general, the answer to the query is “yes” if and only if the LTL expression evaluates to *true*. If not, the error trace given by Spin may serve as a counterexample.

In case of the patterns containing a choice between ever, never, and always, we only translated the “never” and “always” queries to LTL. The answer to an “ever”

query is deduced from the answer to the corresponding “never” query. When a user asks an “ever” query, the “never” query is applied to the model. The resulting answer is negated to obtain the desired answer:

- if the “never” query results in a “no”, the answer to the “ever” query is “yes”. The counter example of the “never” query is an example for the “ever” query.
- if the “never” query results in a “yes”, the answer to the “ever” query is “no”.

Hence, in addition to the common counterexamples, we can return positive examples in some cases as well.

It is not possible to give an example (positive or negative) in all cases. For patterns with a choice between an, each, and all, no examples can be given when the option “an” is chosen. In that case the trace produced by Spin is no counterexample. One cannot show that “No action of X leads to ...” by means of a counterexample; one would have to show all traces from all actions. The full model is the example.

A selection of the translations of the queries is given in Table 1. In this table $X = \{ a_1, a_2, ..., a_n \}$ and $Y = \{ b_1, b_2, ..., b_m \}$. Arrows represent implication. In the LTL formulae a means that action a occurs. We use Spin syntax for LTL (e.g. <> denotes “eventually”, ! is negation, and so on).

7 Evaluating the use of Spin

In general, Spin was well suited for this application. However, there are some practical limitations to the size of the query. The query is automatically translated to a never-claim. The translation has two steps: input to LTL and LTL to never-claim. The problems lie in the second part. When the query contains too many actions, the never claim becomes so large that it causes memory problems. The limitations are determined experimentally, and turn out to be very strong. For most fields the input must be limited to three or four actions. To prevent the system to crash, we allow only manageable inputs.

As for the general use of Spin, few problems were encountered. State space exploration was fast as the size of the state space was rather limited. We tested it with models with more than 150 nodes, leading to a state vector of 36 bytes, with 1500 states and 1700 transitions. This allowed searching for smallest examples without leading to unacceptable response times (always less than 10 seconds on a PC NT Workstation).

For the definition of our patterns in some cases CTL would have been easier. Especially when looking for possibilities in models this would have allowed for a direct translation instead of an encoding.

We also had problems with fairness: fairness in Spin is much too weak a notion to be of real help. When loops are part of a model this immediately leads to unwanted answers (“will this action always be reached? No it will not, as before it the model can loop forever...”). We experimented with counters to restrict the number of iterations in loops. This, however, leads to an immediate state space explosion.

Table 1. Translation of queries to LTL

Pattern 1: does the series of actions [a_1, a_2, ..., a_n] never/ever/always occur		
1.1	never	neverafter[a_1 ... a_{n-1}; a_n] || neverafter[a_1 ... a_{n-2}; a_{n-1}] || ... neverafter[-;a_1] where neverafter[a_1 ... a_{n-1}; a_n] = ([]((a_1 -> neverafter[a_2 ... a_{n-1}; a_n])) neverafter[-;a_1] = [] !a_1
1.2	always	<> (a_1 && <>(! a_1 && <>(a_2 && <>(!a_2 … <>(a_n)...))
Pattern 2: an/each/all action(s) from set X lead(s) to an/all action(s) from set Y		
2.1	an-an	[] (a_1 -> <> (b_1|| b_2 || ...|| b_m)) || [] (a_2 -> <> (b_1|| b_2 || ...|| b_m)) || ... [] (a_n -> <> (b_1|| b_2 || ...|| b_m))
2.2	an-all	[] (a_1 -> (<> b_1&& <> b_2 && ...&& <> b_m)) || [] (a_2 -> (<> b_1&& <> b_2 && ...&& <> b_m)) || ... [] (a_n -> (<> b_1&& <> b_2 && ...&& <> b_m))
2.3	each-an	[] ((a_1 || ... || a_n) -> <>(b_1 || ... || b_m))
2.4	each-all	[] ((a_1 || ... || a_n) -> (<>b_1 && ... && <>b_m))
2.5	all-an	[] ((a_1 && ... && a_n) -> <> (b_1 || ... || b_m))
2.6	all-all	[] ((a_1 && ... && a_n) -> (<> b_1 && ... && <> b_m))
Pattern 3: do all actions from set X always/ever/never occur together do all actions from set X exclude one another always		
3.1	together always	<> a_1 && ... && <> a_n
3.2	together never	((<>a_1 && <>a_2 && ... && <>a_{n-1}) → [] ! a_n) && ((<>a_1 && <>a_2 && ... <>a_{n-2} && <>a_n) → [] ! a_{n-1}) && ... ((<>a_2 && ... && <>a_n) → [] ! a_1)
3.3	exclude one another always	(<> a_1 -> !<> (a_2 || a_2 || ... || a_n)) && (<> a_2 -> !<> (a_1 || a_3 || ... || a_n)) && ... (<> a_n -> !<> (a_1 || ... || a_{n-1}))
Pattern 4: an/each/all action(s) from set Y require(s) an/all action(s) from set X		
4.1	an-an	((!b_1 U (a_1 || … ||a_n)) || [] !b_1) || ((!b_2 U (a_1 || … ||a_n)) || [] !b_2) || ... ((!b_m U (a_1 || … ||a_n)) || [] !b_m)
4.2	an-all	(!b_1Ua_1 && !b_1Ua_2 && ... && !b_1Ua_n) || (!b_2Ua_1 && !b_2Ua_2 && ... && !b_2Ua_n) || ... (!b_mUa_1 && !b_mUa_2 && ... && !b_mUa_n) || []!b_1 || []!b_2 || … || []!b_m
4.3	each-an	((!b_1U(a_1|| ...|| a_n)) || []!b_1) && ((!b_2U(a_1|| ...|| a_n)) || []!b_2) && ... ((!b_mU(a_1|| ...|| a_n)) || []!b_m)
4.4	each-all	((!b_1Ua_1 && !b_1Ua_2 && ... && !b_1Ua_n) || []!b_1) && ((!b_2Ua_1 && !b_2Ua_2 && ... && !b_2Ua_n) || []!b_2) && ... ((!b_mUa_1 && !b_mUa_2 && ... && !b_mUa_n) || []!b_m)

Finally, we had some problems in using bit-state hashing. For our models, bit-state hashing hardly ever lead to correct answers. The reason for this is still unclear. Using different hash functions was not of any help. As we plan to introduce data, which most certainly will lead to large state spaces, this problem needs to be looked at.

8 Conclusions and future extensions

We have shown how model checking can be made accessible to a large, not formally trained audience. Our approach has been validated in real-life situations and the first results are very promising. People are enthusiastic and find this type of analysis very appealing. It complements a number of other analysis techniques that have been traditionally been applied in business modeling, such as stochastic simulation.

A number of extensions are planned after careful validation of the current approach. One of these is to allow (logical) combinations of patterns, such as conjunction, implication and "unless". This is a straightforward extension of our implementation, where, however, the limitations of the Spin LTL translator will form a severe limitation.

Currently, our business modeling language is being extended with an object-oriented data modeling language. It would be desirable to add this language to the translation to Promela as well. However, its impact on business query patterns is still not clear and should be prepared together with end users of the tool.

Finally, we would like to use AMBER models as a requirements language as well, in order to be able to check correspondence between service specifications and implementations. This would support our business modeling approach, where often one starts with a service specification of the business model to be defined, which is then refined into a detailed business model.

In principle it is possible to translate AMBER models to LTL: and-joins roughly correspond to conjunction, or-joins to disjunction. However, it then becomes unclear what part of the model can be viewed as an assumption, and what is the consequence part. The AMBER model $a \rightarrow b$ can be read both as "if *a* occurs, then *b* will occur thereafter", or as "*a* and *b* will always occur, and *a* will precede *b*". To resolve those ambiguities additional graphical syntax or annotations in models are needed.

Acknowledgement

This paper results from the Testbed project, a 120 man-year research initiative that focuses on a virtual test environment for business processes. The Testbed consortium consists of ABP, the Dutch Tax Department, ING Group, IBM and the Telematica Instituut (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.

In the early stages of our work on functional analysis Jan Springintveld and Rob Gerth participated in the project as well. We gratefully acknowledge their

contribution. We would like to thank the anonymous referees and Marc Lankhorst for their comments.

References

[CADP99] *Caesar/Aldebaran Development Package homepage*. Available at: http://www.inrialpes.fr/vasy/cadp.html

[DAC98] Property Specification Patterns for Finite-state Verification, Matthew B. Dwyer, George S. Avrunin and James C. Corbett. In: *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, March, 1998.

[EJO+99] Eertink, H., W.P.M. Janssen, P.H.W.M. Oude Luttighuis, W. Teeuw, and C.A. Vissers, A Business Process Design Language. In: *Proceedings World Congress on Formal Methods.* Springer LNCS. Toulouse, September 1999.

[FrJa98] Franken, H.M. and W. Janssen, Get a grip on changing business processes, *Knowledge & Process Management* (Wiley), Vol. 5, No. 4, pp. 208-215. December 1998.

[Gara98] Garavel, H., *OPEN/CAESAR: An open software architecture for verification, simulation and testing.* INRIA Rapport de recherche n3352, January 1998.

[HALP98] Havelund, K., M. Lowry and J. Penix. Formal analysis of a space craft controller using Spin. in G. Holzman, E. Najm and A. Serhrouchni (eds.), *Proceedings of the 4th International SPIN Workshop*, Paris, France, Nov. 1998, pp. 147167.

[Holz97] Holzman, G.J., The model checker SPIN, *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, May 1997, 279-295.

[JaEJ95] Jacobson, I., M. Ericsson en A. Jacobson, *The Object Advantage - Business Process Reengineering with Object Technology*, ACM Books, 1995.

[JJVW98] Jonkers, H., W. Janssen, A. Verschut and E. Wierstra, "A unified framework for design and performance analysis of distributed systems", in *Proceedings of the 3rd Annual IEEE International Computer Performance and Dependability Symposium* (IPDS"98), Durham, NC, USA, Sept. 1998, pp. 109-118.

[JMMS98] Janssen, W., R. Mateescu, S.Mauw and J. Springintveld, Verifying business processes using SPIN, in G. Holzman, E. Najm and A. Serhrouchni (eds.), *Proceedings of the 4th International SPIN Workshop*, Paris, France, Nov. 1998, pp. 21-36. Also available at: http://netlib.bell-labs.com/netlib/spin/ws98/sjouke.ps.gz

[Kars96] Kars, P., The application of Promela and Spin in the BOS project. In: *Proceedings Second Spin Workshop*. August 1996. Available at: http://netlib.bell-labs.com/netlib/spin/ws96/papers.html

[SMV99] CMU Model Checking Home page. Available at: http://www.cs.cmu.edu/~modelcheck/smv.htm.