ABSTRACT

In this paper we will describe our work on automatic generation of tests for Systems Under Test (SUT) described in the CO-OPN specification language that has been developed in our laboratory. CO-OPN (Concurrent Object Oriented Petri Nets) is a formalism for describing concurrent software systems that possesses most of the characteristics we can find in mainstream semi-formal modeling languages. Given its formal semantics, CO-OPN is a suitable formalism for model-based test case generation.

Within our work we have developed a test selection language for CO-OPN specifications which we have named SATEL (Semi-Automatic Testing Language). The purpose of SATEL is to allow the test engineer to express abstract test intentions that reflect his/her knowledge about the semantics of the SUT. Test intentions are expressed by execution traces with variables that can expanded into full test cases – this is done using the CO-OPN specification of the SUT as a reference for calculating the oracles. We call our test selection language semi-automatic because it allows explicitly choosing the shape of the test cases in a test intention, while relying on automatic mechanisms for calculating the equivalence classes of operations defined in the model.

1. INTRODUCTION

Many are the difficulties that arise when one tries to perform automatic test generation for a software system. Let us start by defining what is meant by "automatic". The approaches and tools that exist today for performing test generation can be mostly divided into two categories: white-box and black-box. While white-box testing is usually based on covering certain execution paths of the SUT (System Under Test) code, black-box testing is concerned about covering functionalities of the SUT described in a model. In any of these cases the automatic aspect of test generation is made difficult by the fact that the set of reachable states of either the SUT or the model is, in the general case, infinite. In this context, the notion of coverage of an SUT or a model by the generated tests depends strongly on which restrictions were made in order to "intelligently" cover the given state space. For example, if one is performing white-box testing, covering all decision points in the code is a classic strategy [1] and can automatically generate test cases. Using this kind of reasoning, many other strategies for "automatic" white-box test case generation might be devised.

On the other hand, by introducing the notion of a model of the SUT it might be expected that the problem of coverage of the SUT would be implicitly solved. Being that the model is an abstraction of the system we pretend to test, we could assume the abstractions introduced by the model would make extensive testing – modulo those abstractions – possible. In reality this is not true. Models of software systems can be also very complex and mainly abstract from having to deal with hardware, operating systems, software libraries or specific algorithms (unless we are aiming at directly testing those entities). What the model introduces in the chain of test generation is rather a form of redundancy – a way of comparing the SUT with an abstraction of itself in order to find differences which are possibly errors in the code. Again, we meet with the problem of reaching an infinite set of states of the model.

1.1 Our Approach

In this paper we propose a semi-automatic approach to test generation. The approach is semi-automatic in the sense that we allow the test engineer to state test intentions, while using unfolding techniques (introduced by Bernot, Gaudel and Marre in [2]) for automatically finding equivalence classes of inputs to operations of the model. It is our intention to explicitly make use of the test engineer’s knowledge in the test generation process. He/She will be able to express which parts of the model are relevant for testing and to impose limits on how that testing should be performed. However, given an operation of the model, he/she will also be able to state that the generated tests should include inputs that automatically cover the various behaviors (the equivalence classes) of that operation. For example, while testing a Banking application, the test engineer would be able to express that he/she wants a certain sequence of operations to be executed during testing (e.g. login user / introduce password / deposit amount / withdrawal amount), but also to...
express that the generated tests should automatically cover all the behaviors of the password operation – the password is either correct or incorrect for a given user.

Figure 1 depicts the process of testing an SUT using our approach. Assuming a CO-OPN [3] specification of the SUT exists, the test engineer writes a script of test intentions. These test intentions may make use of the semantics of the model in order to automatically cover equivalence classes of specifications’ operations (hence the dashed line). The test cases produced by the test intentions are then confronted with the specification for validation purposes. This step is necessary in order to decide whether a test case is a valid or an invalid behavior according to the specification. An example of an invalid behavior would be expecting an incoherent output while applying a given input to an operation of the specification. Although valid behaviors are the most interesting, invalid behaviors can also be used as test cases in the sense that they depict scenarios of execution that should not be allowed by the SUT. The rightmost arrow of figure 1 represents the verification of the SUT by submitting to it the validated test cases. This step is by no means simple and requires the existence of oracles for the generated test sets and drivers actually apply those tests to the SUT. Given that our current research is focused on producing test cases and their oracles, we will not explore the test driver issue in this paper.

1.2 Contribution

The novelty of our approach lies in the fact that we privilege the test engineer’s semantical knowledge of the SUT rather than emphasizing pure automatic test generation as for example in [4]. To our knowledge current research on testing is either focused on test drivers without automatic test generation abilities – e.g., the very successful TTCN-3 [5] – or on approaches which aim at building press-button tools. In the latter category we can mention the CLPS-B for limit testing approach from the university of Franche-Comté [4] which tries to automatically reach possible outcomes of the operations of a model expressed in the specification language B. Although very effective in certain situations, this approach implicitly requires a strong discretization of the model so the state space can be searched. Another approach with which we identify more is the AsmL Test Tool from Microsoft [6]. In this tool it is possible to generate a state space for a specification in the AsmL language and the test engineer can provide abstractions both for the state space itself and for input values. Algorithms for searching the reduced state space allow generating test cases that cover all or part of those states. Although the approach is in certain aspects similar to ours, we consider our test selection language to be more appropriate for the test engineer to express semantical knowledge about the SUT at a high level of abstraction.

The TOBIAS [7] tool is an automatic test generator based on producing combinatorial test sets from test purposes [8]. The idea of using test purposes is similar to that of using test intentions, although the TOBIAS tool suffers from the problem of calculating the oracles for the test sets. This is due to the fact that no semantically exploitable specification exists – contrarily to our approach. Despite, the authors try to overcome that difficulty by using VDM or JML assertions as a means of filtering interesting test cases [8] from the large combinatorial test sets.

Another aspect of our contribution lies in the fact that we use CO-OPN as our specification language for model-based test case generation. CO-OPN is based on Algebraic Data Types for describing data types and on structured Object Oriented Petri Nets for describing behavior (see [9] for details). The Petri Nets semantics of our specification languages places us very near to such formalisms as Harel’s statecharts [10] or the Behavioral State Machines introduced in UML 2.0 [11] (which is in fact an object-based variant of Harel’s statecharts). We are currently pursuing our research in the sense of applying our test intention-based techniques to test generation based on such mainstream models. We accomplish this by translating those models into CO-OPN and profiting from the partial equivalence of the semantics, as explained in [12].

1.3 Organization of the paper

The remaining of this paper is organized as follows: section 2 describes CO-OPN as a modeling language and introduces its semantics by means of an example. Section 3 discusses some of the requirements for a test intention language that will act over CO-OPN specifications. In section 4 we present the syntax and semantics of our test intention language SATEL. Finally, section 5 provides a concrete example of usage of our language and section 6 concludes.

2. THE CO-OPN SPECIFICATION LANGUAGE

In this section we will introduce the CO-OPN specification language. The objective is not to provide an exhaustive definition of the language, but rather to explore issues that are relevant while designing a language for semi-automatic test generation using CO-OPN specifications as models. In order to do this we establish an example of a Banking system model in CO-OPN. This example will be used throughout the paper.

Our Banking system is a multi-user centralized system that provides to users the possibility of managing their accounts via remote automatic teller machines. Before being able to perform operations on his/her account, the user has to authenticate in a two-step process: log in with a username; if the username exists, the system asks the corresponding password. If the user provides three wrong passwords, his/her account will become blocked and he/she will no longer be able to connect to the system. After having successfully authenticated, the operations available to a user are balance display, money deposit and money withdrawal.
The following subsections introduce the different kinds of modules which can be present in a CO-OPN specification: Algebraic Data Types (ADT for short) and Classes. ADT correspond to data structures defined by algebraic sorts and operations. The equational algebraic definition of the data type's operations allow us to perform the unfolding of the behavior of those operations in a way that is useful for test generation, as we will describe in this paper. CO-OPN's Classes are relatively compatible with the popular notion of Class in the Object Oriented paradigm.

2.1 Algebraic Data Types
ADT are an instance of the well known notion of algebraic specifications (interested readers are directed to [13]). An ADT module includes generators and operations. Generators build the set of elements (the sort) of the ADT, while operations are functions over that set. The behavior of the operations is defined by algebraic equations.

Coming back to the Banking example, we can find in figure 2 a partial definition of an ADT for the Banking specification. The ADT defines the sort password, which is the set of passwords the Banking system allows. In the figure it is possible to see that the single generator of the sort is called newPassword and includes four elements of sort digit as parameter (the sort digit – numbers from 0 to 9 – is defined in a separate ADT module). For the password Sort only the equality ("=") operation is defined.

2.2 Classes
In figure 3 we have represented the class model of our Banking system. The diagram models a system that contains zero or multiple instances of class BankingUser. A BankingUser object holds the user's current state (not logged / waiting for password reply / logged / blocked) and a number of accounts. The user's current state is held in one instance of the eUserState class – the prefix 1 indicates only one instance exists per instance of BankingUser. Each account is an instance of class Account and a user can own one or multiple accounts.

2.2.1 Class Interface – Methods and Gates
In CO-OPN an instance of a class is seen as an entity that can require and/or provide events inside a network of objects. Provided events are called methods and required events are called gates. Both method and gate events may be parameterized.

Figure 4 depicts a graphical representation of a simplified Account class of our Banking system. Methods and gates are represented respectively by black and white rectangles on the outside of the class. An object of type Account provides (or is able to respond to) three outside events: balance?, withdraw amount: integer (meaning withdraw takes an amount of money as parameter), deposit amount: integer and checkAccId accId : string. An account is also able to produce one event to the outside: the hasBalance : integer gate outputs the amount of money present in the account. In this particular case, the hasBalance : integer event is produced as a response to the balance? one.

The method init accId : String corresponds to the initialize method of the class. Although all classes in CO-OPN have an implicit create method, in this case init is used in order to initialize the identifier of a particular account.

2.2.2 Petri Nets for Behavior
The state of a CO-OPN object is held in a petri net. For readers who are unfamiliar with the formalism, Petri nets are a means for representing the behavior of concurrent systems. In a Petri net two concepts are fundamental: places that hold resources and transitions that can consume and produce resources. Newly produced resources are again placed on the net. Typically places are represented by circles, transitions by solid rectangles and their interactions by arcs.

A CO-OPN class can be seen as an encapsulation of a petri net with methods and gates. In figure 4 the places of the petri net are represented by the circles labeled balance and accountId. In this particular case the existing transitions are implicitly associated to methods and gates of the class. For example, the withdraw amount : integer method, when activated, takes a resource bal (implicitly meaning an inte-
ger representing the balance of the account) from the place balance and puts it back subtracting the amount that was withdrawn. In order for this event to happen the existing balance has to be superior or equal to the amount of money withdrawn – as the label in arc from the withdraw method to the balance place indicates. Another interesting example is the balance? method that checks the existing balance (without changing it) and activates the gate hasBalance :: integer² with the available balance as parameter. We provide in the following lines the textual representation (the so called axioms) of the withdraw and balance? methods depicted graphically in figure 4.

(b >= amount) = true => withdraw amount :: balance b -> balance b - amount
balance? with hasBalance b :: balance b -> balance b

Briefly, the first axiom for the withdraw method is split into three parts: the condition "(b >= amount) = true", the method (and parameter) withdraw amount and the necessary pre- and post-conditions of the object’s petri net balance b → balance b - amount. The second axiom for the balance? method has a slightly different form in the sense that there is no condition and when the transition balance b → balance b is possible, the balance? method synchronizes with the gate hasBalance b in order to output the account balance.

It is important to mention that due to the usage of Petri Nets as a way of expressing behavior, the concurrency in a CO-OPN model is managed "for free". In fact, although places in a CO-OPN class can be loosely associated to the traditional notion of class attributes, its semantics is very different. The consumption of a resource in a place means the resource no longer exists – this makes it impossible, for example, for two simultaneous calls to the same method to succeed if they access the same resource.

2.2.3 Object Coordination Model and Transactional Semantics

Apart from the underlying semantics of Petri Nets, the CO-OPN language also employs a coordination model that allows expressing that the execution of a given method requires the simultaneous, sequential or disjunctive occurrence of other events in the object network that composes the system. As an example, let us imagine how we would implement the balance? method that checks the existing balance (without changing it) and activates the gate hasBalance :: integer² with the available balance as parameter. We provide in the following lines the textual representation (the so called axioms) of the withdraw and balance? methods depicted graphically in figure 4.

\[(b \geq amount) = true \Rightarrow withdraw amount :: balance b \rightarrow balance b - amount\]
\[balance? with hasBalance b :: balance b \rightarrow balance b\]

In this axiom we can see that the method withdraw is synchronized with the occurrence of three other events: the user has to be logged, the account has to have the correct identifier and the withdraw method of that account object has to be possible. The ”/” operator states that the events must happen simultaneously. A CO-OPN method call can be also synchronized with a simple event or with an event sequence or disjunction – represented respectively by the ”...” and ”+” operators.

The execution of any CO-OPN method (synchronized or not) is transactional in the sense that it is either fully executed or the model’s state does not change.

3. CO-OPN, SUT AND TEST GENERATION

By choosing CO-OPN as a specification language for model-based testing we naturally include into the set of SUTs we can test concurrent, non-deterministic and transactional systems. This section provides a discussion on how SUTs with these features have influenced our choices while designing SATEL and while producing CO-OPN specifications that allow model-based testing.

3.1 Stimulations and Observations

Test cases are in principle sequences of stimulation/observation pairs over the SUT’s interface signature. Given that we are performing model-based testing, we require an isomorphism to exist between the model’s interface and the SUT’s interface. This is an essential assumption of the approach without which it becomes impossible to use the generated tests. In terms of the CO-OPN model we can map the stimulations of the SUT into synchronizations (including the ”/”, ”...” and ”+” operators) of method events and observations of the SUT into synchronizations of gate events. This formalization of stimulations and observations has the advantage of being straightforward. On the other hand we entirely leave the complexity of applying stimulations and calculating the test verdicts up to the test driver machinery.

3.2 Non-Determinism and Test Representation Formalism

Taking into consideration that test cases are execution traces of the SUT, their natural representation is as sequences of events (or stimulation/observation pairs, as we have previously defined). Simple temporal logics such as Traces [14] can describe such executions in a model, but are insufficient to discriminate different non-deterministic behaviors. Given that CO-OPN allows non-determinism (we can for example declare two different methods with the same fire condition) we have chosen as test representation formalism the HML (Hennessy Milner Logic) temporal branching logic which includes and not operators. The unit event in our HML formulas is the stimulation/observation pair. Leaving the formal equivalence relation issues outside the scope of this paper [15], HML allows us to be precise enough to test accurately non-deterministic aspects of the SUT.

As we have previously mentioned in the paper, we consider both valid and invalid behaviors of the SUT as test cases. In that sense we need to add to the HML formulas a logic value true or false in order to distinguish valid from invalid behaviors. Our test cases are thus pairs of the form \(f, r\), where \(f\) is an HML formula with stimulation/observation pairs as events, and \(r \in \{true, false\}\).

3.3 Transactional issues of CO-OPN

CO-OPN’s transactional semantics makes it possible to automatically reject certain operations if the state of the
model does not allow executing them. For example, in figure 4, the \textit{withdraw amount: integer} operation (or method) is only successful if the condition \((b \geq \text{amount}) = \text{true}\). In case the state of the model is such that this condition is \text{false}, the model will simply refuse the execution of the operation. These semantics are interesting for modeling because they allow treating operations in a positive way – an operation is executed if and only if the state of the model allows it, otherwise nothing happens. All error situations and inconsistent states are thus avoided. However, most programming languages do not implement these semantics and real SUTs usually react to (distinct) impossible operations with (distinct) error codes. Methodologically speaking it is thus important to model those impossible situations as observable events of the system. Coming back to the model in figure 4, this means it would be interesting to add a second behavior for the \textit{withdraw amount: integer} method, such as:
\[
(b \geq \text{amount}) = \text{false} \Rightarrow \text{withdraw amount} \\
\text{// errorLowBalance : balance } b \rightarrow \text{balance } b;
\]
This axiom states that if we try to withdraw an amount of money superior to the balance a gate event \textit{errorLowBalance} would occur. Given this new operation it would be possible to associate a stimulation \textit{withdraw amount} (where \text{amount} is superior to the current balance of the account) with an observation \textit{errorLowBalance} in order to test this behavior.

4. SYNTAX AND SEMANTICS OF SATEL

SATEL is a language for expressing test intentions for CO-OPN specifications. The language should be precise enough to tackle in depth the model the test engineer wishes to produce tests from, but at the same time generic and simple enough to accomplish it without exposing the complexity of the test generation engine itself. Given that different test intentions can cover different functionalities expressed in the model (we have relaxed the need for pertinence of the test set as formalized in [16]), we have also designed the language in a way that test intentions can be reused. In fact we have defined test intentions as modules that may be composed, giving rise to the possibility of devising in the future a methodology for testing systems in a compositional way, possibly reusing previously defined test intentions. A formal description of the syntax and semantics of SATEL may be found in [17].

4.1 Syntax of SATEL

A test intention is written as a set of HML formulas with variables, which in the subsequent text we will call execution patterns. The variables correspond to the three dimensions of a test case, namely:

- the shape of the execution paths;
- the shape of each stimulation/observation pair inside a path;
- the algebraic parameters of the methods or gates inside the stimulation/observation pairs.

A test intention is thus written as a set of partially instantiated execution patterns, where the variables present in those patterns are by default universally quantified. All the combined instantiations of the variables will produce a (possibly infinite) number of test cases.

By constraining the domains of the variables in an execution pattern, the test engineer is able to produce test cases that accurately reflect his/her intuition behind a test intention. For each kind of domain we have devised a number of functions and predicates that allow modeling test intentions. The functions have as co-domains the \textit{integers} and \textit{booleans}, which are native data types of SATEL. The predicates are the typical binary predicates for \textit{integers (==,\langle,,\rangle,,\langle,\rangle,\geq,,\rangle,} and for \textit{booleans (==,\langle,\rangle,)}.

- Variables over the shape of execution paths: these variables are constrained by using functions that discriminate the of the shape HML formulas. In particular we have implemented the following functions that have HML formulas as domain: \textit{nbEvents} – number of events in an HML formula; \textit{depth} – length of the deepest branch of an HML formula; \textit{sequence} – \text{true} if the HML formula contains \text{no} and operators; \textit{positive} – \text{true} if the HML formula contains \text{no} not operators; \textit{trace} – \text{true} if the HML formula contains \text{no} and or not operators.

- Variables over Stimulations/Observations: given that stimulations and observations are respectively synchronizations of method and gate events, we have devised a number of functions that discriminate the shape of those synchronizations. In particular we have implemented the following functions that have stimulations or observations as parameters: \textit{simpleEvent} – \text{true} if the stimulation or observation is composed respectively of only one method or gate call; \textit{onlySynchroniztion}, \textit{onlySequence}, \textit{onlyAlternative} – \text{true} if there are only respectively simultaneity (\text{"/\")}, sequence (\text{"\...\") or alternative (\text{"+\") operators present in the synchronization \textit{nbSynchronizations} – number of simple events in the synchronization.

- Variables over algebraic parameters of methods or gates: given the fact that these variables represent values of algebraically defined sets, we use algebraic equations in order to limit the elements from those sets. If we take the example of figure 2, a possible algebraic constraint would be \((a = \text{newChallenge } 1 2 3 4) = \text{true}\), which would limit an algebraic variable called \(a\) to the only possible value of \text{newChallenge1} 1 2 3 4).

4.1.1 Declaring test intention rules

Each test intention may be given by several rules, each rule having the form:
\[
\text{[condition} \Rightarrow \text{] inclusion}
\]
In the \text{condition} part of the rule (which is optional) the test engineer is able to express constraints over variables – those mentioned in the above list of items. In the \text{inclusion} part of the rule the test engineer can express that a given execution pattern is included in a named test intention. Assuming given a CO-OPN specification, consider the following rule (where \(f\) is variable over execution paths):
\[
\text{nbEvents}(f) < 5 \Rightarrow f \text{ in SomeIntention}
\]
This rule would produce all possible test cases for that specification that include a number of events inferior to 5. These test cases would become part of the test set generated by the test intention SomeIntention.

On the other hand it is possible to declare multiple rules for the same test intention. Let us add to the previous rule the following one, where aMethod and aGate are respectively ground stimulations and observations:

\[
\text{HML}\{\langle a\text{Method}, a\text{Gate}\rangle \ T\} \text{ in SomeIntention}
\]

The set of test cases produced by SomeIntention would now become the one produced by the first rule united with the one produced by the second rule. In fact only one test case is produced by the second rule given that there are no variables in the execution pattern \(\text{HML}\{\langle a\text{Method}, a\text{Gate}\rangle \ T\}\).

An interesting feature of the language is that it allows reusing rules by composition, as well as recursion between rules. Consider the following set of rules where \(f\) and \(g\) are variables over the shape of execution paths:

\[
\text{HML}\{\langle a\text{Method}, a\text{Gate}\rangle \ T\} \text{ in AnotherIntention}
\]

\[
\text{HML}\{\langle a\text{Method}', a\text{Gate}'\rangle \ T\} \text{ in AnotherIntention}
\]

\[
f \text{ in AnotherIntention}, g \text{ in AnotherIntention} \Rightarrow f . g \text{ in AnotherIntention}
\]

These rules would produce an infinite amount of test cases which include sequences of the stimulation/observation pairs \(\langle a\text{Method}, a\text{Gate}\rangle\) and \(\langle a\text{Method}', a\text{Gate}'\rangle\) in any order and in any length. In fact, the third rule for AnotherIntention chooses non-deterministically any two test cases generated by any rule of the test intention and builds a new test case based on their concatenation\(^3\).

The composition of test intentions is a very important feature given that it allows establishing a methodology for building test intentions that cover progressively larger parts of the SUT. We are currently investigating these methodological issues and their impact on how to write test intentions. A direction for this research is that, using SATEL, top down or bottom up construction of test cases is possible.

### 4.1.2 Variable quantification constraints

All the variables used in a test intention rule are universally quantified, while satisfying the constraints expressed in the condition of that rule. This will produce test sets which are the combination of all the possible values the variables assume in the execution patterns on the right side of the rule. In some cases this is not practical because we may want to select randomly a value from a given domain – this corresponds to uniformity hypothesis as described in [16], whereas the previously presented constraints correspond rather to regularity hypothesis.

In order to deal with this random aspect necessary for test generation, we have included in SATEL two unary predicates that may be seen as quantifiers for variables of our language. These quantifiers may be applied to any variable of the language (shape of execution paths, shape of synchronizations or algebraic parameters of methods). However, they will only quantify directly the variables which represent the algebraic parameters of methods. In what concerns the remaining variable types, they will be quantified indirectly in the sense that the quantification will propagate to all the method parameters included in the execution patterns that those variables stand for.

- **uniformity(var)**: all method parameters directly or indirectly in the scope of var will be attributed one random value;
- **subuniformity(var)**: all method parameters directly or indirectly in the scope of var will be attributed one random value for each of the equivalence classes in the semantics of the method they belong to.

As an example, consider the following rule for the test intention TestWithdraw where \(\text{amount}\) is a variable of the ADT integer type and \(g\) is a variable of the type observation:

\[
\text{subuniformity(\text{amount})} \Rightarrow
\text{HML}\{\langle \text{deposit}(10), \text{null}\rangle, \langle \text{withdraw(\text{amount})}, g \rangle \ T\} \text{ in TestWithdraw}
\]

Using an object of type Account as specification (see figure 4), this test intention would generate for example the following valid test cases:

\[
\langle \text{deposit}(10), \text{null}\rangle, \langle \text{withdraw}(5), \text{null}\rangle, \text{true}
\]

\[
\langle \text{deposit}(10), \text{null}\rangle, \langle \text{withdraw}(15), \text{error\text{LowBalance}}\rangle, \text{true}
\]

In fact the subuniformity predicate allows choosing two values for the \(\text{amount}\) variable, one for each fire condition of the method withdraw. We have defined two axioms for withdraw with the complementary conditions \((b \geq \text{amount}) = \text{true}\) and \((b \geq \text{amount}) = \text{false}\). Operationally we choose one value satisfying each of those conditions, hence covering the two equivalence classes of withdraw.

### 4.2 Semantics of SATEL

The abstract semantics of SATEL corresponds to three consecutive steps:

1. Expansion of the test patterns defined in the test intentions by instantiating the variables to their possible values. All variables are instantiated except for the ones marked with the subuniformity quantifier and the observation variables. All the combinations of all instantiations yield a first batch of partially HML formulas;

2. For all HML formulas generated in step 1, instantiation of the variables marked with the subuniformity predicate and the observation variables. The instantiation of these variables provide the oracles for those formulas.

3. Checking of the validity of the HML formulas produced in step 2 w.r.t. the CO-OPN specification.

This abstract view of the semantics is not tractable operationally. In fact, to compute the semantics of SATEL’s test intentions we use logic programming (which mixes the three steps). In order to instantiate the remaining variables from step 1, we have built a translator of CO-OPN specifications to the logic programming language Prolog [18]. By computing the Prolog translated specification with the partially

\(^3\)The \(\cdot\cdot\cdot\) is the concatenation function between test patterns.
instantiated HML formulas, we are able to fully instantiate them. In particular, to calculate the variables marked with the subuniformity quantifier we make use of the unfolding technique explained in [19]. The technique involves marking the Prolog translation of the equations specifying each algebraic operation, at certain choice points. These choice points correspond to goals in Prolog for which the solutions will be values inside the equivalences classes of those operations. For an example of application of the unfolding technique, see [20].

We decide of the validity or invalidity of an HML formula by checking its satisfaction in the CO-OPN specification. The Prolog computation of the specification will only instantiate variables in step 2 that lead valid behaviors according to the specification. However, in step 1 we can already have invalid behaviors given that the test intention rules are defined by the test engineer and may include any sequence of stimulation/observation events. In order to decide about the satisfaction of an HML formula in a specification we proceed in the following way:

- the HML formula is true if the stimulation/observation pairs that compose the formula can be either computed in the Prolog translation of the specification or instantiated sequentially through all the branches until the end of the formula;
- the HML formula is false if some stimulation/observation pair of the formula is fully instantiated but cannot be computed in the Prolog translation of the specification. In that case we discard the remaining of the formula after that stimulation/observation pair.

5. CASE STUDY – TESTING THE BANKING SYSTEM

In figure 5 we provide a full example of usage of our test intention language for defining a test intentions module for the Banking system. This example is written in the concrete syntax of the language which we have implemented in a toolset for experimentation.

While writing the test intentions we have assumed that for each stimulation of the Banking CO-OPN model there are two possible observations: the positive one means the operation was successful and starts with prefix OK; the negative one starts with prefix error.

The module TestBanking acts over Class Banking (as defined in the Focus field) and defines several distinct test intentions declared in the fields Intentions. In figure 5 the variable names are presented in bold font in order to make explicit where the test patterns will be expanded. The types for those variables are declared in the Variables field.

- **axiom 1** is composed of only one stimulation/observation pair with two variables, usr for the user login and obs for the observed output. When instantiating usr and obs we get all valid and invalid behaviors of the login operation for all existing users. However, the subuniformity predicate allows making use of the semantics of the model by performing equivalence analysis and picking only two values for the usr variable

\[ \text{axiom 1} \]

[Figure 5: Test Intentions for the Banking SUT]

- one for the case where the user exists and one for when the user doesn’t exist. Four test cases are then generated by axiom 1: the user exists and gets logged in (valid test case); the user doesn’t exist and the SUT issues an error code (valid test case); the user exists but the SUT issues an error code (invalid test case); the user doesn’t exist but the SUT allows him/her to log in (invalid test case).

- **axioms 2** makes use of recursion in order to build a sequence of erroneous introductions of passwords by user “mario”. We assume a user “mario” exists in the SUT and his password is (newPassword 1 2 3 4);
- **axiom 3** uses the previously defined pattern nWrongPins in order to build all possible test cases for behaviors of the loginUserPassword operation. Note the usage of the \( nbEvents \) predicate to constrain the number of stimulation/observation pairs in the sequence of erroneous logins;
- **axiom 4** produces test for the behaviors of the withdraw operation. Variable \( f \) of type HML (in the simplest case, an execution trace) is not instantiated, which means the compiler should generate all possible paths in order for the SUT to reach a state where a deposit operation can be executed. Afterwards the subuniformity predicate over variable \( am \) will find two values for this variable: one under 100 (the amount of the deposit) and another over 100;
- **axiom 5** produces tests for the simultaneous login of two users.
6. CONCLUSION

In this paper we have explored CO-OPN as a modeling language for model-based test case generation. We started by analyzing the CO-OPN and discussing its properties. Afterwards, we have presented the language SATEL which is suitable for producing test cases in a semi-automatic fashion. In the paper we have applied SATEL to a hypothetical Banking system and the results point to a balanced level of abstraction of the language. We are currently validating the approach with an industrial partner and should have reach clearer conclusions in a near future. As future work we plan on broadening the scope of SATEL to other modeling formalisms (e.g., Statecharts), as well as investigating and proposing a methodology for test intention design and reuse.

7. REFERENCES