Leirios Test Generator: from Research to Teaching, through Industry

Frédéric Dadeau, Jacques Julliand and Régis Tissot [1]

Laboratoire d'Informatique
Université de Franche-Comté
16 route de Gray
F-25030 Besançon, FRANCE

Abstract

Once upon a time, at the far far away University of Franche-Comté, existed a research prototype named BZ-Testing-Tools. This tool provided symbolic animation features and test generation from models written as B abstract machines. Based on the scientific and industrial success of the tool, a company had been created, named LEIRIOS Technologies. The BZ-Testing-Tools environment was transferred in the company and redesigned, giving birth to the LEIRIOS Test Generator (LTG). This tool is now being used in the industry by embedded software developers in order to assist them in the validation phase of their product. LTG is still used in the computer science department of the University of Franche-Comté for which it provides a tool support for the practical sessions of the software engineering courses. We illustrate the features of the tool and its use as a teaching environment for the B method on a concrete application of formal modelling. The main idea is that modelling for test generation is a motivating way to introduce formal methods. Teaching the complete application of a model-based test generation process is thus made possible by using the LTG tool. Our students are drawing a benefit from this experience and live happily everafter, using formal methods.

Keywords: BZ-Testing-Tools, LEIRIOS Test Generator, B abstract machine, model-based testing.

1 Introduction

When designing a system by modelling it first, it is important to ensure two things. First, that the model behaves as expected and satisfies some properties, and, second, that the system conforms to the validated model. The first issue can be addressed by employing proof techniques, even if they can only guarantee the preservation of a given set of properties, but not that a system behaves as intended. The second issue can either be solved by employing a dedicated methodology for designing the software or by testing the implementation against the model.

The B method [Abr96] is a quitely used formal modelling technique that makes it possible to describe a complete system, from an abstract model to its implementation, through refinement steps. Each step consists of adding new information in the model w.r.t. the previous level. Proof obligations are associated to each step,

[1] Email: {dadeau,julliand,tissot}@lifc.univ-fcomte.fr

in order to ensure the correctness of the whole workflow. At the abstract level, establishing the proof obligations ensures the correctness of the model w.r.t. the invariant properties, meaning that the initialization of the model establishes the invariant, and the operations preserve it. At each refinement step, the proof obligations ensure that the newly added details preserve the coherence of the system w.r.t. the previous abstract level. This approach is complete, and produces a fully-verified system. Nevertheless, we have noticed that putting it into practice is not so simple, and the targeted audience of this method, industrials and students, are reluctant to use it.

Contrary to this "classical" use of B –where modelling objectives are to derive a correct implementation from the model– we choose to apply an approach where modelling does not aim at producing an implementation but aims at *testing* it. By eluding the successive refinement steps and considering only one B machine with a more or less large abstraction level, this approach is easier to use. Nevertheless, the absence of refinement has to be balanced by other mechanisms in order to related the abstract model with the concrete application. These mechanisms are captured by the adaptation layer, mapping tables of operations of the model and the commands available in the test bench. In this context, B is not restricted to modelling program behaviors, but it can be used to model various systems, such as an automotive windshild wiper controllers, or the security in an airport.

In 2001, the design of a complete framework dedicated to the B notation, that takes these observations into account, started at the University of Franche-Comté, in the LIFC computer science research laboratory. This environment, named BZ-Testing-Tools [ABC+02], takes B abstract machines as input and uses them for generating tests, through a black-box approach [Bei95]. First, BZ-Testing-Tools (BZ-TT) provides a *symbolic animator* in order to validate the model. This semi-automated process consists, for a user, in selecting the operations of the model he/she wants to activate. The system computes the resulting state according to the generalized substitution of the operation in the B machine. The user has then to make sure that the behavior of the models corresponds to what he/she intended. This first feature of the BZ-TT framework is complementary to the verification of properties. It thus makes it possible to validate the behaviors of the model, that will then be used for test generation purposes. Second, the *test generator* feature is dedicated to the automated computation of test sequences from the previously validated model. It extracts test targets as system states that make it possible to activate each operation. Then, the engine automatically computes a test sequence that reaches each of these targets. Coupled with the call of the tested operation, a minimal test case is build, enriched with inputs of the validation engineer.

During its design at the laboratory, the BZ-TT environment was regularily used in the software engineering courses, since it provides an efficient and playful means for validating a B model. It received good feedback from the students, who provided free beta-testing manpower. The BZ-Testing-Tools technology has been transferred in 2003 into the new born company called LEIRIOS Technologies, to be industrialized under the name LEIRIOS Test Generator [JL07]. The new version of the tool is still in use in the computer science teaching department. Whereas the use of the BZ-TT environment was limited to manipulating the animator, we have decided to

provide a full overview of the possibilities of LTG to our students. Therefore, we have designed a class project to be realized during the practical sessions [DT08]. This project consists in a concrete application of a model-based testing approach, from the design of the B model to the run of the tests on set of faulty implementations, through the test generation. This paper also reports on this experience.

The content of the paper is organized as follows. Section 2 introduces the BZ-Testing-Tools framework and its principles. The use of LTG as a tool-support for teaching software engineering is described in Sect. 3. Finally, Section 4 concludes.

2 BZ-Testing-Tools: Features and Principles

The BZ-Testing-Tools environment [ABC+02] is dedicated to the validation of systems specified by B machines. The Z extension had been studied in [Utt00]. The whole process is depicted in Fig. 1. It starts by designing a formal model that is animated in order to ensure its conformance w.r.t. the initial requirements. Then, the model is used as an input for the tests generation process, which automatically generates tests according to model coverage criteria selected by the user. Dedicated scripts are then used to concretize the tests in order to run them on the system under test (SUT). This section presents the two main features of the BZ-Testing-Tools framework: the symbolic animator, and the test generation engine. For each one, the principles are described.

2.1 Symbolic Animation of B Models

The first feature proposed by BZ-Testing-Tools is the possibility to perform *symbolic animation*. Animating a model is a semi-automated tool-supported process which consists in simulating the execution of the model, in interaction with the user which chooses the operations to activate and parameters to pass to these operations. The animation engine computes the resulting state which is displayed in the GUI of the tool. The user checks that the system behaves as expected in the informal require-

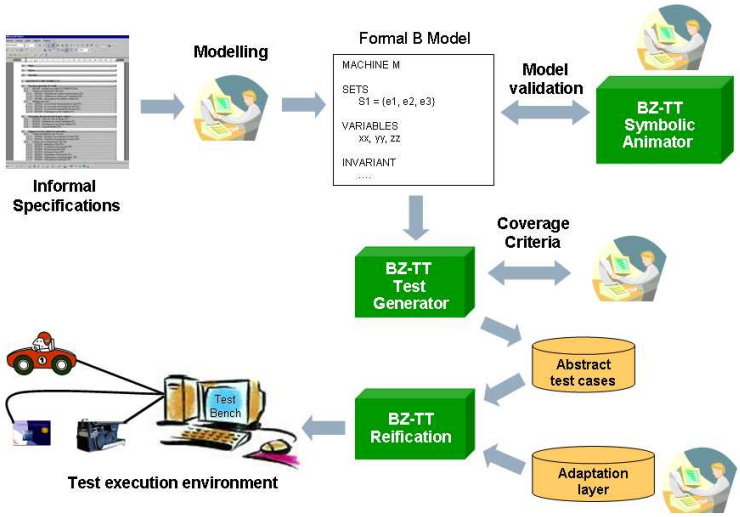

Fig. 1. Process associated to the BZ-Testing-Tools Framework

ments from which the model has been designed. This validation task completes the verification that can be done by proving the correctness of the system w.r.t. the invariants. This is the principle of another B animator, ProB [LB03].

The animation of the models is based on the decomposition of the operations into *behaviors*. Behaviors can be seen as paths in the control flow graph extracted from the generalized substitution of a B operation.

Example 2.1 (Extraction of the behaviors). Let us consider an operation to illustrate the extraction of operation behaviors. The `checkPin` operation hereafter is typical from smart card applications which are the main industrial use of the BZ-TT approach. This operation describes the (simplified) verification of a PIN code given as a parameter (`p`) against the current value of the PIN (`pin`). Notice that this verification is performed only if the parameter is correct and if the PIN differs from its initial value, considered to be -1. The effect of the operation is to update the `isHoldAuth` variable that indicates the authentication status of the user. It also instantiates the result status word `sw` that indicates whether the verification succeeded or failed.

```
sw ← checkPin(p) ≙
   IF p ∈ 0..9999 ∧ pin ≠ -1 THEN
      IF p = pin THEN isHoldAuth := true || sw := ok
      ELSE isHoldAuth := false || sw := wrong_pin
      END
   ELSE sw := wrong_conditions
   END
```

This operation contains three behaviors, that are represented using before-after predicates in which the after values of the state variables are primed:

(i) `p` $\in$ 0..9999 $\wedge$ `pin` $\neq$ -1 $\wedge$ `p = pin` $\wedge$ `isHoldAuth' = true` $\wedge$ `sw = ok`

(ii) `p` $\in$ 0..9999 $\wedge$ `pin` $\neq$ -1 $\wedge$ `p` $\neq$ `pin` $\wedge$ `sw = wrong_pin`

(iii) (`p` $\notin$ 0..9999 $\vee$ `pin` $=$ -1) $\wedge$ `sw = wrong_conditions`

The symbolic animation feature makes it possible for the user to leave some parameters of the operations unspecified. As a consequence these parameters are replaced by symbolic values. The same applies principle applies to the state variable whose values are related to the parameters. Thus, instead of considering concrete states, in which the state variables have a defined and concrete value, symbolic animation considers groups of concrete states, named *symbolic states*. The symbolic values of variables and parameters are managed by underlying customized constraint solvers [BLP04] that make it possible to valuate (i.e. instantiate) these variables, in order to valuate the corresponding execution sequence afterwards.

Example 2.2 (Symbolic animation). Consider the following execution sequence: `init`; `setPin`(X_1); `checkPin`(1234)
in which the `setPin` operation is used to set the value of the PIN code according to the following B operation:

`sw ← setPin(p) ≙ IF p ∈ 0..9999 THEN pin,sw := p,ok ELSE sw := ko END`

The call to operation `setPin` contains a symbolic parameter, which makes it possible to activate the two behaviors of this operation. The first behavior is activated if $\texttt{p} \in 0..9999$. Thus, a symbolic value is used to represent the value of the parameter X_1 with the associated constraint $X_1 \in 0..9999$. Since the `pin` state variable is related to the parameter, its after value also becomes symbolic X_2, with the associated constraint $X_2 = X_1$. The second behavior, which has no effect apart from instantiating the status word, is activated if $\texttt{p} \notin 0..9999$.

The animator GUI provides the user a way to visualize the possible behaviors to activate. He then has to choose the one from which the animation will continue [2]. Suppose that the user selects the first behavior of the `setPin` operation.

The call to the second operation, `checkPin`, detailed in Example 2.1, with a concrete parameter value will have an influence of the current (symbolic) value of `pin`. Because of the value of `p`, it is only possible to active behaviors (i) and (ii) from `checkPin`. If behavior (i) is activated (`pin=p`) then the symbolic value representing the `pin` variable X_2 is instantiated to 1234, and, because of constraint $X_2 = X_1$, we have $X_1 = 1234$. If behavior (ii) is activated ($\texttt{pin} \neq \texttt{1234}$), constraint $X_2 \neq 1234$ is added and propagated so as to obtain: $X_1 = X_2 \wedge X_1 \in 0..1233 \cup 1235..9999$.

2.2 A Model-Based Test Generator

The BZ-TT Test Generator [LPU02] is used to automatically compute tests from a B machine. It considers a model coverage criterion that consists in activating all the behaviors of each B operation. We present here the principles used to generate the tests, and the possible coverage criteria the user can input in order to drive the test generation. Finally, we briefly explain how the generated tests are concretized into executable tests.

2.2.1 Test Targets and Test Cases

Each behavior is composed of an activation condition, i.e. a predicate that has to hold for the behavior to be activated. This predicate corresponds to the before part of the behavior (that does not present any primed variables, or result values) and it is called the *test target*. The computation of the test target is inspired from [DF93] which considers a DNF decomposition of a before-after predicate describing the operations.

Example 2.3 (Test target computation). Consider the `checkPin` operation introduced in example 2.1. The test target corresponding to the different behaviors are the following.

1. $\texttt{p} \in 0..9999 \wedge \texttt{pin} \neq \texttt{-1} \wedge \texttt{p} = \texttt{pin}$ behavior (i)
2. $\texttt{p} \in 0..9999 \wedge \texttt{pin} \neq \texttt{-1} \wedge \texttt{p} \neq \texttt{pin}$ behavior (ii)
3. $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \texttt{-1}$ behavior (iii)

The composition of BZ-TT test cases are shown in Fig. 2. Each test starts with a *preamble* that is a sequence of operation calls that reaches a state satisfying the test target. This preamble is automatically computed by the tool. The test *body*

[2] The animator GUI also provides backtracking features, that make it possible to rewind the animation.

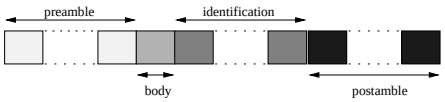

Fig. 2. Composition of an LTG Test Case

is the activation of the behavior itself by invoking the operation with appropriate parameters values. The *identification* is used to perform calls to specific operations, used to observe the system state, and thus, to deduce the verdict of the test. These operations are selected by the user which associates a different set of observers for each tested operation. The optional *postamble* feature makes it possible to reset the system in order to chain the test cases on the system under test. The postamble is optional and, unlike the preamble, it can be automatically computed by the tool.

The preamble and the postamble are computed by performing the exploration of the symbolic state space by the tool. To achieve that, a customized best-first algorithm [CLP04] is put into practice in order to find the shortest path that leads to the target state. This algorithm uses the symbolic animation engine and performs calls to the different operations of the model by systematically leaving the parameters unspecified. The reasearch is bounded on the size of the preamble, the worst case complexity of the algorithm is $\mathcal{O}(n^d)$ in which n is the number of behaviors to consider and d is the depth of the research. Once a symbolic state satisfying the test target's predicate is reached, the computation stops and the corresponding operation sequence is then instantiated in order to concretize the values of the parameters. This process is done by a labelling procedure of the constraint solver that instantiates the symbolic variables to a value that satisfies all the constraints.

2.2.2 Coverage Criteria

As shown in Fig. 1, coverage criteria are selected by the user in order to drive the test generation. BZ-TT considers two kinds of coverage criteria: a decision coverage criterion and a data coverage criterion.

The *decision coverage criterion* makes it possible to improve the test cases by decomposing the disjunctions contained in the test target predicates. This decomposition is based on four rewriting rules of the disjunctions that split the test targets, and, as a consequence, the final number of tests.

Example 2.4 (Decision coverage criteria). Consider test target #3 from example 2.3, which contains a disjunction $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \text{-}1$. The following rewritings can be applied in order to improve the preciseness of the targets.

RW1: $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \text{-}1$. When one of the literals is satified, no matter which one it is, the criterion is satisfied. This rewriting satisfies the Decision Coverage (DC) criterion.

RW2: $\texttt{p} \notin 0..9999 \;[]\; \texttt{pin} = \text{-}1$. The two literals are considered independently, splitting the original test target in two (separeted by []). This rewriting satisfies the Decision and Condition Coverage (D/CC) criterion.

RW3: $\texttt{p} \notin 0..9999 \wedge \texttt{pin} \neq \text{-}1 \;[]\; \texttt{p} \in 0..9999 \wedge \texttt{pin} = \text{-}1$. The two literals have to be exclusively satisfied, splitting the original test target in two. This rewriting satisfies the Full Predicate Coverage (FPC) [OXL99] criterion.

RW4: $\texttt{p} \notin 0..9999 \wedge \texttt{pin} \neq \text{-}1$ [] $\texttt{p} \in 0..9999 \wedge \texttt{pin} = \text{-}1$ [] $\texttt{p} \notin 0..9999 \wedge \texttt{pin} = \text{-}1$. This rewriting considers all the possibilities for satisfying a disjunction, splitting the original test target in three. It satisfies the Multiple Condition Coverage (MCC) criterion.

The *data coverage criterion* makes it possible to specify how the different data, state variables or parameters, have to be covered. The possible choices are "one value", meaning that a single value that satisfies the constraints will be selected, "all values", meaning that all the possible values that satisfy the constraints will be selected, "boundary value" meaning that a numerical data can instantiated to the extrema of its domain. This coverage criterion is employed to define the coverage of the state variables in the test targets, but also to instantiate the parameters after a preamble computation.

Example 2.5 (Data coverage criteria). Consider the last operation sequence issued from example 2.2 ending with the call to the `checkPin` operation using an incorrect pin code (different from 1234). Applying the boundary value coverage on the sequence would produce the two following instantiations: `setPin`(0) and `setPin`(9999), producing two different test cases.

2.2.3 Reification of the Abstract Test Cases

At the end, this test generation process produces abstract test cases[3]. The final step is to concretize the tests in the reification phase. Therefore, the step relies on an *adaptation layer* which is composed of a test translator that computes the tests into the test bench syntax (e.g. JUnit for a Java implementation). It also contains mapping tables that relate the abstract model values to their corresponding concrete values on the SUT. At this level, the user is required to input a correspondance table that maps the abstract names of the model items, more precisely constants and operation names, to concrete ones (e.g. Java constants and Java methods names for a JUnit reification).

The oracle of the test, that decides whether it passes or fails, is deduced by comparing the outputs of the test bench w.r.t. the expected results computed on the model. Here again, the adaptation layer provides the translation/correspondance between these two abstraction levels.

2.3 From BZ-TT to LEIRIOS Test Generator

The LEIRIOS[4] company has been created by Pr. Bruno Legeard, based on the experience and reputation acquired within the BZ-Testing-Tools project. Indeed, this framework has been used in various industrial partnerships, including cutting-edge industries such as Schlumberger/Parkeon (ticket and parking solutions), Axalto (now known as Gemalto) (smart cards) and Peugeot (automotive constructor). These studies have shown the efficiency of the automation of the test design, which provided a better coverage of the code than manual tests.

[3] These tests are said to be abstract because they were computed on the model. They are opposed to concrete tests that can be directly run on an implementation.

[4] http://www.leirios.com

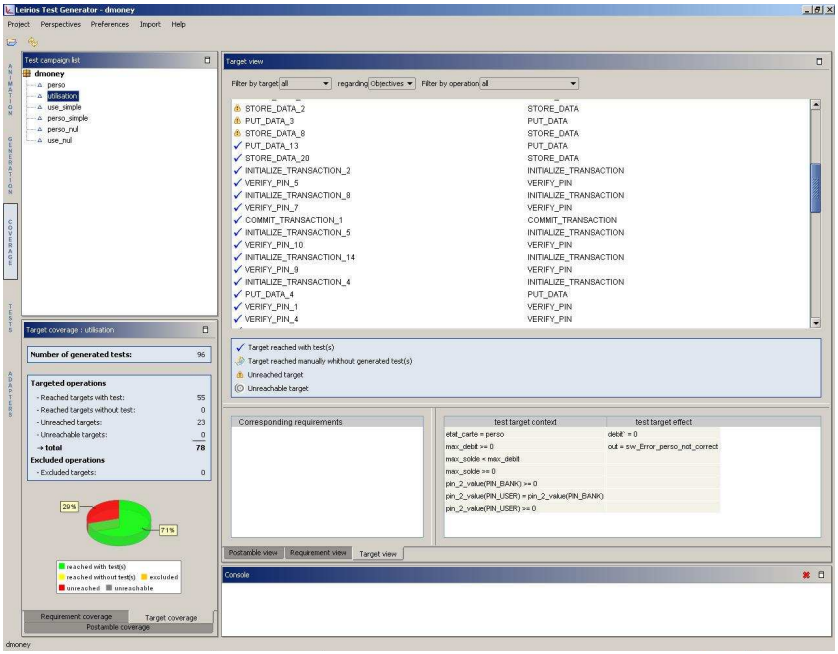

Fig. 3. Screenshot of the LEIRIOS Test Generator GUI

The industrialization and commercialization of the environment, renamed to LEIRIOS Test Generator, gave the opportunity to improve or specialize the approach to adapt it to the needs of industrials [5]. The main evolutions are the following:

- The symbolic animation feature disappeared. It is now replaced by a "classical" animation mechanism in which all the parameters have to instantiated by the user. The reason is that this feature may be interesting for researchers, but not for the industrials.
- Traceability issues of the test cases are taking a central piece. The model operations can now be annotated to introduce *requirements* [BJL+05] relating a piece of the model with an informal need expressed in the specifications. The produced test cases can be related to the original requirements.
- Reports on the coverage of the test targets and/or the requirements coverage are now displayed to the user.
- In order to deal with unreached targets, a special mechanism, named *preamble helpers* has been introduced to manually design execution sequences with the help of the animator. These sequences can be used as preambles for reaching a test target.

LTG is now a complete model based testing solution, of which a screenshot is displayed in Fig. 3. It is used by industrials, but it can also be used as a tool-support for teaching formal methods, as we will now describe.

[5] Notice that industrials are like students: they prefer a tool that works, is displayed in a colorful graphical user interface and provides a lot of interesting services, rather than a research prototype that may be more extensible but less attractive visually, and potentially error-prone.

3 Teaching Software Engineering with LTG

The software engineering course is focused on teaching formal methods, through the learning of the B method to masters' students. The theoretical course covers the scope of the B method, from the abstract machines to the implementation through refinement, proofs and machine composition. The tutorials are dedicated to the application of parts of the methodolgy on several specific exercises. We have decided to focus the content of the pratical sessions on very applied exercises that illustrate to the students a real-life use of formal methods, that also corresponds to our working topic as researchers.

During theoritical sessions and tutorials, our main objective is to teach the B method to the students. Nevertheless, for the practical sessions, our purpose is to encourage students to practice modelling, thus we choose a less reluctant way (for students), which is modelling for testing. This approach has two advantages. First, the modelling work is quickly concretized by using model animation and test generation. Second, it introduces the test to students in a practical way. This approach permit to motivate students to practice modelling and testing which are two notions that they will learn into details in the next year.

As stated in the introduction, the use of a test generation tool is recurring since 2002. At first, a single session, tool-supported by the BZ-Testing-Tools framework, was considered. Now, the maturity of the toolset, especially since the commercial version LTG exists, makes of it a reliable tool on which a complete class project can be led. Thus, we have defined a project that consists, for our students, in employing the full approach of model-based testing on a realistic context, that shows them one of the most widespread application of formal methods.

Before presenting the content of the project and to show the influence of LTG on this project, we first introduce the notion of test-based modelling, that is a different approach from designing a model for properties verification purposes.

3.1 How to Design a Model for the Test?

The test-based design of a model is different from modelling a system for verifying properties or generating code. We first present the differences between the two approaches, then we introduce the test-based modelling technique that is taught to our students. This notion is related to the concept of *design for testability* which consists in adding test-specific features in a system.

3.1.1 Singularity of Test-Based Modelling

As for all design activities, software modelling depends on its final goals. For instance, models can be used either to generate verified concrete code through a refinement and proof process (e.g. using the B method) or to test the conformance of an implementation w.r.t. its specification (e.g. using model-based testing). In this latter, we identify four modelling issues which need to be taken into account during the model design: (*i*) the scope of the test that determines the abstraction level of the model, (*ii*) the abstraction gap between the model and the implementation, (*iii*) the observability of the system under test (SUT), and (*iv*) the restrictions of the test generation technology.

The choice of the model abstraction level depends on what the user wants to test. For instance, consider the validation of a smart card. If the validation objective is to ensure the conformance of the PIN protection system w.r.t. the specification, then we do not need to model data encryption or the other authentication protocols (e.g. asymetric key generation,...).

If the choice of a high abstraction level simplifies the modelling of an application, it creates an abstraction gap between the model and the implementation. Thus, we have to bridge this gap in order to run any test generated from the model on the system under test. Basically, we use an adaptation layer which translates the abstract tests cases (generated from the model) into concrete tests cases (ready to be executed). But sometimes, the developement of an adaptation layer is not easy. For instance, the abstraction of the data encryption feature in the model of the smart card is problematic, because in order to run our test cases on the implementation, each message have to be encrypted before being sent. To solve this problem, we need that the adaptation layer recreates the mechanisms that have been abstracted during the model design. So, choosing an abstraction level for the model of a system is more difficult than choosing what we want to test. It also depends on how it will be possible to concretize the abstract tests.

Observation of the system under test is a crucial problem in the black-box testing domain. In the context, the verdicts only depend on observable elements, through the returned values of the operation calls. For instance, in smart card applications, all the operations can always be invoked (their precondition is true) and return either a success status word or an error code which indicates the cause of the error. An abstraction choice that would consider only two values, e.g. *error* and *success*, is not very relevant, because it reduces the accuracy of the tests verdicts. Indeed, it is not possible to distinguish the different an error from another. Thus, it is very important to identify the informations which can be used to improve the test verdict and to take them into account during the modelling of the system.

The proximity of the data model between the two abstraction levels is crucial for the observation issues. Since the verdict is obtained by comparing the results from the model and the system under test, it is mandatory to be able to compare them. This implies using "compatible" data structures that are the same in the two formalisms (e.g. integers) or that can be translated from one formalism to the other (e.g. functions vs. arrays).

Moreover, the test generation technology is limited by the complexity of the search algorithms. In order to deal with it, it is sometimes required to take into account this issue during the modelling step. Three modelling parts have some influence on this issue: (*i*) the parameters instantiation, (*ii*) the abstraction level, and (*iii*) the initialization of the system.

When modelling for testing, it is mandatory to instantiate all the parameters of the model. Indeed, in order to be able to design relevant tests to be run on the system, the model has to present the same "characteristics" that the system under test. For example, consider a model representing an elevator parameterized by the number of floors. In order to generate the tests for a real elevator (e.g. built in a skyscraper), it is necessary to precisely know how many floors are considered, since

the content of the test sequences may depend on it (e.g. in a test that requests to reach the next floor until the roof). As a consequence, the parameter has to be instantiated.

The abstraction level of the model plays an important role in the reduction of the combinatiorial explosion. This is due to the fact that a very low level of abstraction introduces a lot of unnecessary behaviors in the model w.r.t. the testing scope. For instance, in the smart cards domain, the modelling of the communication protocol between the card and the terminal introduce an unecessary complexity in the model, so it is a good choice to abstract it (as long as this part of the requirement can be restablished by the adaptation layer).

The choice of a good initial state for the model is also important to improve the test cases building. If it is possible to defined one (or more) initial state(s) of the system which reduces the length of the operation sequences to reach the targets. Then, it may reduce both the generation time and the number of unreached test targets. For example, consider a model of a file system; most of the generated tests will require that specific files and directories exist. If the initial state is an empty file system, then the preamble will have to call the operations that create the appropriate configurations for the different tests. Whereas an non-empty file system with existing files and directory will require less effort to the engine, improving its computation time and its chances to reach the most complicated test targets.

Finally, when the test generation is off-line, in order to establish the most accurate conformance relationship between the model and the SUT, the model must be deterministic. In the BZ-TT/LTG approach, a non-deterministic model is not problematic for the test generation, but it may become difficult to establish the verdict when the resulting behavior and values are not the ones that were expected. One solution is to resort to an on-line testing solution, by embedding the model through assertions inside the code of the program that are evaluated at runtime, e.g. by translating the B pre- and postconditions into JML [BCC+05] assertions in a Java program.

3.1.2 Specification Method

Consider software systems in which only the controllable services modify the handled informations and return the observable informations. We propose a method with five steps inspired of [UL06]: (*i*) choose the goal of the test, (*ii*) identify the controllable services and the observable informations, (*iii*) choose the operations of the model separating the controllable and the observable ones; design their signatures and choose how to represent the parameters; design the PROPERTIES clause, (*iv*) choose the state variables and design their types; describe the INVARIANT by expressing invariant properties that have to be verified by the model, (*v*) design the behaviors of the operations by pre- and postconditions; express them as conditions and generalized substitutions.

To choose the test goal many different situations can be considered, depending on the scope of the test and the hypothesis on the environment. For example, we distinguish the case of generating functional tests for the nominal cases from the case of robutness test generation. In the former case, the model contains only nominal cases of interactions with the system. In the latter case, additional elements have to

be taken into account in the model so that the relevant test data and test context can be targeted by the approach. For example, this may result in establishing a defensive model of the system, which adds the behaviors that were not described in the informal specification.

The controllable services will be modelled by operations whose one must define the signature from its definition in the SUT and from the abstraction level of the model. To an implemented service can be associated one or many services of the model. In general, as the model is more abstract than the SUT, a service of the model may be associated to a composition of several services of the SUT. This is the adaptation layer that re-establishes this mapping. The test driver determines the conformance relation between the expected results defined by the model and the results computed by the SUT.

The operation of the model being chosen, we then design their parameters' list and their results. We have to choose their types and how to represent them. Some will be constants to limit the combinatorial explosion if it is compatible with the goal of the test. The design of the state variables is guided by the abstraction level. When the state variables are chosen, we have to design their types and specify their properties in an invariant.

The preconditions of the operations are established according to the test purpose and the kind of operation (controllable or observable). When the robustness is targeted, the specification of the controllable operations is likely to be defensive and the preconditions are only typing conditions of the parameters. On the contrary, if the goal of the test is limited by assumptions on the environment of the system, then the preconditions are strengthened in order to specify the targeted cases. The preconditions of the observable operations express the conditions in which the environment triggers them. The postconditions are defined by the design of generalized substitutions that modify the state variables. The exceptional cases are ordered by the `IF` imbrications designing them in a deterministic manner. The multiple error cases have to be considered separately, or the conformance relationship has to be softened, otherwise, false negative tests can be reported, if the order of errors treatment differs between the implementation and the model.

Moreover, the design of the conditions and the coverage criteria may have an influence on the generated tests. These latter two can be combined in different ways to take the goal of the test into account. For example, condition $x \in \{1..3\}$ will require to correctly drive the test generation using the appropriate data coverage criterion in order to produce three test cases (one for each value of x), that would have been covered by writing $x = 1 \vee x = 2 \vee x = 3$ instead and selecting the appropriate condition coverage criterion.

3.2 Using LTG on a Realistic Application

In this part, we present the project we designed and proposed to the students, in order to reach two objectives. First, it aims at evaluating the students with a realistic case study and to lead them to consider the usefulness of the formal methods for software testing. We try to reach this goal by applying a full model-based testing approach in order to validate a realistic application. Second, our idea

is to introduce this project as a challenge for the students in order to motivate them and to force them to do their work seriously. To accomplish this objective, we ask them to detect faulty programs among a set of different implementations of the same specification. In this project, we rely on the previous modelling techniques to help them designing their model. LTG is used as a tool-support that makes it possible for them to generate the tests.

3.2.1 The Challenge and its Rules

Like any challenge, this project is governed by some rules. Here, we present these rules and the related workflow, given in Fig. 4, that the students have to follow in order to bring their project to fruition.

Each pair of students have to design a B model of the system w.r.t. the specification document provided. They also have to define some invariant properties over their model –some properties are explicitly described in the specifications, and some others are suggested. In order to ensure the correctness of their model, the students have to perform some proof using Click'n'Prove [AC03] to verify the preservation of the invariants properties. They also have to validate the behavior of their model by using the LTG animator. These two complementary approaches (proof and animation) provide an interesting tool-support to design a model, little by little, by successive round trips between the model, the prover and the animator.

Once their model is verified and validated, the students have to generate functional tests using the LTG tool. Here, they have to apply what they have seen previously about the choice of the coverage criteria parameters and the observation phase. These parameters are very important in order to generate tests which cover most of the interesting behaviors of the system and which enable to establish accurate verdicts. The abstract tests generated by the students (saved in XML files) need to be concretized in order to be executed over the real implementation (a Java program) of the system. We provide to them an adaptation layer which translate a XML test file into a JUnit test campaign. This translation is not fully automated: the students have to define a mapping between the operations (resp. parameters) names and types defined in the model and the operations (resp. parameters) defined in the implementation.

The challenge aspect of this project is introduced with the execution of the test cases. Each pair of students receives a different set of 20 implementations which contains 19 mutants and a correct program. Each mutant represent a wrong

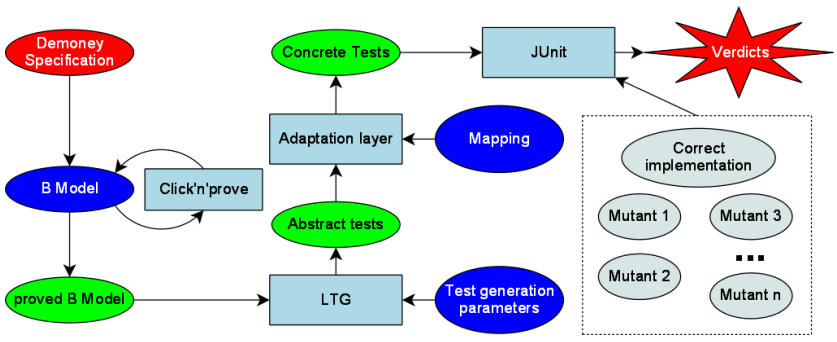

Fig. 4. Workflow of the project, from specification to test execution

implementation of one of the initial informal requirement. Thus, the students' objective is to kill (i.e. to detect) all the mutants and make all the tests pass on the correct program. This motivates them to design a model of the system which is both correct and complete.

3.2.2 Demoney: a Realistic System to be tested

The case study we propose to the student is based on the specification of *Demoney* (Demonstrative Electronic Purse) [MM] developed by *Trusted Logics* for research applications. No implementation of this specification is used in real life, but it is closed to credit card applets and thus, pertinent. Demoney is a good example of a critical system, a banking transaction, which require a lot of validation and verification, but it is also simple enough to be studied as a student project. Our specification describes a card with four life cycle states: *personalization*, *use*, *blocked* and *disabled*. The card is secured with two PIN objects –user and bank– which are associated to retry counters and values. Moreover, the card contains informations such as the current balance, the maximal balance and the maximal debit.

During the *personalization* phase, the PUT_DATA operation makes it possible to set the informations and the objects of the card. Then, the STORE_DATA operation terminates this phase if all the informations are setup and coherent with the security policy. Once the card has quit the personalization phase and started a use phase, it will never come back to the former state. During the *use* phase, the card holder can invoke some commands to gain authentication on PIN, using VERIFY_PIN, and to initialize a financial transaction, with INITIALIZE_TRANSACTION, that has to be completed by a call to the COMMIT_TRANSACTION command. If the user fails so much times to be authenticated that the retry counter of the user PIN reached zero then the card becomes *blocked*. When the card is *blocked*, the authenticated bank can unblock the user PIN and change its value. But if the card is blocked and the retry counter of the bank PIN reaches zero then the card become *disabled* and will never be used again. In addition, the implementation presents observation operations that can be used to retrieve the current balance, the maximal balance, the maximal debit and the authentication state of the PINs. An informal specification is given to the students that details the different behaviors of the system. They have to choose the most relevant design for their model so that interesting test cases are generated.

3.2.3 Model-based Testing of the Demoney Case Study

The students are asked to design a defensive model, that for each operation, considers its correct behavior and the potential errors. In order to be able to deal with multiple errors, the order in which the potential errors have to be checked is indicated in the specification document.

The interest of this specification is to provide some behaviors which require a non-trivial command chaining to be reached. It forces the students to target these behaviors and help the test generation engine. This specification raises the notion of life cycle which is important in security testing. Moreover, the specification contains important properties of the system. Some of these properties can be expressed as invariant properties, whereas some others, like operations chaining properties or

more generally temporal properties, must be respected by operations but can not be formalized as invariant properties. Thus, the students have to select what they are able to model as invariant properties or not. For instance, an invariant property over the system is: "If the card is not in the personalization phase then the balance can not be negative or higher than the maximal balance". At the opposite, the property: "All initialization of a transaction must be immediately followed by a transaction confirmation, otherwise the transaction is cancelled", can not be easily formalized as an invariant property. In this case, the use of the LTG animator can help the students to check that this property is correctly designed in their model.

Finally, in order to deal with the potential complexity of the model, that will slow down the computation time of the test cases, we require the students to produce two test campaigns. The first one is dedicated to the personalization phase, and the second considers a given personalization, that the students have to determine (and justify), so that these tests directly starts from the use phase of the card.

3.2.4 Feedbacks

The main objective of the approach, interesting students to formal methods by using a practical point of view, seems to be sucessfully fulfilled, according to the feedback they gave us on the course evaluation. Moreover the study of a smart card application was really interesting for them.

The challenge introduced by using some mutants maintained the students' interest until the end of the project, this is partially due to the fact that some mutants were very difficult to detect, thus, they spend the very last minutes to generate and run tests to kill the remaining mutants. If the use of faulty implementations to motivate the students and to evaluate their ability to produces quality tests, it also offer a good benchmark to them to evaluate their model and to help them to improve it, by using the correct model as a reference.

The tool chain that we have proposed worked perfectly and none of the students complained about bugs or strange features of the LTG tool. The animator was very useful in the early phases of model design. The test generator was used without any problems, even if some of the students had to perform many tries before understanding the proper use of some of the coverage criteria. Basically, the choice of the LTG tool was well-received by the students who appreciated the visual of the tool and the features it proposed.

4 Conclusion

We have presented in this paper the principles and the story of the BZ-Testing-Tools framework, initially developed in our research laboratory, and transferred into the industry. This tool consists of a model animator and a test generator that we use as a support for teaching formal methods to masters students. The maturity acquired by the tool makes a reliable tool-support for our students to learn the B notation, and to apply it into a realistic context. This year, we have designed a project [DT08] that makes it possible for the students to apply a complete model-based test generation approach, from the design of the model, to the execution and the analysis of the tests.

The design of the project offered a good support to practice formal methods in an application validation context. Furthermore, it provided an industrial-like model based testing approach to test the conformance of a system. The similarities between this class project and an industrial validation activity reside in both the method and the tools that were used. Moreover, the Demoney case study is very close to a "real" smart card application. Thus, all these facts motivated the students and their feedback on the project was excellent.

References

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[Abr96] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AC03] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In D.A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference (TPHOLs 2003)*, volume 2758, pages 1–24, Rome, Italy, 2003. Springer.

[BCC+05] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[Bei95] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[BJL+05] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. Requirement traceability in automated test generation - Application to smart card software validation. In *Procs. of the ICSE Int. Workshop on Advances in Model-Based Software Testing (A-MOST'05)*, St. Louis, USA, May 2005. ACM Press.

[BLP04] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, August 2004.

[CLP04] S. Colin, B. Legeard, and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[DT08] F. Dadeau and R. Tissot. Teaching Model-Based Testing with the Leirios Test Generator. In *Proceedings of the International Workshop on Formal Methods in Computer Science Eduction (FORMED'08)*, Budapest, Hungary, March 2008.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In J. Julliand and O. Kouchnarenko, editors, *7th International Conference of B Users (B'2007), Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *LNCS*, pages 277–280. Springer, 2007.

[LB03] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer Verlag.

[MM] R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse - card specification.

[OXL99] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5^{th} IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006. 550 pages.

[Utt00] Mark Utting. Data Structures for Z Testing Tools. In *4th workshop on Tools for System Design and Verification (FM-TOOLs 2000)*, Reisenburg Castle, Germany, July 2000.