Semi-Automatic Test Case Generation from Business Process Models

Tibor Bakota[1], Árpád Beszédes[1], Tamás Gergely[1], Milán Imre Gyalai[1], Tibor Gyimóthy[1], and Dániel Füleki[2]

[1] University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 546724
{bakotat,beszedes,gertom,gyalaim,gyimothy}@inf.u-szeged.hu
[2] IDS Scheer Hungária Kft.
Infopark sétány 1., H-1117 Budapest, Hungary, +36 1 4630900
d.fuleki@ids-scheer.hu

Abstract. In this work, we describe our method for designing test cases based on high level functional specifications – business process models. Category Partition Method (CPM) is used to automatically create test frames based on possible paths, which are determined by business rules. The test frames can then be used in the process of test case design, together with filtering and prioritization also given as CPM rules. We present the details of the adaptation of CPM, together with first experiences from applying the method in an industrial context.

Keywords: software testing, test case design, test case generation, functional testing, black-box testing, equivalence partinioning, Category Partition Method, CPM

1 Introduction

In test planning and test case design, functional (or black-box) tests play a crucial role. Among testing professionals, there is a common agreement that functional tests should always have higher priority than structural (or white-box) tests [1], provided that the basis upon which tests are designed (the specification, normally) is accessible and well-elaborated. One of the most commonly applied functional test case design techniques is *equivalence partitioning*, where the set of possible program inputs and states is represented by a single representative, which is believed to produce the same behavior as any other representative from the given partition. This way a limited, yet representative combination of all possible behaviours of the software will be tested.

Equivalence partitioning is a well-known technique and also very pragmatic, but unfortunately, it is rarely described in a more precise and formal way. Probably the most well-elaborated approach formally described is still Ostrand and Balcer's *Category Partition Method (CPM)* [2]. In this paper, we describe our method – which is based on CPM – for semi-automatic test case design from

very high level functional software specifications, usually in the form of various business process models. The CPM method is re-interpreted and specialized so that it is applicable for automatically generating high level test frames based on possible paths and determined by business rules in the business process model. The test frames can then be used, by the means of filtering, weighting and specialization, to create the actual test cases. As such, it is a kind of model-based test design approach employing automatic generation to aid the manual test case design process. The method: (1) ensures that no important path in functional description is ommitted, (2) ensures that all infeasible paths are eliminated and (3) provides a way to assign priorities to the generated test cases. This way, the required effort of test case design is significantly reduced while providing higher reliability in the resulting test cases.

We applied the method to one of the leading business process design frameworks and description languages, and validated its usefulness in an industrial case study. To our knowledge no previous work dealt with applying CPM to such kind of functional descriptions and reported practical benefits of this formal equivalence partitioning approach.

The paper is organized as follows. In Section 2, we overview the approach and compare it to related work. The details of the method are given in Section 3. We report on practical experience in applying the method in an industrial context in Section 4 and conclude in Section 5.

2 Overview and Related Work

In this section we give an overview of the method and elaborate on related work.

The main steps of the method are shown in Figure 1. The method works from a high-level representation of the process (hereinafter process means a business process), created by a business specialist and extended by a test designer who is aware of the implementation of the various steps. The extended model contains certain additional information about the process. From the extended model the test designer or tester prepares and finally runs the tests. Note that test designer and tester may be the same person.

The input model of our method describes the control flow of the process, that is, the order of the process steps, and their connections. This model is worked out by the process analyst. Then the test designer adds further information to the model. The amount of such extra information is usually significant but it pays off since testing with this technique will be much more elaborate and reliable. This information is clearly test-oriented and it is the duty of the test designer to derive it from the (business and technical) specification of the system. The modified model contains, for each activity (manual function by humans or system function to be carried out by software), the relevant operating parameters such as input, external system status, etc.

Although the aim of the generated test cases is to test the implementation of the whole business process (instead of testing the individual implementations of the steps) different parameter values – which have influence to the process flow

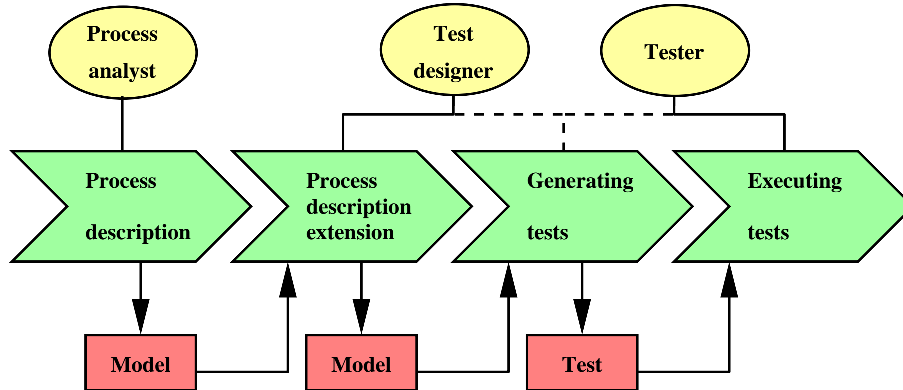

Fig. 1. Levels and steps of test generation

– must be considered as well. We used (CPM) to reduce the number of different inputs required for testing. We do not detail on this method in this paper, an interested reader should consult article [2]. The main idea of it is that the input is divided into (logically independent) parts, called *categories*. Then, for each category, the possible values of the category are grouped into partitions in a way that two values from the same partition cause the same effect during execution. This way one actual input for each partition of a category are enough to fully test the program behaviour regarding the given category.

In our interpretation of CPM, the parameters are described by categories. The possible parameter values are organized into partitions for each category that basically describe the various implementation (test) modes of the given function which are important for the business logic. During testing one value is chosen from each partition to represent the corresponding partition. At the forks (decisions) of the control flow information can be added to the branches for selecting the direction based on input categories.

Then so-called *test frames* can be generated from the model. The difference between a test frame and a test case is that a frame binds partitions to categories, while test cases bind values to inputs. Thus, test frames are specific views of the process described in the model extended with the information about input partitions assigned to the individual steps. Test cases are produced from the test frames so that an arbitrary element of the specified partition is used as input. The test case is a test scenario that can be implemented, representing a potential functional implementation of the given process.

There are two ways to reduce the number of test frames and test cases generated. First, certain unrealizable or irrelevant combinations can be filtered based on the relationship between the functions and certain properties assigned to the functions (so-called properties). Priorities can be assigned to the test frames (test cases) based on various weights (e.g. risk factor) to the functions.

There is a significant amount of literature on methods for test case generation, and a small portion of the approaches deal with business processes. However, most of these articles describe methods specialized for a specific description language (e.g. for BPEL), and the area of web services, while our method can be applied for any model (satisfying some basic requirements).

Yuan et al. propose an extended control flow graph to represent a BPEL process [3,4]. They generate sequential test paths from this representation, according to branch coverage criterion. Then these are combined into concurrent test paths and a constraint solver is used to remove infeasible paths, and to generate test data. Zhang et al. transforms web services flow into a dynamic testing model based on extended UML 2.0 activity diagram [5]. They provide a test coverage definition, and a test case generation method that generates only relevant test cases. Garcia-Fanjul et al. generate test suite specification from BPEL representation for the compostiton of web services [6,7]. A systematic procedure based on SPIN is proposed to select test cases according to a transition coverage criterion. Xiaoyan et al. generate test cases based on the OWLS requirement model to test the interaction of web services [8]. Neither of the methods above use CPM.

CPM is not a new method. It had been used in practice by testers prior to its formal definition. First Ostrand et al. described it formally in [2]. They proposed a method for creating functional test suites from system specifications by annotating the test specification with constraints. Jeff and Offutt described a formal method that extends CPM and deals with infeasible paths [9]. Offutt and Irvine combined CPM with a memory management faults detector tool to test object oriented software migrated from procedural language [10]. Vieira et al. describe a UML based test case generation method combining several existing techniques including CPM [11]. To reduce the number of generated test cases, the authors employ custom annotations and different coverage criteria. Bertolino et al. presented a CPM based approach for testing applications expecting XML documents as input [12]. To limit the number of generated instances, they employ practical strategies for handling element weights and type values.

As far as we know CPM has not been used earlier for high-level test case generation for business processes.

3 The method in detail

In this section the method is described in detail. The method is defined so that it could be used with any kind of model which meets certain requirements (see below). The method is for the testing of the process itself, and assumes that the individual functions themselves operate correctly.

In the model the inputs, parameters, outputs, and output parameters of the process can be linked to the given activities. The individual activities themselves can have additional inputs and outputs. The method assumes that the input and parameters of the process are supplied "manually" by the tester, while other inputs of the activities are filled-in through automatic dataflow within the process.

The outputs and output parameters of the process are used to determine the expected values, but can participate in automatic data flow as well. Other outputs of the activities participate only in the automatic data flow.

The method is based on CPM. The main idea is that categories are assigned to the individual process steps to be tested. A category can be anything that has significant effect on the operation of the given step, such as input data or its properties/features (e.g. size), environmental parameter, etc. Then partitions are assigned to each category. This is in fact the classification of the possible values of the given category. This classification can be done in such a way that the given step of the process should, in some way, work similarly for the values of the same partition.

The method consists of four levels in test generation:

Model level. A model is a directed graph describing the process to be tested. The graph indicates the activities and forks of the process by vertices and the possible control flows by edges. The model also contains meta-information assigned to the vertices and edges (inputs, outputs, categories, partitions, etc.).

Path level. A path is a chain constructed from the model by traversing the graph from the starting point to the end-point. The graph vertices visited several times during traversal are represented as individual vertices of the path. The meta-data assigned to the points and edges are also present here.

Test frame level. A test frame is a series of activities where each activity holds information on the type of input it takes.

Test case level. A test case is a series of functions with given input values.

The individual levels and the transitions between the levels are shown in Figure 2. During path-generation the possible paths are generated from the model, in which the filter parameters specified by the model designer are also taken into consideration. Several test frames are constructed from such paths with the help of the given categories and partitions, out of which the specified filter conditions filter out the "bad" frames. Then, with the help of weighting, some parts of the total frames can be omitted. The remaining frames make up particular test cases through selecting actual input values in accordance with the partitions.

3.1 The Model

The model is a directed graph describing the process to be tested. The model should meet the following requirements:

- The model has a starting point.
- The activities (functions) of the model indicate the parameters (any factor influencing operation, for example input or the state of an external system) they work with, and also the output of the activities.
- Categories are assigned to the individual parameters of the process. A category represents an aspect based on which the parameter values can be

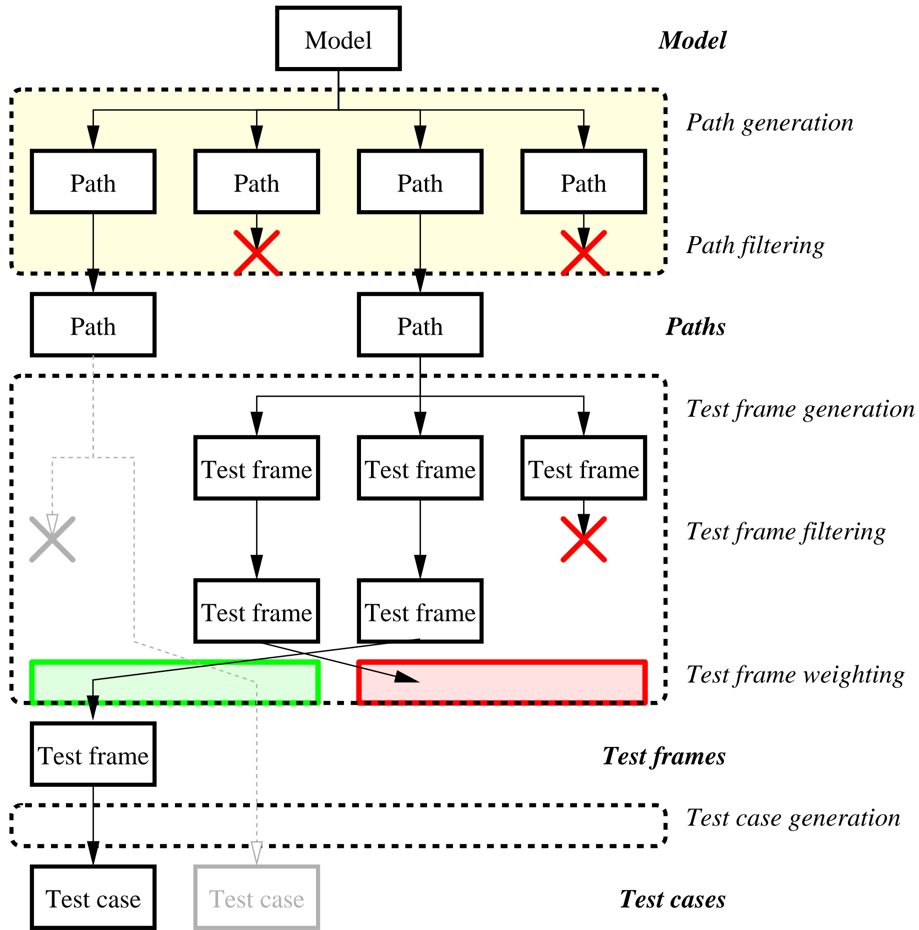

Fig. 2. Test case generation process

classified into partitions. A partition means a set of values linked by some aspect. (For example if the parameter is a string, one category can be its length, another one the features of the characters it contains.

- Those inputs of the activities that are not inputs of the process will not be classified into categories. However, since this is only possible if the output of other activities is used, these relationships should be indicated in the model.
- There are three types of forks: in case of *AND* all branches shall be executed in parallel; in case of *XOR* exactly one branch will be executed; in case of *OR* any number (but at least one) of the branches shall be executed in parallel. All fork types have their corresponding jointer counterpart as well.
- *OR* and *XOR* forks have conditions assigned to the individual branches. The conditions use the categories of the preceding functions as variables, and

their partitions as values. In case of a *XOR* fork exactly one path can be followed depending on the categories used.
- Relationship can be established between the states of the model. Such relationship could be *excludes*, in which case the two states cannot be present on the same path; or *implies*, in which case one state can only be present on a path if the other one is present as well.
- Other information can also be assigned with the elements of the model that can help processing of the model.

3.2 Generating Test Frames

In the following, we give two approaches to generating test frames. The first one follows what is a logical approach in two steps, while the second one is an optimized version combining the two steps into a single one.

Generation in two steps. In this case first all possible paths are generated from the model. A path will be a graph free of forks that contains all or part of the information assigned to the vertices and edges of the graph. The path is generated by traversing the graph from the starting point to the end point. When a function is reached the function in question is copied together with all attached information. When a fork is reached it is also copied. The branches of the forks are managed in the following way:

AND: All branches are traversed and the resulting function-series are combined. Since such branches are theoretically independent of each other, the function series of the branches can simply be placed one after the other (although more complex combining algorithms are possible). One single path is made of one single *AND* fork (disregarding forks within the branches).

XOR: One of the branches is chosen and executed. Since the choice of the branch unambiguously determines what partition-combinations could the input categories of the preceding functions use, other combinations are forbidden. n paths are generated from one single n-directional *XOR* fork (disregarding forks within the branches).

OR: The branches to be executed are selected. The partition-combinations of the preceding functions are forbidden in a similar way as with *XOR*, and the branches selected are treated as branches of an *AND* fork. Thus $2^n - 1$ paths are generated from one single n-directional *OR* fork (disregarding forks within the branches).

All possible different paths are generated from the graph, where two paths are regarded to be different if the series of functions involved is different (disregarding the information assigned). Should the model contain a loop, the number of all possible paths would be infinite. To avoid this we assume that different executions of the loop do not differ from each other. Thus, in this case a path can be generated from the loop in two ways: one, where the elements of the path are not executed, and two, where they are executed once.

In the second step of the two-step method test frames are generated from the path. If n partitions are allowed (i.e. not forbidden) for one category of the parameters of a function, then n new paths are constructed from the single path in a way that for all new paths of the category all but one partition will be forbidden for the given category. A test frame is constructed from the path when a partition is fixed for all categories of all functions.

Generation in one step. The two actions of the above two-step method are so tightly connected that they can practically be executed in parallel. Or, to be more precise, fixing partitions will unambiguously determine the paths that can be chosen at the forks, thus only the serialization of the branches of the *AND* forks (including the branches selected by the partitions in *OR* forks) shall be done in the way indicated above.

For the loops the above method can also be applied here, namely, only the 0 and 1 iteration versions are generated.

3.3 Decreasing the Number of Test Frames

It is obvious that the previous step results in a huge number of test frames. We give two methods to decrease this number, furthermore they can even be used jointly to increase the efficiency of optimization.

The features of functions and partitions. One of the methods can be used during generation to decrease the number of paths/frames. Here the idea is that certain states/partitions are forbidden in case of certain conditions, and special features are assigned to partitions and complete functions.

If a path contains two states that are in *excludes* relationship in the model, the path is not valid – thus it shall be disregarded. And if one state of the model *implies* another state that is not present in the path, this one is also omitted from among the possible paths.

To establish the pre-conditions for the partitions the categories and partitions can be used in a way similar to the one used when assigning conditions to the *OR* branches. It can also be done through defined features that could be determined by partitions or functions.

In practice this means the following. Two additional elements are assigned to each partition:

– A feature list.
– A logical expression.

Two types of features can be defined in the feature list. Special features affect the management of the partition, while other features help in establishing pre-conditions.

The logical expression encodes the pre-condition of choosing the partition. The expression describes whether it is a property or a category-partition pair.

The feature as elemental part is true in a preliminary test frame if one of the partitions already fixed in the preliminary test frame defines it, meaning that it is present in the feature-list of a partition already fixed. A category-partition pair as elemental part is true in a preliminary test frame if the given partition of the pair is fixed for the given category of the pair in the preliminary test frame. Given this, a partition with pre-conditions can only be chosen to be fixed during generation of the frames if the evaluation of the pre-condition gives a true value.

The special features of a function or partition can be the following:

unique: It is enough to include a partition or function with this feature in one single test frame. That is, if it is already fixed or chosen for at least one test frame, there is no need to consider it in generating other test frames. However, care should be taken that test frames disregarded due to unique features must not result in losing other partitions or partition combinations. For example, if partition a_1 of category A is a pre-condition of both partition b_1 and b_2 of category B, it is of no importance that a_1 is of unique feature, since omitting either of the $A = a_1$, $B = b_1$, and $A = a_1$, $B = b_2$ combination would result in "losing" one partition of B.

error: Not more than one of the partitions or functions with this feature should be chosen for a test frame. That is, if an element with this feature is already fixed or chosen for the preliminary test frame, other such choices can be omitted.

Weighting of functions. Weighting of functions decreases the number of test frames generated subsequently.

Here the test designer – based on the business specification and possibly with the help of the process analyst – assigns weights to the individual functions based on the probability values fixed in the model. Then the test frames are generated and sequenced according to a combined value calculated from these weights. This ensures that only the first limited number of test frames must be executed in case of resource limitation. Section 3.4 presents the details of this method.

3.4 Adaptive Risk Model

The aim of the adaptive risk model outlined below is to reduce the number (or establish a priority) of test frames determined through the general method in such a way that functions with higher risk levels precede those with lower risk in the resulting list. In this context risk level is a complex concept applied to single functions, with the following primitive constituents:

Importance: The higher the loss due to malfunction, the more important is the function. The following business importance categories are defined:
- *Business critical*: The malfunction of the given function has effect on the entire daily business activity. Correct operation is essential for the process as a whole.

- *Highly important*: The correct operation of the function is important. In case of malfunction the function can be bypassed, although with high cost (manual resource expenditure).
- *Important*: In case of malfunction the function can be relatively easily (at low cost) repaired/bypassed in such a way that the process as a whole would not be affected in business terms.
- *Not important*: Although the function is part of the given process as a whole, yet its malfunction does not affect the result of the process as a whole. (Later it can easily be reproduced independently of the process.)

The actual values of the Importance concept defined above are provided by empirical data for the given functions that should be fixed during modelling. For functions with no such values fixed default values should be assumed, which is '*Important*.'

Each category should – also empirically – be assigned a numerical value in such a way that the rate between the numerical values should reflect the relationship between the corresponding categories. It can be assumed that *Business critical* is assigned the value of 1, while *Not important* 0.

In addition to the above weighting of the functions the probability of each event should also be approximated empirically:

Probability: The probability of the given event (empirical estimation). The values are assigned to the edges starting out from the functions. The following categories are defined:

- *Always*: The given event always happens after the completion of the function in question.
- *Often*: The given event often happens after the completion of the function in question. In this case an alternative event should exist that happens in the other cases. The probability of the alternative event should be *Rarely*.
- *Rarely*: The function usually does not appear during the completion of the process. In this case an alternative event should exist that happens in the other cases. The probability of the alternative event should be *Often*.
- *Never*: The given event never happens after the completion of the function in question.

Real numbers are assigned to the categories defined above that indicate the probability of the events. Thus, the value assigned to the *Always* category is 1, and the value of *Never* is 0. *Often* and *Rarely* take the values of 0.75 and 0.25 respectively. If, based on the specification of the process, the function in question can only result in a pre-defined state, then this state should necessary be assigned the value of 1. In case the execution of the given function can result in several possible states and no individual values are assigned to each of the given states, then all of them should, by default, be assigned the value

$$\frac{1}{\text{number of possible states}}$$

(that is, all possible transitions have the same probability for realisation). In addition to the pre-defined categories listed above any real number between 0 and 1 could be assigned to the events with the condition that the sum of probabilities assigned to the possible events of the given functions must be equal to 1.

With respect to the probability model above it is important that the possible states resulting from the functions (and their probability), described in the description made during modelling the process, should be independent of the functions that lead to the point in question.

If the description of the process meets the above requirement and the states are assigned probability weights in the way described above to reflect conditions of real execution, then we get a so-called Markov chain which can be interpreted as abstract mathematical expression and subjected to statistical/probability examination.

For example, we can, in a simple way, calculate for any 2 functions the possibility of the transition of one process into the other one, the average number of other functions on the route, etc.

Let $\mathcal{C}(F_i)$ be the above defied risk of function F_i. Since test frames can be described as series of functions, the expected risk of a test frame can be aggregated from the costs of the functions involved. The following extension estimates the expected risk of the test frame:

Let $F_1, F_2, ...F_n$ be the functions included in the test frame. Then these unambiguously determine a T test frame. The expected risk of an F_i function can be estimated in the following way:

$$\mathcal{C}_i^{avg} = \mathcal{C}(F_i)\,\mathcal{B}(F_i),$$

where $\mathcal{B}(F_i)$ is the probability of error occurring in function F_i. If there are (empirical) data available on the extent to which the given function is error-prone, then $\mathcal{B}(F_i)$ can be approximated using these estimations. If a function is not error-prone at all (based on experience no problem occurred so far), then assigning the value 0 to the appropriate weight the expected risk of the given function will also be 0 – thus it is included in the risk estimation of the entire test frame with 0 weight as well. If the inclination to error cannot be estimated empirically, the functions shall be assigned the same weight for this inclination (e.g. 0.5). Later on these weights will adaptively conform to the values appearing in real environment.

The expected risk of the individual functions can unambiguously aggregated to the test frame containing them, as follows:

$$\mathcal{C}(T) = \sum_i \mathcal{C}_i^{avg} p_{i-1,i}$$

where $p_{i-1,i}$ is the probability that function F_{i-1} is followed by F_i in the process (this information is available, as described earlier).

The above expression estimates the expected risk in case test frame T contains an error. By calculating this value for every test frame a risk order, that

is testing priority can be established. Let us suppose that following the steps of the method we receive the output that contains $T_1, T_2, ..., T_n$ test frames. Let us take the one with the highest expected risk value and execute it. If no malfunction is observed during operation, then we decrease the weight of error-proneness for the functions involved in the execution (the extent of decrement is an input parameter of the method). Then we establish a new priority with the new risk values for the newly generated test frames. In the other case – if malfunction is observed during operation – we increase the weight of error-proneness for the functions involved in the execution. The iterative process ends if the risk of the test frame with the highest risk falls below a pre-defined limit.

3.5 Test Cases

A test frame becomes a test case if the partitions assigned to the inputs are replaced by their "representative", that is, by one particular value from the respective partition. Since test cases can be defined in several ways, we propose a flexible method. We assign a script to every partition of each category in the model that generates test data appropriate for the given partition of the given category. In a simple case it can be done through generating random values, but it is also possible to retrieve real data from an existing database for testing. In the case of more elaborate scripts the actual values already assigned to the categories of the test frame might be required for the generation of test cases. For this reason we provide call patterns with the scripts, where the parameters of the scripts are categories, and upon calling the script the actual value assigned to the given category in the preliminary test case shall be included.

Similarly, scripts can also be assigned to the outputs that generate the expected values of outputs/output parameters.

4 Application Experience

In this section we present the results of a pilot implementation of the method with one of our industrial partners from the financial sector. During the assesment of the partner's business processes, we selected one of its 33 processes and applied the method to it. The selected process was modelled in ARIS Business Architect [13] as an extended EPC diagram [14], and we used the software's own reporting module to generate test frames (it can be programmed in Visual Basic).

The first variant of the process contained 59 functions, 59 events, 29 branches (forks), and 5 loops. The test frame generation resulted in about $40,000$ unrestricted test frames. To reduce this number, the original process was divided into three distinct parts, and some refactoring were made on the branches. This optimization reduced the number of functions to 33, events to 32, branches to 10. The number of unrestricted test frames of this optimized model was about $1,500$, which was finally reduced to 11 after filtering unfeasible paths.

Then the adaptive risk model was applied on the resulted frames (using 0, 0.25, 0.75, 1 as $\mathcal{C}(F_i)$ and $p(i)$ values; 0.1, 0.2, 0.3, 0.5 as $\mathcal{B}(F_i)$ values). The priorities of the test frames varied between 5.2875 and 0.45.

Although the detailed description of the business process required much more efforts than simply modelling the process flow, applying the method resulted in a more elaborated process description. Using the automated test frame generation the formerly applied ad-hoc testing was significantly improved resulting in the improvement of testing efficiency. Although no exact data have yet been collected, according to our industrial partner the final 11 test frames were good representatives of the most important business cases of the system.

5 Conclusions and Future Work

In this paper we described a method that can be used for high-level test frame generation from a process specification. The method is not specialized to any existing process description language, instead it can be used when the model provides a particular set of required information. The method is based on CPM, and results in a test suite that covers all possible and different executions. To reduce the size of the test suite, two techniques are described. The first is based on information given in the model, and exludes infeasible test frames. The second orders the remaining frames, and selects the most important frames to be executed first. For latter an adaptive risk model was given.

An experimental implementation of the method in ARIS resulted in 11 test frames out of $1,500$ for the example business process making its testing much simpler.

In the future we plan to enhance the experimental implementation. The flexible test case generation method we proposed in Section 3.5 is too general. As its implementation depends on the different attributes of the implementation of the business process steps, we omitted it from our prototype. We plan to work out this step in a more detailed level and implement it. We also plan to implement the method in other modelling environments. In addition, a measurement on the benefits of the method should be done on a large number of different real-world business processes.

Acknowledgements

This research was supported in part by the Hungarian national grants RET-07/2005, OTKA K-73688 and TECH_08-A2/2-2008-0089 SZOMIN08.

References

1. ISTQB: International Software Testing and Qualification Board homepage http://www.istqb.org/.
2. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. Communications of the ACM **31**(6) (1988) 676–686

3. Yuan, Y., Li, Z.J., Sun, W.: A graph-search based approach to BPEL4WS test generation. In: International Conference on Software Engineering Advances (ICSEA'06), IEEE Computer Society (2006) 14
4. Yan, J., Li, Z., Yuan, Y., Sun, W., Zhang, J.: BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In: 17th International Symposium on Software Reliability Engineering (ISSRE'06), IEEE Computer Society (2006) 75–84
5. Guangquan, Z., Mei, R., Jun, Z.: A business process of web services testing method based on UML 2.0 activity diagram. In: Intelligent Information Technology Application Workshop. (December 2007) 59–65
6. Garcia-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for BPEL compositions of web services using SPIN. In: International Workshop on Web Services-Modeling and Testing. (2006) 83–94
7. Garcia-Fanjul, J., de la Riva, C., Tuya, J.: Generation of conformance test suites for compositions of web services using model checking. In: Testing: Academic and Industrial Conference-Practice and Research Techniques. (2006) 127–130
8. Xiaoyan, Z., Ning, H., Ying, Y.: OWL-S based test case generation. Journal of Beijing University of Aeronautics and Astronautics (March 2008)
9. Jeff, P.A., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: In Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94), IEEE Computer Society Press (1994) 69–80
10. Offutt, J., Irvine, A.: Testing object-oriented software using the category-partition method. In: Seventeenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'95). (August 1995) 293–304
11. Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J.: Automation of GUI testing using a model-driven approach. In: AST '06: Proceedings of the 2006 international workshop on Automation of software test, New York, NY, USA, ACM (2006) 9–14
12. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Automatic test data generation for XML schema-based partition testing. In: AST '07: Proceedings of the Second International Workshop on Automation of Software Test, Washington, DC, USA, IEEE Computer Society (2007) 4
13. IDS Scheer: ARIS Business Architect http://www.ids-scheer.com/.
14. Kim, Y.: Process modeling for BPR – Event-process chain approach. In: 16th International Conference on Information Systems. (1995) 109–121