Formal Test Generation from UML Models*

Didier Buchs, Luis Pedro, Levi Lúcio

Software Modeling and Verification laboratory
University of Geneva, Switzerland,
didier.buchs,luis.pedro,levi.lucio@cui.unige.ch,
WWW home page: http://smv.unige.ch

Abstract. In this paper we will explain our approach for generating test cases for a UML system model. Despite the fact that UML authors claim that UML semantics are precise enough to define non-ambiguous models, we find that the overlap of the different views makes it difficult to explore and make deduction on the state space of the modeled system in order to generate test cases. Our approach is thus based on a subset of UML (inspired from the Fondue approach) for which we have defined clear transformation semantics. We provide these semantics by delineating transformation rules using MDA (Model Driven Architecture) architecture as foundation. We transform UML models into CO-OPN (Concurrent Object Oriented Petri Nets) ones, CO-OPN being a formal specification language defined in our Laboratory.
We have also defined a language for expressing test intentions for CO-OPN models. This language allows selecting interesting executions (tests cases) of a model by providing constraints over all possible traces of that model. By exploring the model's semantics with the tools we have built for our CO-OPN language we are able to generate test cases based on those test intentions. We are also able to partially eliminate redundancy in the produced test cases by finding equivalence classes in the model operation's inputs.

1 Introduction

As the complexity and size of a system increases, modeling techniques that address abstraction and decomposition play a very important role. At the same time different view points of the system are envisaged in order to fully describe it by means of a model. This raises the problem of relating and keeping consistent the different models that represent, sometimes orthogonally, several perspectives over the system. At the same time, the need to isolate errors in the implementation motivates our work that aims of automatically generating test sets from a well defined model. Figure 1 shows the general picture of our approach that will be explained in detail during this article.

* This project was partially funded by the DICS project of the Hasler foundation, with number 1850.

Our development approach encompasses three steps: *Analysis − Prototyping − Implementation*. For the analysis phase we use the UML Fondue[1] development method that allows specifying a system using different view points by means of different diagrams - at the same time Fondue allows producing a complete description of the system by providing a logical relation between each individual diagram.

The product of the implementation phase is a system that will be used to execute the tests produced by the test case generation framework. In our Model-based testing approach, there is an implementation relation between the model (developed during the *Analysis* phase) and the System Under Test (SUT) based on the idea that the observational behavior of both model and implementation are compatible.

Taking into account that the objective of our current work is to automatically generate a set of tests that will afterwards be applied to the SUT, we need to transform the system specification into the language that we use for the purpose of test case generation - CO-OPN [2] (Concurrent Object Oriented Petri Nets).

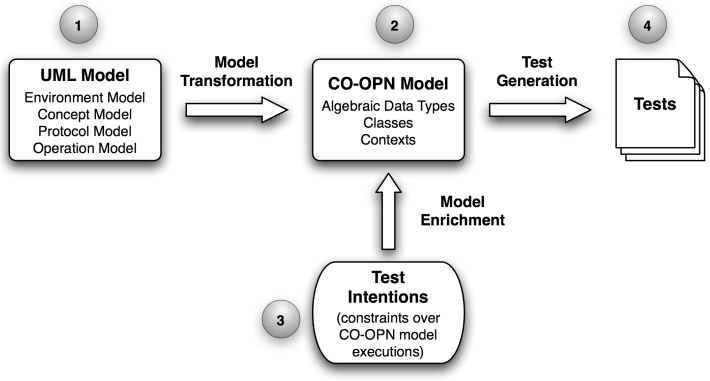

Fig. 1. The process of test generation from UML

In order to produce test sets from the system's model, we developed a language (*TestSel*) that allows expressing test intentions for the system specification. This language provides the syntax and semantics that permits narrowing the initial (usually infinite) number of tests present in a system specification.

2 From UML to CO-OPN

Our approach aims at easily generating tests from a well known and widely used modeling language. One of the key points to achieve this is being able to transform it into a specification language that provides an unambiguous representation of the system. In this section we will focus on the activities that allow

achieving point *2* from point *1* in figure 1, i.e. UML models creation and its transformation into CO-OPN. Before we continue detailing how this process is accomplished, we will briefly introduce the two specification languages: Fondue [1] (UML Dialect) - the source language; and CO-OPN [3] the target language.

UML Fondue provides two main artifacts: *Concept* and *Behavior* Models. The first one is represented as *UML class diagrams* and defines the static structure of the system. 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 charts* and *OCL operations*.

CO-OPN is a formal specification language built to allow the expression of models of complex concurrent systems. Its semantics is formally defined in [4], making it a precise tool not only for modeling, but, thanks to its operational semantics, also for prototyping and test generation. It groups a set of object-based concepts such as the notion of class (and object), concurrency, sub-typing and inheritance that we use to define the system specification coherently regarding notions used by other standard modeling approaches. An additional coordination layer provides the designer with an abstract way of representing interaction between modeling entities and an abstract mapping to distributed computations.

The CO-OPN object oriented modeling language is based on Algebraic Data Types (ADT) and Petri Nets. It provides a syntax and semantics that allow using heterogeneous Object Oriented (OO) concepts for system specification. The specifications are collections of ADTs, *classes* and *context* - the CO-OPN modules. Detailed information about COOPN language can be found in [2].

2.1 The Model By Example

This section will explain, by means of an example, how we use UML Fondue in order to model our system. We start presenting the case study by stating the problem description and we continue defining the model using the Fondue methodology. The example is not intended to be a complete and exhaustive one, but rather an illustration of the full process for our approach.

The proposed system consists of a mobile phone with a SIM card that can be authenticated with a PIN number. The phone has three different functioning states: phoneOff - when the phone is not operating; phoneStandBy - when the mobile phone is waiting for the user to insert a PIN number; phoneOn - corresponding to the state where the phone is ready to perform calls.

The behavior of the system is such that the user is asked for a PIN number in order to be able to turn the phone to the phoneOn state that allows performing calls. The inserted PIN (stored in the SIM card) is checked and it can only be wrong a maximum number of three consecutive attempts. If this number is reached the card's state changes from unlockedPin to lockedPin.

Environment Model: This model precises the incoming and outgoing messages of the system. In the figure that shows the *Environment Model* for this

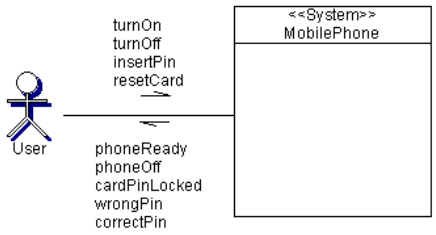

Fig. 2. Fondue Environment Model for the Mobile Phone System

system (figure 2) it is possible to observe that the possible incoming messages are: turnOn, turnOff, insertPin and resetCard. The last one is used for demonstration purposes just in case the SIM card is locked after the maximum number failed insertPin operations. The system is able to send to the user the following messages with obvious meanings: phoneReady; phoneOff; cardPinLocked; wrongPin; and coorectPin

Concept Model: The static structure of the system is accomplished by the realization of the *Concept Model*. This structure is defined as an UML class diagram and it is presented in figure 3 for our example system.

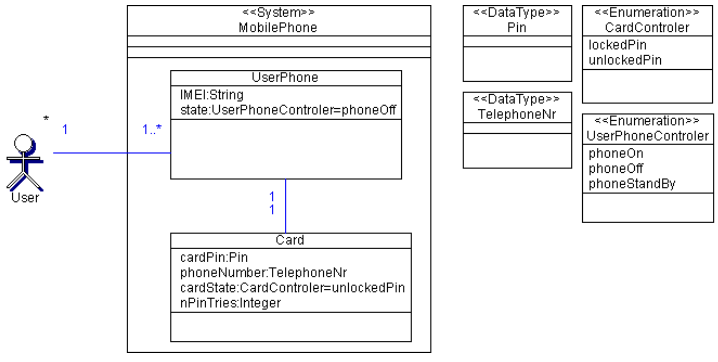

Fig. 3. Fondue Concept Model for the Mobile Phone System

We consider the system composed by two classes related as presented in the figure. The diagram shows a system in which one user can have one or more mobile phones (represented by the association User ⟶ UserPhone). On its turn, the UserPhone is identified by its IMEI[1] and by its state (state attribute in UserPhone class) and provides an association of cardinality 1..1 with class Card. The SIM card (class Card in the figure) is identified by phoneNumber, cardPin, cardState and nPinTries representing respectively: the phone number associated to the card; its valid Pin; the card's state; the number of previous Pin insertions with a wrong value. Both state and cardState attributes are defined by the CardControler and UserPhoneControler enumerators. These enumerators provide the allowed states of the two transition systems that will be further specified by the two *Protocol Diagrams*.

Protocol Model With Fondue's *Protocol Model* it is possible to specify the dynamic behavior of the system over logical time. This model is expressed by means of UML state charts which capture the way the system responds to requests depending on its current state.

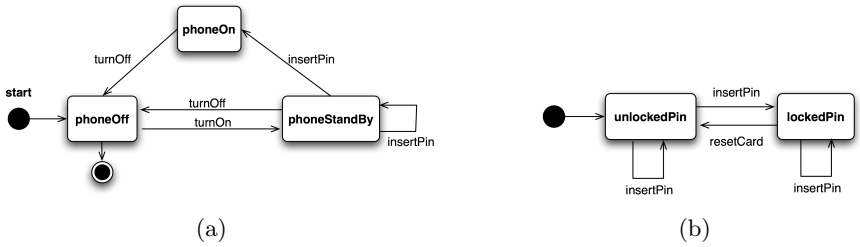

Fig. 4. Fondue Protocol Models - (a): UserPhoneControler state machine; (b): CardControler state machine

As can be seen in figure 4, some of the actions (in the same system state) can lead to more than one system state - this presents a typical example of unwanted non-determinism in the system's specification. The non-determinism can be suppressed in the *Operation Schemas* by means of OCL constraints expressed using pre and post-conditions.

Operation Schemas: Describe the services offered by the system. For simplicity, in this section we are going to present only one of the Operation Schemas: the operation insertPin(pin:Pin) that is presented in 5.

[1] International Mobile Equipment Identity Number (IMEI) is an unique electronic serial number of the Global System for Mobile Communication (GSM) mobile phone handsets

```
Operation: MobilePhone:insertPin(pin:Pin)
Description: This operation describes how the system behaves during the Pin insertion
process.
Messages: User::{phoneReady; wrongPin; correctPin; cardPinLocked}
New: c: Card;
Alias: u:UserPhone IS sender.UserPhone
Pre: u.state::phoneStandBy and
     not u.state::phoneOn and
Post: if c.cardState::lockedPin then
          sender^cardPinLocked
      elseif c.cardPin <> pin
          c.nPinTries = c.nPinTries@pre + 1 and
          sender^wrongPin
          if c.nPinTries = 3 then
              c.CardStatus::lockedPin and
              c.nPinTries = 0
          end
      else
          c.nPinTries = 0 and
          u.state::phoneOn and
          c.cardState::unLockedPin and
          sender^correctPin and
          sender^phoneReady
      end
```

Fig. 5. Fondue Operation Schema for operation insertPin(pin:Pin) for the Mobile Phone System

This operation describes the behavior that the system should have when the message insertPin is sent. Albeit the sequence of allowed transitions has already been defined in the *Protocol Model*, this definition identifies possible points of non-determinism in system's state phoneReady and provides the necessary logic in order to solve them.

2.2 Transformation Process

In order to perform the transformation from Fondue to CO-OPN we need to clearly define the methodology both in terms of technology used and in what concerns the formal definitions: Regarding the former one, our approach proposes that we use technologies that use MDA [5] as base framework. This implies using the *metamodel* of both languages and defining the transformation based on them; The *metamodel* of a language is a description of all the concepts hat can be used in that language - it is also known as its *abstract syntax*. A *metamodel* is composed of *metaclasses* and their *relationships* - in conjunction they compose the complete *metamodel* of the language. Thus, every element in an ordinary language is an instance of the respective *metaclass*. *Metamodels* are models and this implies that we can manipulate them in the same was - they just reside in another level of abstraction; The second aspect implies a clear and formal definition of transformation rules and mapping. This definition must be formally expressed and mapped into a transformation language (see for example [6] or [7]). The transformation language is the artifact that will allow executing the transformation and will act as the bridge between the technology used and the transformation definition. Transformations are composed of a series of

rules which are applied to the source Fondue model. Each rule attempts to find some pattern in the source model and, if successful, generate some corresponding pattern in an target CO-OPN model. One can see the transformation rule as consisting of two parts of a graph: a left-hand side (LHS); and a right-hand side (RHS).

These two aspects together can precisely define and execute a transformation from a *Language A* to a *Language B* - In particular we are interested in the transformation from UML Fondue to CO-OPN that we are going to detail in the remaining parts of this section.

The sequence of the transformation is presented in figure 6 using a Petri Net. The places with a token represent the set of different types of Fondue Diagrams. When firing transitions $T1$ to $T4$ in the figure, the transformation will evolve transforming, step by step, each one of the Fondue diagrams. Each transition

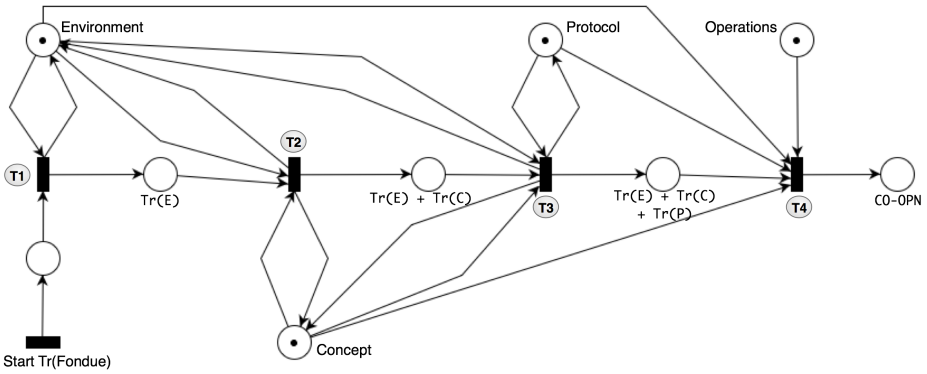

Fig. 6. UML Fondue to CO-OPN transformation sequence

represents the process of transforming one diagram and the place after it contains the result of that process together with the result of the previous transformation. Albeit each transition is meant to represent the transformation of a specific type of Fondue diagram, with the exception of $T1$ all the others need information from other diagram(s) besides the one that the transition concerns. More specifically:

- Transition $T2$ (transformation of the *Concept Model*) includes two distinct tasks: transformation of the *Concept Model* itself: and a second iteration in the *Environment Model* transformation to complement the transformation of the its input messages with their parameters. The *Environment Model* provides the input messages to the system but no information is specified in what concerns the possible parameters for each message.
- Transition $T3$ (*Protocol Model's* transformation) needs to have available the *Environment Model* information so that it can check if the system transitions specified correspond to messages previously defined in the *Environ-*

ment Model. At the same time, information regarding the *Concept Model* is also required to inspect if the names of the *Protocol Models* have any class/enumerator with the same name in the *Concept Model* and if the states defined in the *Protocol Diagram* correspond to the ones provided by the *Concept Model*.
- Transition $T4$ (*Operation Schema* transformation process) requires information from all the other Fondue models: from the *Environment Model* in order to understand if the name of the Operation was already defined; from the *Concept Model* to analyze if the invoked methods (and the classes that correspond to the type of the object) have been defined; from the *Protocol Model* to control if the states defined in the pre and post-conditions have been defined.

2.3 Formalization of the Transformation

The transformation process from Fondue to CO-OPN consists in the composition of transformation of each one of the Fondue models. A transformation from Fondue to CO-OPN is a function:

$$\forall F \in Fondue, \exists C \in COOPN : T_r(F) = C \tag{1}$$

At the same time, the transformation $T_r(F)$ is a composition of the transformation of each one of the Fondue models:

$$\forall F =< e, c, p, o >\in Fondue, e \in E, c \in C, p \in P, o \in O :$$
$$T_r(F) = T_r^{env}(e) + T_r^{con}(c) + T_r^{prot}(p) + T_r^{op}(o) \tag{2}$$

with, E the set of *Fondue Environment* diagrams, C the set of *Fondue Concept* diagrams, P the set of *Fondue Protocol* diagrams and O the set of *Fondue Operation Schemas*. The '+' operator is the disjoint union.

Defined that we have the generically the transformation from Fondue to CO-OPN, in the following lines we will particularize how the transformation from each type of Fondue diagram is defined.

Environment diagram: This transformation constitutes the first iteration in order to achieve the complete transformation from Fondue to CO-OPN. The *Environment Diagram* is composed of one System, messages going to the system and messages sent by the system to the outside - as presented in figure 2. Being S, M_i, M_o the System, the set of input messages and the set output messages respectively we can formalize the transformation of a Fondue Environment diagram as:

$$\forall s \in S, m_i \in M_i, m_o \in M_o :$$
$$T_r^{env}(E) = T_r^{env}(s) + T_r^{env}(m_i) + T_r^{env}(m_o) \tag{3}$$

Taking into account that one system is transformed in a CO-OPN Context, the input messages into methods of the Context and the output messages into gates of the CO-OPN Context, and being Γ the set of CO-OPN Contexts, M the set of CO-OPN Methods and G the set of CO-OPN gates:

$$\forall s \in S, m_i \in M_i, m_o \in M_o : \\ T_r^{env}(E) = Id(co) + Id(m) + Id(g) \quad (4)$$

where $Id : transformation_new.tex, v1.102005/10/0614 : 05 : 12 pedroExp$ is the isomorphic transformation between Fondue and CO-OPN.

Concept Model: The transformation of the *Concept Model* is basically the transformation of a reduced UML class model.

Generically, the transformation of the *Concept Model* is performed as follows:

- the class name in the *Concept Model* is transformed as the name of a class in CO-OPN;
- the attributes are transformed in Places[2] in the CO-OPN class;
- the class associations are CO-OPN classes with source and target values;
- get and set methods must be created in order to access and modify each one of the class attributes;
- both the user defined and primitive data types are transformed in CO-OPN ADTs.

Taking this into account, for class Card in the *Concept Model* from figure 3, the result of the transformation of a CO-OPN class will be as presented in the figure 7.

Protocol Model: This transformation is similar to transformations from UML state charts to Petri Nets (like in [8]). Figure 8 present the equivalent Petri Net in the CO-OPN MobilePhoneController class to the *Protocol Model* in 4(b).

Operation Schemas: The transformation in what concerns the *Operation Schemas* can be defined as:

$$\forall op \in M_i, msg \in M_o, pre \in PRE, post \in POST, \exists o \in O : \\ T_r(o) =< T_r(op), T_r(msg), T_r(pre)..T_r(post) > \quad (5)$$

taking into account that: O is the set of Fondue Operation Schemas; PRE set of pre-conditions; $POST$ the set of post-conditions;

The pre and post-conditions are based on control operators ($if...then...else...$), affectation based on OCL expressions.

[2] a Place in a CO-OPN class is like a place in a Petri Net with the difference that, in CO-OPN, a Place is of a certain type provided by the associated ADT

```
Class Card;
Interface
 Type
    card;
...
Body
 Use
    Naturals;
    CardControler;
    TelephoneNr;
    Pin;
 Places
    cardPin _ : pin;
    nPinTries _ : natural;
    cardState _ : cardControler;
    phoneNumber _ : telephoneNr;
 Initial
    cardState unlockedPin;
 ...
 ...
End Card;
```

Fig. 7. Class Card transformed from Fondue to CO-OPN

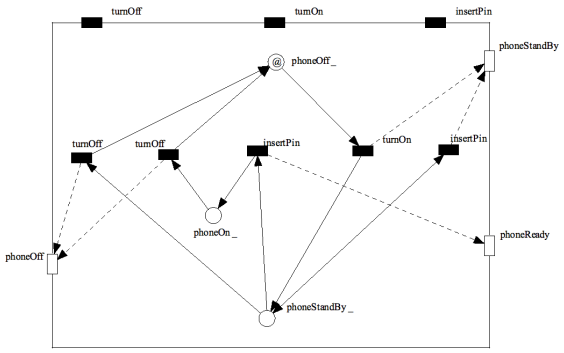

Fig. 8. Protocol Model UserPhoneController transformed from Fondue to CO-OPN

For transformation, all expressions of type

$$\{if\ \ lexpr\ \ then\ \ expr\ \ else\ \ expr|logicalvar\ :=\ oclexpr|expr, expr\} \quad (6)$$

will be transformed into only positive conditional axioms:

```
if cond1 then
  if cond2 then do1 else do2
else
  do3
```

Will be transformed into 3 positive conditionaql axioms:

```
if cond1 and cond2 then do1;
if cond1 and not(cond2) then do2;
if not(cond1) then do3;
```

or, more specifically in CO-OPN syntax:

```
(cond1) = true & (cond2) = true => op With do1
(cond1) = true & (cond2) = false => op With  do2
(cond1) = false => op With do3
```

In general, the transformation will produce several components in CO-OPN of format: $TrOCL(lexpr) =< logical\ expr, synchronisation >$

We should note that, for logical expressions that are simple boolean conditions without access to elements in Class model, we will have $synchronization = \oslash$. Moreover, the .. operator is used to gather the result of each sub expressions. It means conjunction of logical expressionss and sequence of synchronizations. The result will be one CO-OPN axiom for each flatenned axioms.

Taking the following piece of the *Operation Schema* in figure 5:

```
elseif c.cardPin <> pin
  c.nPinTries = c.nPinTries@pre + 1 and
  sender^wrongPin
  if c.nPinTries = 3 then
    c.CardStatus::lockedPin and
    c.nPinTries = 0
end
```

The CO-OPN resulting axioms for class Card will be of form:

```
(c.getcardPin p = pin)=false
  =>insertPin pin With wrongPin // c.getnPinTries n .. c.setnPinTries n+1

(c.getcardPin p = pin)=false and (c.getnPinTries 3 = true)
  =>insertPin pin With c.setcardStatus lockedPin // c.setnPinTries 0
```

where the operator // represents the execution in parallel of the different expressions.

2.4 Transformation Execution

The transformation execution is the "map" from the transformation formalization into one (or several) transformation languages. Since not all of the transformation languages provide the same functionalities we decided to adopt several of them. Thus, we will use them in order to "enrich" each other and to be able to provide an execution to our transformation. In Fig. 9 it is possible to see the general process of the transformation execution. The grey parts of the figure represent what was previously mentioned: more than one approach of transformation can be adopted. In this case we present an architecture using a transformation language that is based on a transformation model (e.g. the Model Transformation Language [7] (MTL)) and another based on directly specifying transformation rules (e.g. Mod-Transf - a XML and ruled based transformation language [9]).

In general, the mechanism of executing the transformation follows the standard process defined by the Object Management Group **??** (OMG). This means defining and using the Meta Model of each one of the languages as an instance of Meta Object Facility[10] (MOF). The transformation rules (or transformation model that is also an

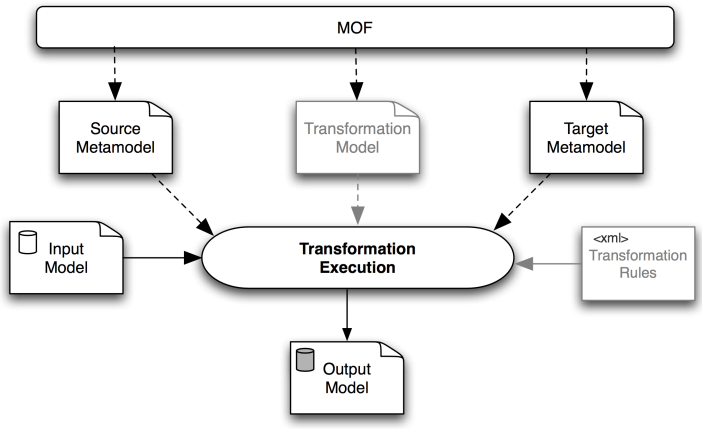

Fig. 9. General transformation execution

instance of MOF) can be written profiting from the fact that source (Fondue in our approach) and target (CO-OPN) languages abstract syntax are defined using the exact same methodology. As for the technology used, standards for (meta)model exploration and creation have been defined meaning that we can use them in order to coherently execute the transformation.

The basic idea of this particular transformation is to give part of the source language semantics using the transformation rules applied to the abstract syntax, being the other part provide directly and automatically by the fact that the transformation leads to a CO-OPN model (like described in [11]). This leads to a model in CO-OPN formal specification language allowing state space exploration and thus automatic test generation as explained in the next sections.

2.5 Tools

The software utilities we have under development in our laboratory that support the work described previously include: model transformation tools that handle MOF models exploration; generic model browsing; Java interfaces generation in order to generically explore a create a model of a given type; and model transformation features. The tool is capable to cope with plugins (basically the definition of the transformation rules and their algorithms) that will use existing generated Java interfaces to achieve transformation. These interfaces are a direct map for the metamodels of both target and source languages to be transformed.

3 Introduction to Model Based Testing (using CO-OPN Specifications)

Our approach to test case generation stems from the pioneer works of Bernot, Gaudel and Marre [12] on model-based testing using formal specifications. This work has been

extended by Barbey and Péraire in their Phd. Thesis which address the problematic of testing Object Oriented systems and finding practical tools for doing so.

In a nutshell, the approach can be described as follows: given a non-ambiguous model of the SUT (System Under Test), the test engineer will provide hypotheses about the functioning of the SUT. The purpose of these hypotheses is to generalize the SUT's behavior so that equivalent behaviors can be mapped into classes of system inputs – the test cases. By generalizing the behavior of the SUT we reduce the amount of system inputs necessary to perform exhaustive testing, which is in the general case infinite as Bernot, Gaudel and Marre describe in [12]. In fact, if we look at an SUT as being a black box with a number of available operations, the number of test cases necessary to fully test that system will include all possible sequences of calls to the SUT's operations. Moreover, if the operations the SUT makes available include parameters, all the possible values of those parameters will have to be explored. The generalization hypotheses are then provided either about sequences of calls to operations of the SUT, or about the values that are the parameters of those operations. Ideally, our approach will reduce a test set of infinite size to one of finite size that can be applied the system SUT in practicable time.

Clearly, the approach is biased by the quality of the hypotheses the test engineer will provide about the functioning of the SUT: while hypotheses which are too weak will lead test sets which are too large to be practical, hypotheses which are not correct generalizations of the SUT's behavior will lead to test sets which are not representative of the SUT's full behavior.

3.1 The model and the SUT (System Under Test)

Since in model-based testing the idea is to compare the SUT to its model, let us now discuss the model. Firstly, it is necessary that there exists a one-to-one morphism between the signatures of the operations of the SUT and the ones of the model – otherwise is makes no sense to try to compare them. By being non-ambiguous, the model allows exploring a state space which is in principle more abstract but equivalent to the one of the SUT. The test cases that are inferred from the hypotheses about the SUT behavior can be "ran" through the model in order to provide them with semantics (i.e., are the test cases valid or invalid behaviors according to the model).

Another purpose of the SUT's model is to provide a means for automatically determining classes of input parameters to SUT's operations that will produce an equivalent behavior in those operations (see chapter 4 of the well known book [13] from Glenford J. Meyers for an introduction to the subject). Given that our modeling language CO-OPN includes syntactic constructs to define the behavior of operations over the SUT, we perform an analysis of these constructs in order to find those equivalence classes.

3.2 Oracle and Test Driver

Other important issues related to testing are the *test driver* and the *oracle* as Péraire, Barbey and Buchs describe in [14, 15]. The purpose of the test driver is to provide a means of applying the generated tests to the SUT. The test driver will also be in charge of recovering the observable results of executing the test cases. These results will be passed to the oracle which will decide of the success of the test, i.e., of the conformance of the results observed in the SUT to the ones predicted by the model. We will not go into these topics in this paper, although some experiments we have realized which have produced interesting results are reported in [16].

3.3 Formalization of the Approach

We can then summarize the objective of model-based testing as follows: being P a program belonging to the class of all possibles SUTs, SP a CO-OPN specification, $\vDash$ a satisfaction relation between SUTs and CO-OPN specifications and $\vDash_o$ an oracle satisfaction relation between SUTs and test sets:

$$P \vDash SP \Leftrightarrow P \vDash_o T_{SP} \tag{7}$$

We would like to find T_{SP} which is a set of tests having the same semantics as SP. Ideally, comparing the SUT to the model would be equivalent to comparing the SUT to the test set T_{SP}. The latter comparison is done by the oracle which examines the result of running the test cases through the test driver. However, our approach includes hypotheses about the SUT, which means we extend equation 7 to the following (where $T_{SP,H}$ stands for a test set having the same semantics as SP but reduced by hypotheses H[3] which generalize behaviors of P):

$$(P \text{ satisfies } H) \Rightarrow (P \vDash SP \Leftrightarrow P \vDash_o T_{SP,H}) \tag{8}$$

We can only say the equivalence on right hand side of equation 8 holds if SUT P satisfies hypotheses H. Since it is not trivial to prove that an SUT satisfies hypotheses about its behavior, the quality of the obtained test set will necessarily depend on the quality of the hypotheses – which reflect the knowledge the test engineer has about the functioning of the SUT.

In the following sections of this paper we will focus on how tests are actually generated from a CO-OPN specification which results from the transformation of the initial UML model of the SUT. Figure 1 puts in evidence this test generation process, which consists on enriching the CO-OPN model with test intentions (or hypotheses about the SUT's behavior) and deriving the resulting tests using a set of tools we have developed for that purpose.

4 Testing CO-OPN Specifications: Brief Discussion on Methodology

As previously described, the CO-OPN specification language is an Object-Oriented formalism. When designing an approach to test CO-OPN specifications we want to take into consideration the fact that we want to test specifications as a whole, but also its parts. A first important remark to be done about our test methodology is that since it follows the model-based philosophy, we will always perform back-box testing. However, we can perform black-box testing focusing on a part of the specification. In figure 10 we exemplify testing at different levels of detail of the same specification. Three cases can be differentiated:

- *Testing a context*: A context is a particular feature of the CO-OPN language given it only acts as a coordinator for the objects it holds. Since a CO-OPN context does not have its own state, only one instance of context exists in a specification and

[3] In fact, since the oracle cannot always decide P satisfies a test case, it may become necessary to include in H additional hypotheses that extend the oracle's capability of observation.

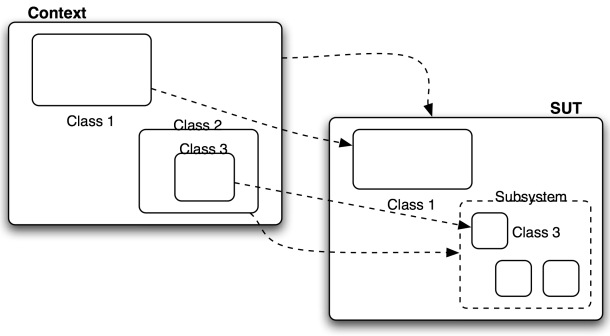

Fig. 10. Testing a CO-OPN specification at different level of detail

there are no variables of type context. Typically contexts are used as the outermost layer of a specification, defining methods and gates which correspond to inputs and outputs of the system. In terms of test artifacts, the outermost context corresponds to the interface of the SUT;

– *Testing a class*: In figure 10 it is possible to differentiate two scenarios while testing a class:
  - *A class in the model corresponds to a class in the SUT*: In this case we are in the scenario of unit testing a class. We can consider the class to be an SUT on its own and generate tests for it. We can envisage using a commercially well known unit test driver such as JUnit [17] for Java in order to practically apply the tests. However, if a class has references to objects (as they usually do), these references need to be initialized so the class can be tested correctly. If the references are initialized when the object is built, there is no problem. However, if the references are passed by reference to the class constructor, it may become necessary that the test is also able to pre-generate those references;
  - *A class in the model corresponds to a subsystem in the SUT*: It may happen that there is not a direct mapping between a CO-OPN class and an implementation class. A CO-OPN class may correspond to a subsystem of several classes in the implementation. In this case the implementation subsystem has to be encapsulated by the interface defined in the specification. The test driver will then have to perform the connections between the calls to the interface and their mapping on the subsystem.

5 The Test Selection Language *TestSel*

The test intentions appearing in figure 1 are expressed in our test selection language *TestSel*. Barbey, Péraire and Buchs present in [15, 14] specifically devised templates of hypotheses (can be considered as building bricks for more complex hypotheses) and a methodology for applying them. They claim in their work the methodology they present leads to a good quality of hypotheses, thus to test sets that uncover errors in a wide range of possible SUTs. *TestSel* extends their work by introducing language constructs to compose templates of hypotheses. With these constructs we are able to

build refined hypotheses about the SUT's behavior. In order to present $TestSel$ we will start by the language we have chosen to represent our test cases which is called HML (Hennessy-Milner Logic).

5.1 Test Representation Media - The HML Formalism

HML is a simple temporal logic built to express properties of processes. Its capability to express process properties as graphs of events over time makes it an interesting language for expressing test cases. In particular, HML_{SP} stands for the language of HML formulas over a given CO-OPN specification SP. By this we mean that HML_{SP} corresponds to HML formulas over CO-OPN events, where a CO-OPN event corresponds to a pair $\langle Input, Output\rangle$, $Input$ and $Output$ being synchronizations over the CO-OPN specification's methods and gates respectively. In the following definition of the abstract syntax of HML_{SP}, T represents the always true constant (verified by any process in any state) and $Event_{SP}$ is the set containing all the Input/Output synchronization pairs over SP.

Definition 1. *Syntax of* HML_{SP}

- $T \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\neg f) \in HML_{SP}$
- $f, g \in HML_{SP} \Rightarrow (f \wedge g) \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\langle e \rangle f) \in HML_{SP}$ *where* $e \in Event_{SP}$

We express CO-OPN's semantics using transition systems, so before providing the semantics of HML_{SP} let us start by defining the notion of transition system: the transition system denoted by a CO-OPN specification $SP$is a quadruple $\langle Q, Event(SP), \rightarrow, i\rangle \in \Gamma$ (Γ being the class of all transition systems) where Q is the set of all states in SP, $\rightarrow \subseteq Q \times Event_{SP} \times Q$ and i is a non empty initial state. We also define equivalence in the CO-OPN world as the *bisimulation* equivalence $\underline{\Leftrightarrow}$ (see Biberstein's Phd thesis [4] on CO-OPN's semantics) between the transition systems denoting the semantics of CO-OPN models. Taking again equation 7, we can better define the satisfaction relation $\vDash$ between an SUT P and a CO-OPN specification SP using the bisimulation relation. Being $G(P)$ and $G(SP)$ transition systems representing the semantics of P and of SP we can write:

$$P \vDash SP \Leftrightarrow G(P) \underline{\Leftrightarrow} G(SP)^4 \tag{9}$$

The semantics of HML_{SP} is defined in terms of the satisfaction relation $\vDash_{HML_{SP}}$ between the transition system denoted by specification SP and HML_{SP} formulas. Formally, given a transition system $G = \langle Q, Event(SP), \rightarrow, i\rangle$ denoting SP and a state $q \in Q$, the satisfaction relation $\vDash_{HML_{SP}} \subseteq \Gamma \times Q \times HML_{SP}$ is defined as:

Definition 2. *Semantics of* HML_{SP}

- $G, q \vDash_{HML_{SP}} T$
  (specification SP always satisfies formula T at state q)

[4] We will not discuss how to obtain a transition system from a CO-OPN specification SP, given that the purpose of this paper is not to explain the semantics of a CO-OPN specification – we rather aim at expressing the relation between the test language and the specification language.

- $G, q \vDash_{HML_{SP}} (\neg f) \Leftrightarrow G, q \nvDash_{HML_{SP}} f$
  ($\neg f$ is satisfied by specification SP at state q if SP in that same state q does not satisfy f)
- $G, q \vDash_{HML_{SP}} (f \wedge g) \Leftrightarrow G, q \vDash_{HML_{SP}} f$ *and* $G, q \vDash_{HML_{SP}} g$
  ($f \wedge g$ is satisfied by specification SP at state q if f is satisfied by SP at state q and g is satisfied by SP at state q)
- $G, q \vDash_{HML_{SP}} (e\langle f \rangle) \Leftrightarrow \exists e \in Event_{SP}$ *such that* $q \xrightarrow{e} q' \in \rightarrow$ *and* $G, q \vDash_{HML_{SP}} f$
  ($e\langle f \rangle$ is satisfied by SP at state q if there is an event $e \in Event_{SP}$ leading from state q to q' and f is satisfied by SP at from state q').

If we consider G to be the transition system representing the semantics of a CO-OPN specification SP and $q \in Q$ the initial state of model SP, the test set obtained from a set of HML_{SP} formulas is such that:

Definition 3. *Test Set for a given set of formulae $F \subseteq HML_{SP}$:*

$$Test_{SP,G}(F) = \{\langle f, Result\rangle \in F \times \{true, false\} \mid (G, q \vDash_{HML_{SP}} f \text{ and } Result = true) \text{ or } (G, q \nvDash_{HML_{SP}} f \text{ and } Result = false)\} \quad (10)$$

In this way we classify HML_{SP} formulas as valid or invalid behaviors of the model of the SUT described by SP.

5.2 Full Agreement between HML and CO-OPN Semantics

We are now in measure to define the satisfaction relation $\vDash_o$ used in equation 8 more precisely. Being $G(P)$ the transition system representing the semantics of SUT P (which we want to observe through the execution of test cases), $G(SP)$ the transition system representing the semantics of the CO-OPN specification SP and $F \subseteq HML_{SP}$, the $\vDash_o$ relation is given by:

Definition 4. *Oracle satisfaction* $\vDash_o$

$$P \vDash_o Test_{SP,G(SP)}(F) \Leftrightarrow Test_{SP,G(P)}(F) = Test_{SP,G(SP)}(F) \quad (11)$$

Equation 4 illustrates the fact of applying a test set to an SUT. It states that a CO-OPN specification SP and an SUT P should satisfy any set of HML_{SP} formulas is the same manner. In other words, SP and P should have the same behavior.

We will not go deeply into this subject, but it is possible to use HML as a testing language because there exists a full agreement between CO-OPN equivalence and HML equivalence. CO-OPN equivalence is given by the bisimulation relation and Hennessy has shown in [18] that two transition systems can be distinguished by HML if and only if they are not bisimulation equivalent.

According to equation 7 we are checking if SUT P has the same semantics as specification SP. We do this by first calculating a test set T_{SP} with the same semantics as SP (definition 3) and then checking if P also satisfies T_{SP} through black-box observation of its behavior when the tests are ran. This said, the full agreement between CO-OPN equivalence and HML equivalence is fundamental for our approach. It is this result that allows us to say that comparing an SUT P to a CO-OPN specification SP through the usage a test set T_{SP} is equivalent to comparing directly the transition systems denoted by P and SP using bisimulation. In other words, this means that we don't lose discriminating power between $\vDash$ and $\vDash_o$.

5.3 Advantages and Disadvantages of HML as a Test Formalism

Another interesting aspect of using HML as a test representation media is the fact that we can make use of *not* ($\neg$) and *and* ($\wedge$) operators. The *not* operator allows us to state that an SUT does not produce a given behavior, while the *and* operator allows us to discriminate branching non-determinism. While the semantics of these operators is straightforward as previously explained in this paper, it may be not trivial to apply them in practice while testing a real SUT. Let us exemplify by imagining the application of a negative HML formula (a negative test case) to an SUT. In this case, the *oracle* would have to decide about the satisfaction of the negative formula which is not a trivial task. In fact if the SUT blocks during the execution of a negative test case, the oracle may not always be able to distinguish between the blockage required by the specification and a blockage provoked by a fault present in the SUT. By a blockage in a CO-OPN specification we understand the fact that an operation is not available from a given state. In that sense the we need the oracle to able to distinguish between that kind of blockages and the blockages that are due to errors in the code of the SUT.

On the other hand, if we would like to apply to an SUT a test case represented by a conjunctive HML formula (including *and* operators) other problems would arise: the semantics of the *and* operator are so that for a transition system to verify a formula $(f \wedge g)$ both f and g have to be satisfied starting from the same state. Practically, to apply a formula $(f \wedge g)$ to an SUT a test driver would have to be able to either first test f and then g, or the reverse. In order to be able to do this, a "backtracking" capability of the SUT would be necessary in order to go back to the state where the formula splits. Although we do not provide solutions for an oracle and a test driver capable of testing negative or conjunctive formulas, we point these two problems as issues to take into consideration while using HML as a test case formalism.

5.4 Example – A Set of Selection Hypotheses for the Mobile Phone Example

Before presenting formally the structure of $TestSel$ we will provide an example of a set of test selection hypotheses for the mobile phone example provided in section 2.1. The example in figure 11 is given in the concrete syntax of $TestSel$ that we have implemented in our IDE for the CO-OPN language. This IDE is called CoopnBuilder and includes editors for context, class and ADT modules of CO-OPN specifications. $TestSel$ is implemented as an extra module for CO-OPN specifications. Instances of these types of modules are called *constraint* modules – is the sense that they constrain the whole set of possible executions of the SUT.

Figure 11 depicts a single *constraint* module called *NatelCons*. CoopnBuilder allows multiple constraints modules per CO-OPN specification. Informally, the structure of a constraint module includes the following sections:

- *Interface*: defines the name of the constraints that are defined by a module. Each constraint name corresponds to a set of HML formulas and the union of all these sets is the final test set defined by the module. In the future we would like to compose constraints coming from several modules;
- *Body*: declares the properties necessary to the construction of the constraints declared in the interface. It includes five sections:

```
ConstraintSet NatelCons;
   Interface
      Constraints
         pinTest;
         block;
         reachOn;

   Body
      Constraints
         nWrongPins

      Use
         Boolean
         Pin

      Axioms
1        subUniformity(p) => HML(<turnOn with null>,<insertPin(p) with g> T) in pinTest;

2        [] in nWrongPins;

3        f in nWrongPins => f . HML(<insertPin(newPin(1 1 1 1)) with g> T) in nWrongPins;

4        f in nWrongPins & nbEvents(f) < 4 => HML(<turnOn with null>) . f .
         HML(<insertPin(newPin(1 2 3 4)) with g> T) in insertPins;

      Variables
         p : Pin
         f : HML
         g : gate

      External

End NatelCons;
```

Fig. 11. Test selection hypotheses for the Natel example

- *Contraints*: declares the constraints defined locally to help in the construction of the exported constraints. They are not exported from the module;
- *Use*: declares specification modules (namely class and ADT modules) that are used to build the constraints;
- *Axioms*: declares axioms and rules that establish elementary behaviors of the SUT. The conjunction of these behaviors in the *constraint* module corresponds to the reduction hypotheses as it was stated in section 3;
- *Variables*: establishes the type of the variables used in the definitions of the *Axioms* section;
- *External:* declares functions used in HML formulas (test cases) during testing time (as opposed to test generation time). The purpose of these functions is to calculate values over non-deterministic outputs of the SUT. For further explanations on this subject we direct the interested reader to [19].

The *Axioms* section is clearly the most relevant one. In this paper we will not provide a textual description of the semantics of the language of constraint module's axioms since it can be found in [19]. However, some comments about the axioms that can be found in figure 11 follow. Please keep in mind that an axiom is of the form `condition => assignment` where the *assignement* (of a set of HML formulas to the set represented by the constraint name) only happens if the *condition* holds.

– **axiom 1**

```
subUniformity(p) => HML(<turnOn with null>,<insertPin(p) with g> T) in pinTest;
```

This axiom generates test cases that start by turning on the phone and then insert a pin value. The *subuniformity* operator selects for variable p values according to the behavior of the operation *insertPin*, which either validates or invalidates the introduced pin number. This axioms will then produce two tests: turn on the phone and insert a correct pin; turn on the phone and insert an incorrect pin. The language makes available two more operators *exhaust* and *uniformity* that are similar to *subuniformity* but that select all values or only one value in the domain from the variable in parameter, respectively.

– **axioms 2 and 3**

```
                                    [] in nWrongPins;
f in nWrongPins => f . HML(<insertPin(newPin(1 1 1 1)) with g> T) in nWrongPins;
```

This couple of axioms is used to produce the same set of tests. Axiom 3 is recursive since it builds an HML formula that inserts a wrong pin and then concatenates it ("." is the concatenation symbol for HML formulas) with formulas of the same type. Axiom 2 represents the base HML formula T, which is the stop condition for the recursion in axiom 3 (given that axiom 3 is defined in terms of itself). The tests produced by these axioms are sequences of any size of wrong pins insertions (assuming *(newPin 1 2 3 4)* is the correct pin).

– **axiom 4**

```
f in nWrongPins & nbEvents(f) < 4 => HML(<turnOn with null>) . f .
    HML(<insertPin(newPin(1 2 3 4)) with g> T) in insertPins;
```

This axiom uses a *nbEvents* operator that limits the number of events in HML formula *f*. In this case the idea is to use the previously defined constraint name *nWrongPins* to build sequences of at most four wrong pins. These sequences are then concatenated at the beginning with an operation to turn on the phone and at the end with a correct pin insertion (notice however that the output of the event is variable). The idea in this case is to test if the system only blocks at the introduction of more than three wrong pins and behaves correctly in the remaining cases. Examples of other operators over HML formulas are: *depth* – number of events of the deepest branch of an HML formula; *nbOccurences* – number of occurrences of a method name in an HML formula; *positive* – HML formulas without *not* ($\neg$) operators; *sequence* – HML formulas without *and* ($\wedge$) operators.

5.5 The Structure of *TestSel*

As its name indicates, *TestSel* is a test selection language rather than a test reduction language. Despite the fact that we have defined in 3 the process of finding tests as the progressive reduction of exhaustive test set, this process cannot be reproduced in the real world for a simple reason: it is not possible to generate the exhaustive (infinite) test set in finite time and then reduce it. In order to overcome this operational difficulty while still employing the presented theoretical framework, we have thus decided to implement practically the test finding process as one of selection – using logic programming principles. The basic approach is explained by Barbey in [15], where he starts by defining the language $HML_{SP,X}$ – our HML_{SP} language extended with variables. The test selection is then practically achieved by instantiation of the variables present in $HML_{SP,X}$ formulas. Given a CO-OPN specification SP, the set X_{HML} of variables over HML formulas and the set X_{Event} of variables over SP's events, the syntax of $HML_{SP,X}$ is defined as follows:

Definition 5. *Syntax of* $HML_{SP,X}$

- $T \in HML_{SP}$
- $x \in X_{HML} \Rightarrow x \in HML_{SP,X}$
- $f \in HML_{SP,X} \Rightarrow (\neg f) \in HML_{SP,X}$
- $f, g \in HML_{SP,X} \Rightarrow (f \wedge g) \in HML_{SP,X}$
- $f \in HML_{SP} \Rightarrow (\langle e \rangle f) \in HML_{SP}$ *where* $e \in Event_{SP,X}$)

The set $Event_{SP,X}$*) includes CO-OPN pairs* $\langle Input, Output \rangle$*, Input and Output being synchronizations including variables. Two CO-OPN events can be synchronized simultaneously, in sequence or in parallel. A CO-OPN event is a method or a gate name, followed by a set of parameters.* $Event_{SP,X}$ *includes variables on methods or gates names of the specifications (which we will call* X_{MG}*) as well as variables over event parameters – these can be sets of values described in ADT modules (which we will call* X_S*) of the specification or references to objects of classes defined in the specification (which we will call* X_C*).*

We can then consider the exhaustive test set to be represented by $\langle f, r \rangle$ where $f \in HML_{SP,X}$. In fact, given that f has free variables it cannot be applied directly to the SUT. Hypotheses about the behavior of the SUT will serve the purpose of instantiating those free variables – leading to ground HML formulas that can be used as a test cases for an SUT. The process of test selection can then be seen as the process of transforming an $HML_{SP,X}$ formula into an HML_{SP} one, by means of hypotheses about SUT that can be translated in constraints on the formula's variables.

The Abstract Syntax of *TestSel* Before providing an example of using $TestSel$ we will present its abstract syntax. The purpose of this section is to layout the basis for being able to precisely define the semantics of $TestSel$. While reading this section, please keep in mind section 5.4 of this paper where the syntax and the semantics of $TestSel$ was informally introduced. $TestSel$ has three syntactic layers, namely:

- *CO-OPN event*: includes the possible Input/Output synchronizations pairs of a CO-OPN specification. The set of these pairs for a specification SP is given in definition 5 by the set $Event(SP, X_S)$;
- *HML*: set of $HML_{SP,X}$ formulas over a CO-OPN specification SP;
- *Constraints*: constraints over variables of $HML_{SP,X}$ formulas. Our language allows constraints over variables that represent: execution paths – sequences of events with HML operators $\wedge$ and $\neg$); values that are parameters of operations – CO-OPN class instances or CO-OPN sorts (sets of values) defined in ADT modules.

The abstract syntax of the first two layers in the above list has already been provided in definition 5. In what concerns the third layer we will provide the abstract syntax of a constraint module over a specification SP. The language will be defined in a top-down fashion:

Definition 6. *Constraint module*

A constraint module over a CO-OPN specification SP *is a quintuplet* $\langle SP, K, ax, X, F_{SP} \rangle \in \Psi_{SP}$*, where:*

- SP *is a CO-OPN specification;*
- K *is the set of constraint names defined by the constraints module;*

- X *is a set of typed variables* $X_{HML} \cup X_{MG} \cup X_C \cup X_S$*;*
- F_{SP} *is a set of function signatures defined in ADT modules of specification* SP*;*
- $ax \subseteq AX_{K,X,SP}$ *is a set of axioms defined over* $HML_{SP,X}$ *formulas, predefined operators, constraint names in* K *and variables in* X*;*

Intuitively speaking, a constraint module for a specification SP will define a set of constraint names – each name representing a different generalization of a behavior of the SUT. The constraints are defined by axioms that belong to the $AX_{K,X,SP}$ language. Still, before proceeding with the definition of this language we will present the syntax of terms over $HML_{SP,X}$ formulas as this will be necessary for subsequent definitions:

Definition 7. *The terms* $T_{HML_{SP,X}}$ *over* $HML_{SP,X}$

- $t \in HML_{SP,X} \Rightarrow t \in T_{HML_{SP,X}}$
- $t1, t2 \in HML_{SP,X} \Rightarrow t1 \,.\, t2 \in T_{HML_{SP,X}}$

the intuition behind this definition is to 7 provide us with the necessary syntax for the concatenation of $HML_{SP,X}$ formulas. We thus define the language of constraint axioms over a CO-OPN specification SP as follows:

Definition 8. *Given* K*,* X *and* SP *as defined previously, a constraint axiom is a triplet belonging to the relation* $AX_{K,X,SP}$ *such that:*

$$AX_{K,X,SP} = Cond_{K,X,SP} \times T_{HML_{SP,X}} \times K$$

where:

- $Cond_{K,X,SP}$ *is a conjunction of atomic conditions;*
- $T_{HML_{SP,X}}$ *is term built from* $HML_{SP,X}$ *formulas;*
- K *is a constraint name.*

This syntax for constraint axioms allows us to see constraints as sets of HML formulas – an instantiated $HML_{SP,X}$ formula $Formula$ is produced by a constraint $ConsName$ only if we can find a substitution to the variables of $formula$ that satisfies the condition $Condition$. We are now missing the definition of $Cond_{K,X,SP}$:

Definition 9. *Conditions* $Cond_{K,X,SP}$ *of a behavioral axiom*

Given K*,* X *and* SP *as defined previously, the set* $Cond_{K,X,SP}$ *is a conjunction of atomic conditions such that:*

$$\forall n \in \mathbb{N}, ac_i \in AC_{K,X,SP}, i \in \{0..n\}, ac_1 \wedge ac_2 \wedge \ldots ac_n \in Cond_{K,X,SP}$$

Finally we will define the set $AC_{K,X,SP}$ of atomic conditions. An atomic condition is a constraint over variables of X.

Definition 10. *Atomic conditions* $AC_{K,X,SP}$

$\forall k \in K, t \in T_{HML_{SP,X}}, tn, tn' \in T_{NAT}, tb, tb' \in T_{BOOL}, x \in X_S,$

$$t\,in\,k,\ uniform(x),\ exhaust(x),\ tn\,cmpOp_{\mathbb{N}}\,tn',\ tb\,cmpOp_{\mathbb{B}}\,tb' \ \in AC_{K,X,SP}$$

where $cmpOp_{\mathbb{N}} \in \{=, <>, <, >, <=, >=\}$ *and* $cmpOp_{\mathbb{B}} \in \{=, <>\}$.

T_{NAT} represents the set of terms over arithmetic expressions. Given $t \in T_{HML_{SP,X}}$, T_{NAT} is defined as:

$$n,\ depth(t),\ nbEvents(t),\ nbOccurrences(m,t),\ tn\ op_{\mathbb{N}}\ tn' \in T_{NAT}$$

where $n \in \mathbb{N}$, m is a method name defined in SP, $tn, tn' \in T_{NAT}$ and $op_{\mathbb{N}} \in \{+, -, *, /\}$.

T_{BOOL} represents the set of terms over boolean expressions. Given $t \in T_{HML_{SP,X}}$, T_{BOOL} is defined as:

$$\{true, false\},\ onlyConstructor(t),\ onlyMutator(t),\ onlyObserver(t),\ sequence(t),\\ positive(t),\ trace(t) \in T_{BOOL}$$

Semantics of *TestSel* After having described the abstract syntax of $TestSel$, we are now in measure of providing its semantics:

Definition 11. *Semantics of TestSel*

Given a CO-OPN specification SP, the semantics of a constraint module $CONS = \langle SP, K, ax, X, F_{SP} \rangle \in \Psi_{SP}$ is the set of all $HML_{SP_{CONS}}$ formulas such that:

$$HML_{SP}^{CONS} = \bigcup_{axiom \in ax} \{f \in HML_{SP} \mid f \vdash axiom\}$$

The informal meaning of definition 11 is the following: for each axiom of the constraints module all the HML_{SP} formulas (without variables) that satisfy it are collected in a set. The set of test cases produced by a constraint module is the union of all sets of test cases produced by each individual axiom. We will not further develop the $\vdash$ relation in this paper. In order to verify if $f \vdash axiom$ we have to find a substitution of the variables of $axiom$ so that we can find f. In particular, the substitution of variables that are quantified with the $subuniformity$ operator is complex given that it becomes necessary to analyze the behavior of the operations in the CO-OPN specification.

In order to finish this section of the paper we will state the validity of the test sets obtained by the *constraint* modules of our $TestSel$ language:

Theorem 1. *Given a CO-OPN specification SP and a constraint module $CONS$, the test set $Test_{SP,G(SP)}(HML_{SP}^{CONS})$ obtained from $CONS$ is a valid test set, meaning it does not reject correct programs:*

Proof.

$$Test_{SP,G(SP)}(HML_{SP}^{CONS}) \subseteq Test_{SP,G(SP)}(HML_{SP})$$

is trivial by construction. □

In theorem 1 we show that test sets that are selected by $TestSel$ for a CO-OPN specification SP are part of the exhaustive test set. Thus the selection process does not introduce invalid test cases in the final test set. Validity of the tests is necessary but not sufficient for measuring the quality of a test set. In fact, we would have to prove that the union of all the behaviors described by all the constraint modules about a specification corresponds to a correct generalization of the behavior of the specification. This can be reduced to the problem of measuring the coverage of the obtained test set.

5.6 Tools for Test Production

As we have already mentioned in this paper, an IDE for the CO-OPN language called CoopnBuilder already exists. We have used this infrastructure in order to implement our language *TestSel*, as can be seen from the example in 5.4. From this front end we are able to produce *Prolog* code that generates the test sets. The reason why we have used Prolog for this task has to do with the fact that the resolution mechanism of this language allows relatively straightforward mapping between its semantics and the semantics of *TestSel*. In fact, *Prolog* is a theorem prover that tries to verify if a logical clause can be induced from the available rules. If the logical clause to be proved includes variables, Prolog will find all the substitutions for those variables that make the clause true. This is similar to the semantics of *TestSel* – we want to find substitutions for the axioms of a constraint module that make the constraints over the variables of those axioms true. In this process we find fully instantiated HML formulas which are sequences of inputs for the SUT.

On the other hand, only (syntactically) finding is sequences of inputs is not enough. We also need to provide them with semantics in order to turn them into test cases, as shown in definition 3. To do that we have two options at our disposal: a prototyping tool that turns CO-OPN specifications into Java programs [20]; a translator that converts CO-OPN specifications into Prolog programs [21]. Both options are currently implemented and allow verifying if an HML formula is a valid behavior of a CO-OPN specification. We are inclined to pursue the latter option given that the integration with the Prolog code produced from the constraint modules becomes more natural. Other reasons for this choice have to do with the fact that Prolog is a language where the concepts of code and data are mixed. This allows a natural reflection which is extremely useful for analyzing and decomposing the behavior of the operations defined in the specification.

```
X = next(coopnEvent(method(turnOn)), next(coopnEvent(method(insertPin(newPin
1 2 3 4)), gate(coopnSyncSim(pinResult(true),phoneReady))), t)) ;

X = next(coopnEvent(method(turnOn)), next(coopnEvent(method(insertPin(newPin
1 1 1 1)), gate(pinResult(false))), next(coopnEvent(method(insertPin(newPin 1
2 3 4)), gate(coopnSyncSim(pinResult(true),phoneReady))), t))) ;

X = next(coopnEvent(method(turnOn)), next(coopnEvent(method(insertPin(newPin
1 1 1 1)), gate(pinResult(false))), next(coopnEvent(method(insertPin(newPin 1
1 1 1)), gate(pinResult)), next(coopnEvent(method(newPin 1 2 3 4)), gate
(coopnSyncSim(pinResult(true),phoneReady))), t)))) ;

X = next(coopnEvent(method(turnOn)), next(coopnEvent(method(insertPin(newPin
1 1 1 1)), gate(pinResult(false))), next(coopnEvent(method(insertPin(newPin 1
1 1 1)), gate(pinResult(false))), next(coopnEvent(method(insertPin(newPin 1 1
1 1)), gate(coopnSyncSim(pinResult(false),cardPinLocked))), next(coopnEvent
(method(insertPin(newPin 1 2 3 4)), gate(cardPinLocked)), t))))) ;
```

Fig. 12. Tests generated for the Natel example

In figure 12 we present some of the tests which are (semi) automatically generated by our tool. In fact, the four lines in the figure represent four solutions to the constraint name *insertPins* declared in the constraint module of figure 11 for the Natel example. Our tool is not yet able to verify if the tests generated are valid or invalid behaviors of the specification, so we have chosen them by hand. In fact, given that in the axiom for the *insertPins* constraint (and previously, in the axiom for the *nWrongPins* constraint) the variable g is a gate that remains to be instantiated, many other solutions are possible. However, they will all represent invalid behaviors of the specification.

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 [22] 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.

Pretschner et al explain in [23] 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 Conclusion

In this paper we have presented our work on automatic test case generation for UML (Fondue) models. We have decided to tackle the problem in two phases, the first one being the translation of UML into a the formal specification language CO-OPN. CO-OPN has clearly defined semantics which allow us to explore the model soundly in order to produce test cases. The translation process that we formally define is based on the decomposition of the Fondue sub-models (environment, concept, protocol and operations) and their individual mapping into CO-OPN modules (ADT, classes and contexts). We also take into consideration the fact that the Fondue modules overlap or complement each other at certain points and include these implicit semantics in the translation. The process is introduced also by means of an example of a specification of a mobile phone.

The second phase of the problem concerns the automatics test generation. Here also we present a formal process, insisting on the definition of a language that will enable this process. We define the language at several layers of complexity, which allow to take our theory of testing into consideration while adapting it to the test selection needs of an engineer. We also provide a semi formal semantics for this language and illustrate it generating a test set for the mobile phone specification.

Tools are currently being implemented for the processed we describe. We are already able to partially automate the processes we describe.

References

1. Alfred Strohmeier. Fondue: An Object-Oriented Development Method based on the UML Notation. In *X Jornada Técnica de Ada-Spain, Documentación, ETSI de*

Telecommunicación, Universidad Politécnica de Madrid, Madrid, Spain, November 2001.
2. Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, july 2000.
3. Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. CO-OPN/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Canterbury, UK, July 21-23 1997*, pages 57–72. Chapman and Hall, Lo, 1997.
4. Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.
5. Object Management Group. Mda guide version 1.0.1. Technical report, OMG, 2003.
6. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
7. Triskell team. MTL Documentation. URL: http://modelware.inria.fr/rubrique4.html.
8. Zhaoxia Hu and Sol M. Shatz. Mapping uml diagrams to a petri net notation for system simulation. In Frank Maurer and Günther Ruhe, editors, *SEKE*, pages 213–219, 2004.
9. Triskell team. Mod-Transf - xml and ruled based transformation language. URL: http://modelware.inria.fr/rubrique15.html.
10. Object Management Group. Meta-Object Facility. URL: http://www.omg.org/technology/documents/formal/mof.htm.
11. Luis Pedro, Levi Lucio, and Didier Buchs. Prototyping Domain Specific Languages with COOPN. In *Rapid Integration of Software Engineering techniques*, 2005.
12. M.-C. Gaudel G. Bernot and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, 1991.
13. Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.
14. Cécile Péraire, Stéphane Barbey, and Didier Buchs. Test selection for object-oriented software based on formal specifications. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 385–403, London, UK, UK, 1998. Chapman & Hall, Ltd.
15. Stéphane Barbey, Didier Buchs, and Cécile Péraire. A theory of specification-based testing for object-oriented software. In *EDCC*, pages 303–320, 1996.
16. Levi Lucio, Luis Pedro, and Didier Buchs. A Methodology and a Framework for Model-Based Testing. In N. Guelfi, editor, *Rapid Integration of Software Engineering techniques*, volume LNCS 3475, page 5770. LNCS, 2005.
17. Erich Gamma and Kent Beck. Junit.org. URL: http://www.junit.org/.
18. Matthew Hennessy and Colin Stirling. The power of the future perfect in program logics. *Inf. Control*, 67(1-3):23–52, 1986.
19. Levi Lucio, Luis Pedro, and Didier Buchs. 16th ieee international workshop on rapid system prototyping (rsp 2005), 8-10 june 2005, montreal, canada. In *IEEE International Workshop on Rapid System Prototyping*, pages 195–201, 2005.
20. Ali Al-Shabibi, Didier Buchs, Mathieu Buffo, Stanislav Chachkov, Ang Chen, and David Hurzeler. Prototyping object oriented specifications. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN*

2003), Eindhoven, The Netherlands — Lecture Notes in Computer Science / Wil M. P. van der Aalst and Eike Best (Eds.), volume 2679, pages 473–482. Springer-Verlag, June 2003.
21. M. Buffo and D. Buchs. Symbolic simulation of coordinated algebraic petri nets using logic programming. University of Geneva – Internal Note.
22. Bruno Legeard and Fabien Peureux. Generation of functional test sequences from b formal specifications-presentation and industrial case study. In *ASE*, pages 377–381, 2001.
23. Jan Philipps, Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, Stefan Kriebel, and Kai Scholl. Model-based test case generation for smart cards. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.