A Methodology and a Framework for Model-Based Testing

Levi Lucio, Luis Pedro and Didier Buchs

University of Geneva, Software Modelling and Verification Group
24, Rue du Général Dufour
CH-1211 Genève 4, Switzerland

Abstract. In this paper we will present a survey on the test case generation process and tools we are currently developing. It will reflect the new ideas that we're pursuing while keeping in mind our previous work on formal specification languages and theory of test case generation.
The model based test case generation method we propose is based on a subset of the Unified Modelling Language (UML) and the Object Constraint Language (OCL). It uses UML diagrams in what concerns the conceptual point of view and, in addition, OCL expressions for the system's behavioral description.
The research builds on past experience of the group while generating test cases starting from a model of the SUT (System Under Test) described in the *CO-OPN* formalism - formal language for system specification that acts as an intermediary format between the model and the tests.
Our method makes use of well known techniques such as symbolic execution by means of a logic resolution engine (i.e. Prolog) for state space exploration of the SUT.

1 Introduction

Testing has always been recognized as a fundamental part of the software development process. Despite, during long years testing was considered as an "annoying" task which often was done in an ad-hoc fashion. Nowadays, modern software development teams recognize the need for systematic testing in order to produce quality software. Unit-level testing tools allows to begin testing early in the process and motivates developers to think ahead to what tests they want to see running.

But, unit-level testing is usually performed at the class level and it only validates a specific unit of code. Since a system can be composed of many classes that agglomerate into subsystems, integration tests are also necessary. At the system level, these tests will reflect use cases that are pertinent and that exercise the system such that a certain level of confidence in it's reliability can be gained.

In order to build integration tests development teams usually follow certain protocols to achieve a good coverage of the SUT (System Under Test). These protocols are however often informal and the integration tests are built by hand according to the knowledge of the test engineer. This produces coverages of the

SUT which are hardly qualifiable, much less quantifiable. The problem complexity grows exponentially with the size of the system.

On the other hand the theoretical field has already identified and sometimes solved many of the problems related to testing. See for example the work by Doong and Frankl in [2] or by Binder in [1]. However, the gap between theory and practice is still quite wide – only few tools for test case generation exist that make use of strong theories and are known/used in the software development arena. The goal of our research is to try to narrow this gap by wrapping the power of formal methods. In this paper we explain the methodology and the framework we are developing for test-case generation, which are fundamentally based on the work described in [3] by Gaudel and Marre and extended by us [7].

In order to make the description more pragmatic we have adopted a drink vending machine (DVM) example that we will use throughout the paper as an illustration of our approach.

The structure of the remaining text is as follows: first we'll provide the ***Motivation and Principles*** of our work; In second place, section ***Modelling the Application***, defines model-base testing and we describe the modelling language that we are using; The next section, ***Test Case Generation Machinery***, goes into the details and explains the process of generating test cases from a model of the SUT. We describe both the process and the tools that are already implemented or that we envisage for a near future; In ***Application of Test Cases*** we show how abstract test cases (sequences of operations) can be executed on a real application by means of a test driver;

2 Motivation and Principles

In a nutshell the goal of our work is to automate the generation of test cases as much as possible. To do that, we try to raise the level of abstraction at which humans have to intervene in the process. The idea is to capture the "intuition" of the tester about which aspects of the SUT should be tested, while leaving the job of translating those high-level intentions into concrete test cases to specialized machinery.

Since we use the model-based test case generation technique, an initial model of the application needs to be provided to the test case generation machinery. We have experience generating test cases from models expressed in the formal specification language *CO-OPN* (described by Biberstein, Buchs and Guelfi in [4]). However, since models of the SUTs have to be redone using *CO-OPN*, we found it difficult to apply to general software systems – concepts like abstract data types imply a construction of a new model for each SUT.

This said, we have decided to "wrap" our work using the Unified Modelling Language (UML) as the departure language for modelling the SUT – this will make our approach more general and easy to follow. Also, UML 2.0 includes OCL (Object Constraint Language) which allows more precise semantics than previous versions of this modelling language and enable us the possibility to use more than just constraints (where constrains stand for restriction on one or more values of an object oriented system) in order to better specify the system behavior - by means of OCL expressions in UML 2 it is possible, for example, to define queries.

Given a sufficiently expressive UML model of the SUT we then process it in order to generate test cases. The main problem with generating test cases for a software application is the fact that the exhaustive test set[1] is, in general, infinite. This is due to the fact that, for instance, a loop can be executed an infinite number of times or an input variable may have an infinite domain. We cope with this by introducing high-level test intentions, called *hypotheses*.

3 Modelling the Application

In this section we explain the concept of model test case generation. Furthermore, we discuss the *Fondue* [5] (object-oriented software development method that uses UML notations) modelling language we use to model the SUT and from were the test cases are generated.

3.1 Model-Based Test Case Generation

In model-based testing there are two main participants to the test process: the **specification model** and the **system under test (SUT)**. These two are connected by an *implementation relation* as shown in Fig. 1. For instance, based on the idea that the implementation has an observational behavior equivalent to the specification[7]. The idea behind it all is the following: in order to test a

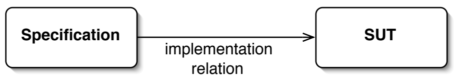

Fig. 1. Model-based test case generation

SUT specification model of it must exist. Sequences of pairs *(operation,result)* are generated from the specification model. These sequences and respective outputs, which we call *test cases*, can then be applied to the SUT. If the SUT reacts to the stimuli in the same way the model did, the test case is successful. If not, the *implementation relation* cannot be established and the SUT does not conform to the model.

[1] Exhaustive meaning all the possible test cases over an SUT.

3.2 The Drink Vending Machine (DVM) Example

Throughout the paper we will often make reference to a drink vending machine (DVM) example as means of exemplifying our ideas. In the next lines we briefly state the DVM problem.

The controller for the DVM is supposed to coordinate the activities of the several components of the machine:

- *money box*: divided into two parts – the money collector that keeps the coins inserted to buy one drink; the money repository that holds all the coins gathered for selling drinks;
- *drink shelves*: there may exist several shelves, each one of them holding a particular kind of drink;
- *drink selection buttons*: one for each available drink type;
- *return coins button*: to abort a buy operation and return the already inserted coins;
- *display*: to display the amount of money already inserted.

A typical use case of the DVM would be the following: a client inserts a number of coins in the DVM and presses a drink selection button. The DVM distributes the chosen drink.

3.3 *Fondue* as Modelling Language

Since UML encompasses many different views over the same model, we have decided to restrict the number of these views we are effectively using. In particular, we have adopted the specification language of a methodology to specifying reactive system behavior. This approach called *Fondue* has been developed at the EPFL[5].

In terms of model specification, Fondue provides two main artifacts in order to describe the problem domain and the functional requirements of the system: *Concept* and *Behavior* Models. The first one is represented as *UML class diagrams* and defines the static structure of the system information. The *Behavior Model* defines the input and output communication of the system, and is divided in three models: *Environment*, *Protocol* and *Operation* - represented respectively by *UML collaboration diagrams*, *UML state diagrams* and *OCL operations*.

The following items provide a small description of each one of the Fondue Models illustrates them using the DVM example.

Environment Model: the environment model describes the interaction between the system and its environment. All possible inputs and outputs of the system are made explicit. Figure 2 depicts the environment model for the DVM.

Concept Model: To define the system's state the *Fondue* approach uses the Concept Model. This model allows the expression of concepts of the problem's domain in terms of *classes* and *associations*. The populations of instances of those classes and associations define the system's state at a given moment. Figure 3 shows the concept model for the DVM.

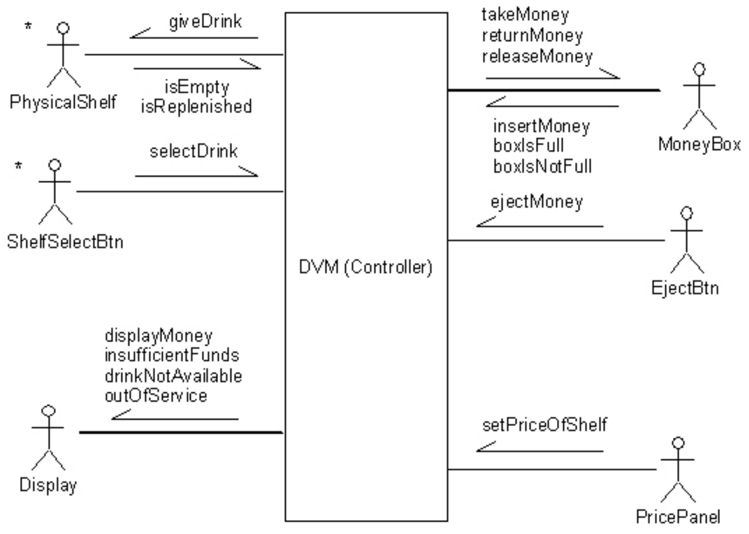

Fig. 2. Environment model for the DVM

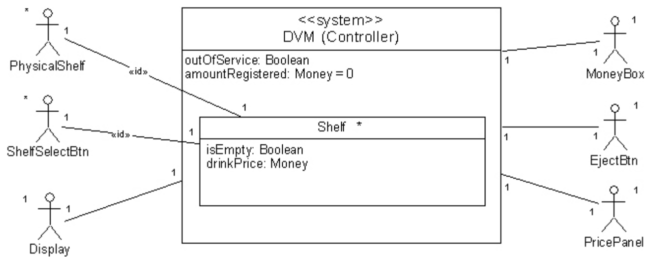

Fig. 3. Concept model for the DVM

- **Protocol Model:** with the Protocol model it is possible to specify the dynamic behavior of the system over logical time. This model is expressed by means of state machines which capture the way the system responds to requests depending on its current state.
- **Operation Model:** the Operation model defines the requests that the system is able to answer and how these effect on the system's state. Besides defining the parameters of an operation, this model defines its pre- and post-condition. The pre-condition describes assumptions about the state of the system before the operation is executed. The post-condition describes how the state of the system evolves after the operation is executed and which output messages are produced. The next example describes an operation for the DVM:

```
Operation: DVM::insertMoney (m: Money);
Description:
```

```
Use Cases: buy drink;
Aliases:
Messages: Display::{DisplayMoney; DrinkNotAvailable; InsufficientFunds};
                  MoneyBox::{ReleaseMoney;};
Pre:
Post:
   self.display^drinkNotAvailable(false) &
   self.display^insufficientFunds(false) &
   if not self.outOfService then -- condition
                                 ensures that money box is not full.
      self.amountRegistered = self.amountRegistered@pre + m &
      3self.display^displayMoney(self.amountRegistered)
   else
        self.moneyBox^releaseMoney()
   endif;
```

4 Test Case Generation Machinery

In this section we will describe our approach to the generation of test cases from a *Fondue* model. We start by giving an overview picture of the process we are putting in place and in a second phase we focus on each of the individual parts of that process.

We can define *Test Case* as a pair composed of a sequence of requests on the SUT/expected outputs, and a verdict. The verdict belongs to the set $\{true, false\}$ and reflects the validity of the sequence of requests against the expected behavior of the SUT, as defined in the specification.

This definition allow us to search for test cases that should be both accepted and not accepted by the SUT. The implementation relation requires strict respect of the model behavior.

4.1 The Test Case Generation Framework

The activity of test case generation can be seen as the process of finding a set of test cases which is pertinent and finite. Pertinence means that: the test set will discard a SUT if it does not fulfill the expectations; the test set will never reject a correct SUT.

Having in mind that the application under test may allow infinite sequences of events or an input value may belong to an infinite domain, the process we are putting in place needs to provide the tools to reduce the initial test set to a finite one that is pertinent.

In Fig. 4 we depict the full picture of the process of test case generation we are investigating. In the figure the arrows represent activities and the ellipses/circles represent artifacts involved in the process. The process is divided in two different parts: the left side of the figure represents the test case generation process; the right side, their application, results and the various components that are used. In this subsection we will focus on the part of the process that concerns how the tests are generated - this means that all references to Fig. 4 are related to its left side.

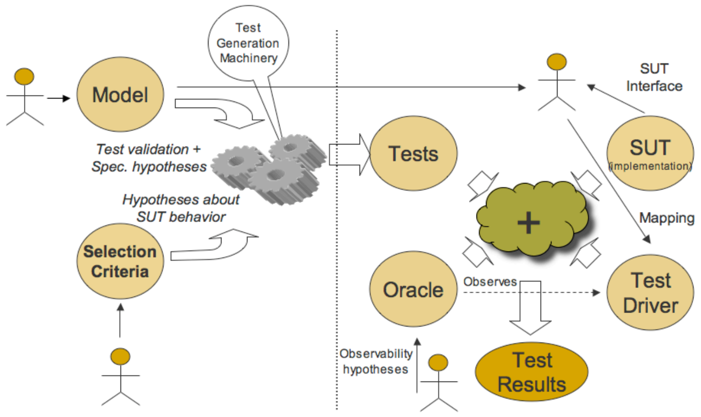

Fig. 4. Test Case generation process

The bent arrow coming from the the Model denote the activities of deriving from the specification: the *signatures*[2] of the possible operations over the SUT and a *behavioral model* of the SUT. During the test case generation process the operation signatures are organized in sequences that form the test cases. On the other hand, it becomes necessary to validate those sequences, i.e. to find the verdict for the test case. The *behavioral model* is an executable model derived from the specification that allows the automatic manipulation of an abstract state space of the SUT.

The process we describe in Fig. 4 is not fully automatic. The bottom left matchstick man represents human interactions while providing heuristics that reduce the initial infinite test set. These heuristics are provided as hypotheses over the shape of the test cases to be generated. These hypotheses are based on temporal logic formulas that allow us to express test intentions for COOPN specifications. The input/output pairs mentioned earlier in this paper, are expressed by the temporal logic formulas which are the main basis of the test/constraint language that is under development by our group. In general, we encapsulate the temporal logic using a language of constraints and the graphs of input/output pairs that they represent correspond to: operations performed on the system - input; observable results of the operations - outputs. The hypotheses (bottom bent arrow in the figure) may be provided either as patterns over the sequence of operations – path hypotheses; or as constraints over the parameters of those operations – value hypotheses.

[2] The signature of an operation is composed of the operation name and the name and type of each of the operation's parameters.

The remaining part of the process that we are analyzing has to do with the iterative refinement of the test cases: the test engineer expresses hypotheses about the functioning of the SUT that narrow the initial infinite test set. If the produced test set is not yet the expected one, another more refined iteration of the process is done.

4.2 Deriving the Behavioral Model

The behavioral model is the component that allows the automatic exploration of the state space derivable from the specification of the SUT. However, the step of turning a *Fondue* model into a sort of high level prototype of the final application is not trivial.

We split our test case generation research project in two main areas: **model treatment** – meaning extracting from the *Fondue* model a functional behavioral model – and **generating test cases** using that behavioral model. In between, we have an *intermediate format*: representation of the SUT using *CO-OPN*. The reason why we have chosen *CO-OPN* is the fact that we have already a translator that transforms it into Prolog. The Prolog result of this transformation can then be used to explore operationally the behavior of the specification. This means that the UML model has to be interpreted into a *CO-OPN* specification and that the test case generation engine will pickup from there. We have built a Prolog runtime engine (described by Buffo and Buchs in [6]) *CO-OPN* that allows us to manipulate automatically the state space produced by the UML specification.

The runtime engine is able to simulate the evolution in the state space of the model as operations are applied to it. In fact, we are able to represent in Prolog both the model itself – in the *CO-OPN* – and the state space produced by it. Since *Fondue* is an object-oriented approach, the runtime engine has to know how to deal with such concepts as: object management, polymorphism, navigation through associations between classes and concurrency[3]. The next example shows (a part of) the concept diagram of the DVM (in Fig. 3) in our intermediate format:

```
Class DVMController;
Interface
    Methods
        put _ outOfService : Boolean;
        get _ amountRegistered : Money;
Body
    Places
        outOfService : Boolean;
        amountRegistered : Money = 0;
End DVMController;
```

[3] The work on concurrency is inspired by our research on expressing the operational semantics of *CO-OPN*.

```
Class DMVShelf;
Interface
    Methods
        isEmpty : Boolean;
        drinkPrice : Money;
End DMVShelf;

Class ControllerShelfAssociation;
Morphism
    DVMControllerInstance -> DVMShelfInstalce;
End ControllerShelfAssociation;
```

The two classes in the system – DVM and Shelf – are defined as well as the composition association between them.

The behavior of the Fondue operations is translated into pre- and post conditions of the intermediate format. Meta rules defined in the runtime kernel provides rules for: embedding local states of each pre and post condition into the whole object system; managing the synchronizations between sub-systems; compute concurrent behaviors as well as sequence of operations. These meta-rules are inspired by the transactional and concurrent semantics of the *CO-OPN* formalism. In the future we expect to be able to deal, using all *CO-OPN* features, with composition of Fondue subsystems and with concurrency related behaviors. A *pre* or *post* condition has the form *IFprepost(event,pre,post) :- body* where:

event = with(operation,emitted events) is the behaviour of the operation *operation* described for a given structure of events emitted (*emited events*). The emitted events can be structured with sequential, non-determinism and simultaneity operators. Creation operators can also be included into this event structure;

pre and post describe the states before and after the operation occurs.

body is a collection of constraint that must be fulfilled by the variables that can appear into the *event*, *pre* and *post* expressions.

We use the runtime engine to: verify whether a concrete test case (in other words, an execution trace) exists[4] in the state space of the model; find test cases that obey to certain criteria (e.g. if the criteria is *true*, then Prolog would return all successful test cases, meaning all the possible execution traces of the specification).

Although the runtime engine for the intermediate format already exists and the intermediate format itself is nearly stable, all the interpretation machinery that will convert the *Fondue* model into a *CO-OPN* model is under development. We are counting on doing this by expressing the model in a tool for representing

[4] Prolog replies *yes* or *no* depending on whether the test case is executable or not.

UML/Fondue diagrams[5]. The tool provides the functionality that allow us to export the diagrams using the XML Meta Data Interchange (XMI). This XMI can then be loaded by any implementation of the Meta Data Repository (MDR) and the navigation within the exported model is enabled by means of Java Metadata Interface (JMI). A set of well defined transformation rules provide the functionality to map a Fondue model into a COOPN one. This part of the work is being developed at the time that this paper is being written and the general approach is to use Model Driven Architecture (MDA) technics and Meta-Model levels both to concretize a transformation definitions between the two models. It is also our objective to use standard approaches to implement a transformation tool that will allow to systematize and automatize this part of the Test Generation framework.

4.3 Applying hypotheses

To express the heuristics that help excluding from the final test set the ones that aren't relevant, we have defined a language to express *hypotheses* about the functioning of the SUT. The rationale behind the approach is that the stronger the hypotheses, the more important the reduction in the exhaustive test set. We have built a theory[7] behind the concept of applying hypotheses that reduce the exhaustive test set without losing pertinence.

As was previously mentioned in the paper and shown in Fig. 4 there are two main variables over which it is possible to express hypotheses: *paths*, meaning the form of the sequences of operations; *values*, meaning the parameters of those operations. In what concerns *paths*, the hypotheses are given by regular expressions that allow the definition of sequences of operations. The language makes use of the operators: $*$ for zero or more repetitions; $+$ and one or more repetitions; AND and OR for connecting sequences of operations. For convenience we have also defined a $lowerT$ operator that bounds the upper limit of repetitions of an operation. As an example assume we have operations x and y in our SUT. With the hypotheses *x*y OR x*, possible test cases would be yx, xy or $xxxyx$. In what concerns *values*, we can apply two types of hypotheses:

Uniformity: if a test case containing a variable v is valid for one value of v, then it is valid for all of v's domain. This type of hypotheses can be compared to random testing;

Regularity:: if a test case containing a variable v is valid for a subdomain of v satisfying a given complexity criteria, then it is valid for all of v's subdomains of greater complexity.

Uniformity hypotheses can be seen as a particular case of the regularity hypotheses where the subdomain under test only contains one element.

The next example shows a Prolog goal that calculates a test set for the DVM:

[5] Any CASE tool that could provide the possibility to endorse a Fondue Module can be used. For the moment, we're using a module of Fondue for Together (http://www.borland.com/together/)

```
solve([pattern(_,and(star(ev(insertMoney(1),_),N),ev(selectDrink(S),_)),L),
lowerT(nat,N,5)]),uniform(L),valid(L,true).
```

The goal can be splited into three parts:

1. prefixed by the predicate *solve*[6] states an hypotheses over the *path* that reduces the focus to sequences of five *insertMoney* operations followed by one *selectDrink* operation[7];
2. prefixed by the predicate *uniform* chooses randomly one value for all the variables that remain uninstantiated in the test set. In this case the only uninstantiated variable is the type of drink (denoted by S).
3. prefixed by the predicate *valid* validates the previously obtained test sets against the behavioral model in order to find their verdict. In this case we are only interested in tests that are accepted stated here by the *true* parameter.

Since the mechanism for automatically generating the behavioral model is not implemented, we have coded by hand a behavioral model of the DVM. This allowed us to run the Prolog goal stated above and to obtain the test set:

```
with(selectDrink(Water),[insufficientFunds])
with(insertMoney(1),[]), with(selectDrink(Fanta),[insufficientFunds])
with(insertMoney(1),[]), with(insertMoney(1),[]), with(selectDrink(Fanta),
   [giveDrink(Fanta)])
```

In fact, five test cases are generated by the engine but due to their size we only present three. It is possible to see that the mechanism generates sequences of operations along with their expected outputs.

4.4 Subdomain Decomposition

While discussing the application of *uniformity hypotheses* we mentioned that it can be compared to random testing. In fact, applying uniformity over a variable with a given domain will yield a single value from that domain, picked at chance. Although this hypotheses is useful, it can be refined.

Consider for example that in the application of the hypotheses in Sec. 4.3, Pag. 11, we have not a sequence of *insertMoney(1)* operations but rather one single *insertMoney(N)* operation – where N represents the number of coins to insert – followed by the *selectDrink(S)* one. In this case, a uniformity hypotheses applied over N and S would produce one single test case with N and S instantiated to random values. The interest of this test set is arguable.

To improve the situation, one can reason about the interesting values that N can assume. There are three situations that can be induced by values of N:

[6] The *solve* predicate is related to the fact that we have substituted Prolog's SLD by another resolution mechanism to better fit our needs

[7] In our Prolog representation, the $*$ operator is expressed by the predicate *star* and the $+$ operator by the predicate *plus*.

– if $N \in \{0, .., price\,of\,drink\,S - 1\}$ drink S is not distributed;
– if $N \in \{price\,of\,drink\,S\}$ drink S is distributed;
– if $N \in \{price\,of\,drink\,S + 1..\infty\}$ drink S is distributed and the excess money is given back to the client;

Ideally, a uniformity hypotheses over N would produce not only one but three values, corresponding to each one of the three situations stated above. This can be done by searching the behavioral model of the specification symbolically in the sense that the points of decision are explored for both $true$ and $false$ conditions. At the end of this exploration, each possible execution path will be represented by a set of constraints on variables (operation parameters) produced by the accumulation of conditions at each of the points of decision. The final step will be to instantiate the variables participating in the set of constraints that denotes each execution path. The implementation of this mechanism is done in Prolog with resolution stopping at the decision points.

5 Application of Test Cases

Finally, it is necessary to apply to the SUT the tests produced by the process described in the previous section. In Fig. 4 this step is expressed on the right side of the picture. As can be seen, a piece of machinery called the *test driver* is needed in order to apply the sequence of operations to the SUT must to observe if the outputs correspond to the expected ones. The top matchstick man acts as an intermediary between the Model, SUT interface and the test driver. The signatures of the operations are derived from the Model and, since there is a relation of 1 to 1 between the specification and the SUT connect, we can then interface the *test driver* to the SUT interface by means of mapping rules.

Coupled with the test driver is the *oracle*, which is a decision procedure that decides whether a test case should pass or fail. The user (bottom matchstick man) provides observation hypotheses that will be used by the *oracle* to decide wether the test is accepted or not. A test case accepted by the model should be accepted by the SUT and vice-versa for one that is not accepted. If this is the case the SUT passes the test, otherwise an error is detected.

We have implemented a very simple DVM on the web that allows all the main interactions described by the problem statement in Sect. 3.2. The user is capable of (virtually) buying a drink by clicking on an *insert money* link a number of times and then clicking a link corresponding to name of the drink to choose it. We have put online two DVMs, one functioning correctly and one that takes the money but does not distribute the drink.

In order to find the error we applied the test cases that were found in Sect. 4.3. To apply the tests we have used a trial version of a tool called Astra QuickTest^{TM}. The tool provides facilities to test web pages by providing unitary actions that can be assembled into test cases. A small translator script was written to pass from the abstract test case format (see **??**) to the language of Astra QuickTest^{TM}. An example of a test case translated into the driver's language is shown in the following lines:

```
Browser("The Drink Vending").Page("The Drink Vending").Link("Reset DVM").Click
Browser("The Drink Vending").Page("The Drink Vending").Link("Insert 1 coin").Click
Browser("The Drink Vending").Page("The Drink Vending").Link("Insert 1 coin").Click
Browser("The Drink Vending").Page("The Drink Vending").Link("Choose 1 Fanta").Click
Browser("The Drink Vending").Page("The Drink Vending").Check CheckPoint("Heres your")
```

In this specific case the test driver tool is coupled with the *oracle* since Astra QuickTest^{TM} decides automatically whether a test case passes or not.

6 Related Work

A large number of papers on model-based test case generation exists in the literature. However, not many deal with models expressed in semi-formal languages such as UML.

At the university of Franche-Comté an approach to test case generation similar to ours is being developed. Legeard and Peureux explain in [8] their method which consists in: translating a UML specification into a program in an adapted logic programming language similar to Prolog; explore symbolically the state space of the model searching for values for parameters of operations that are interesting to test. The procedure is an evolution of what we described in Sect. 4.4.

Pretschner et al explain in [9] their approach which starts from a model described in AUTOFOCUS^{TM}, a tool based on UML-RT (for Real-Time systems). The framework also makes use of a logic programming language to explore symbolically the state space.

7 Future Work

This paper describes work that is in progress. In the agenda for the next steps the coding of the algorithms that will translate a *Fondue* model into the *CO-OPN* and the operational implementation of full sub-domain decomposition. At the same time we are re-evaluating our hypotheses language with the goal of specializing it – a catalog of domain-dependent hypotheses is something we are looking into – and also to develop a graphical user interface for the expression of hypotheses based in a formalized test/constraint language.

8 Acknowledgements

The present work is part of the VeDiSS project which is partially funded by HaslerStiftung, DICS initiative.

References

1. Binder, R. V.: Testing object-oriented software: a survey. Journal of Testing, Verification and Reliability, 6:125252, 1996.

2. Doong, R.-K. and Frankl, P. G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology, 3(2):101130, 1994.
3. Bernot, G., Gaudel, M.-C., and Marre, B.: Software testing based on formal specifications: a theory and a tool. IEEE Software Engineering Journal, 6(6):387405, 1991.
4. Biberstein, O., Buchs, D. and Guelfi, N.: Object-oriented nets with algebraic specifications: The *CO-OPN/*2 formalism. In G. Agha and F. De Cindio, editors, *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
5. Strohmeier, A.: Fondue: An Object-Oriented Development Method based on the UML Notation. In X Jornada Técnica de Ada-Spain, Documentación, Lunes 12 de Noviembre 2001, ETSI de Telecommunicación, Universidad Politécnica de Madrid, Madrid, Spain, November 2001.
6. Buffo, M. and Buchs, D.: Symbolic simulation of coordinated algebraic petri nets using logic programming. To be published: internal note, University of Geneva, 2004.
7. Péraire, C., Barbey, S. and Buchs D.: Test selection for object-oriented software based on formal specifications. In Proc. of Programming Concepts and Methods (PROCOMET) 98, pages 385-403, 1998.
8. Legeard, B. and Peureux, F.: Génération de séquences de tests à partir d'une spécification B en PLC ensembliste. In Proc. Approches Formelles dans l'Assistance au Développement de Logiciels, pages 113-130, June 2001.
9. Pretschner, A. et al: Model-based test case generation for smart cards. In Proc. Formal Methods for Industria Critical Systems, 2003.