A Test Selection Language for CO-OPN Specifications

Levi Lúcio, Luis Pedro, Didier Buchs
University of Geneva
Centre Universitaire d'Informatique
24, rue du Général-Dufour CH-1211 Genève 4, Switzerland
Levi.Lucio,Luis.Pedro,Didier.Buchs@cui.unige.ch

Abstract

In this paper we propose a test language that allows expressing test intentions for CO-OPN (Concurrent Object-Oriented Petri Nets) specifications - a formal specification language designed to handle large complex concurrent systems.

Our test language is based on temporal logic formulas for expressing graphs of input/output pairs - the inputs correspond to operations performed on the system and the outputs to the observable results of those operations. We encapsulate the temporal logic using a language of constraints, which purpose is to shape the tests that are to be produced. In this paper we discuss the syntax and provide the semantics of this test language. One of our main worries while designing the test language were to keep it modular in order to promote reusability. Another worry was to be able to cope with non-determinism coming from the system under test. We illustrate managing non-determinism as well as other features of our language by showing how we can generate tests for the login part of an e-banking system.

A framework for editing CO-OPN specifications exists and one of its features is the possibility of automatically generating high level Java prototypes that can be completed/extended by human developers. We discuss the applicability and the usefulness of our test language while verifying systems built using this methodology.

1. Introduction

CO-OPN (Concurrent Object-Oriented Petri Nets) is a formal specification language built to allow the expression of models of complex concurrent systems. Its semantics is formally defined in [1], making it a precise tool not only for modeling, but also for prototyping and test generation. CO-OPN's richness gives us the possibility to specify in a formal fashion models of the systems we will further use as the System Under Test (SUT). 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.

Our development approach encompasses three steps: *Analysis – Prototyping – Implementation*. After the first phase of analysis we get a specification model in CO-OPN. This model will be used to test (meaning test generation and verification) the fruit of the implementation period - what we call the System Under Test (SUT).

Using CO-OPN we can profit from the advantages of having the system specified in a formal language: the unambiguous representation of the system; the easy reusability based on the fact that, usually, formal expressions are mathematically based and by definition independent of the time; also the fact that testability and verification are more precise as more independent of the context. As such, CO-OPN is the basis for our test selection language.

1.1. The test generation process

Our test generation process is based on the fact that we assume having a correct specification of the SUT. This specification can be compared with the SUT in order to track discrepancies which will indicate errors (see [6] for details on the subject). This comparison can be achieved by means of tests, which consist of graphs of input/output pairs validated by the specification. If a test is a valid behavior of the specification, it should also be a valid behavior of the SUT. Conversely, a test that describes an invalid behavior of the specification should not be validated by the SUT. An adequate formalism must be chosen in order to guarantee the correctness of the test process w.r.t. the implementation relation. However this subject is out if the scope of this paper and we direct the interested reader to [7].

The main problem to cope with, while following this approach, lies in the fact that, in the general case, it is impossible to test all the possible behaviors of an SUT. Imagine an operation over the SUT that takes in a parameter from

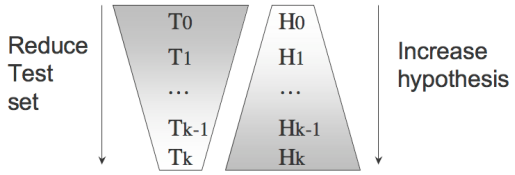

Figure 1. Test set reduction by hypotheses' application

an infinite domain (e.g. natural numbers): a set of tests that would try all possible calls to that operation would be infinite. The same could be said about trying all the possible behaviors of an operation that loops. For example, how does one know if after 100 deposits using an e-banking system, one's account will not be emptied? It is impossible to assume with complete certainty the operation is correctly implemented unless the loop is executed an infinite amount of times.

Given the above, in the general case the test set that would cover all the possible behaviors of an SUT – the *exhaustive test set* – would be infinite. Since it is impossible to apply an infinite test set to an SUT in finite time, it is necessary to reduce the exhaustive test set to a finite one, while keeping pertinence. A pertinent test set is one that is: *valid*, meaning all correct SUTs satisfy the test set; *unbiased*, meaning only correct SUTs satisfy the test set.

Our reduction technique consists of stating hypotheses about the functioning of the SUT (inspired from the work of [4] and [3]. These hypotheses are generalizations of the behavior of the SUT. A possible generalization would be *"if the e-banking system logs in correctly a set of three different users, then it logs in correctly all users"*. Another would be *"if the e-banking system performs 10 deposits correctly, then all deposits are performed correctly"*.

Figure 1 is meant to be read from top to bottom and depicts the reduction of the exhaustive test set T_0 to a T_k one that is finite and pertinent. This is done by stating hypotheses H_0 through H_k about the SUT. The assumption that the test set T_k is pertinent only holds if the hypotheses stated are actually satisfied by the SUT.

In [5] we have discussed extensively the problem of deciding whether a test generated using this approach is valid or invalid according to the specification. We have also discussed how we apply valid or invalid tests to an SUT and check their satisfaction. In the present paper our interest is focused on the syntactic generation of tests that obey the constraints posed by hypotheses on the SUT. We will thus describe a constraint language for CO-OPN specifications that allows a test engineer to express these hypotheses.

2. Requirements

In order to define the test selection language that will allow expressing hypotheses about the SUT we need to establish the requirements for this language.

Given we want to test CO-OPN specifications, one of the requirements is that the structure of the language follows closely the philosophy of the CO-OPN specification language. Another significant issue is the fact that we should be able to integrate the test language in a development environment and in particular to be able to easily interface with development tools that allow fast iterations in the *prototyping − testing − prototyping* process. Since CO-OPN provides an infrastructure for automatic code generation [2] (e.g. Java), it will allow us to integration out test generation language in a development environment.

We would also like to be able to cope with a certain degree of non-determinism coming from the SUT, as well as to promote reusability of tests we express in our language.

2.1. Philosophy of the CO-OPN specification language

CO-OPN is a specification language suitable to specify complex reactive software which proprieties and interaction with the environment need to be captured. 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 object oriented concepts for system specification. The specifications are then a collection of ADT, classes and context modules. The interaction with the environment is produced by CO-OPN coordination modules which are in fact the classes and context.

Generally, each module encompasses, syntactically speaking, the following structure: *Name* - which represents the name of the module; *Interface* - that mainly comprises the types and elements accessible to the outside; *Body* that includes the internal (private) information of the module. In addition, the *Body* section includes different parts like *Axioms* - operations' proprieties expressed by means of equations; *Use* field that indicates the list of dependencies from other modules.

2.2. Test reusability

We need a way to define for each specification module basic constraints that can be reused in other constraint modules - that can be associated to the composition of specification modules. Naturally, constraints can be glued using a conjunctive semantics (constraint refinements) and also a kind of shuffle semantics (for union specification composition). The hierarchy of constraints will not follow the specification module hierarchy but mainly reflect the testing

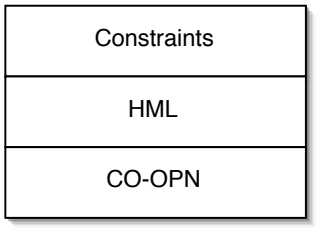

Figure 2. The three layers of the test selection language

process strategy. It thus seems important to be able to develop some kind of "design for testing" and consequently to provide testing modules during the specification design process. Integration testing will provide modules that glue basic test patterns in order to form an adapted testing strategy. This is why basic constraints must reflect not only usable tests but pattern of properties that will serve in the elaboration of integration testing selection.

2.3. Dealing with non-deterministic SUTs

An SUT may respond non-deterministically to a given input. Let us introduce an example: it is common practice for banks to provide their e-banking clients paper cards displaying a grid of strings indexed by a coordinate. These cards are used for authentication while logging into the e-banking service since at some point the system asks for the string corresponding to a randomly generated coordinate.

Imagine one would like to generate a test that would verify that by introducing the string corresponding to the proposed coordinate the user would succeed that login step. If a SUT always responds deterministically to inputs, it is possible to calculate in advance the full graph of input/output pairs that make up a test. In the present e-banking login example, we do not know which pair of coordinates the system is going to propose. To make the login step succeed we would then have to calculate the reply to the proposed coordinate not during test generation time, but rather in testing time – while the test is being applied to the SUT. This implies building partially instantiated tests.

3. Syntax and Semantics

The test selection language we propose is composed of three layers, namely *CO-OPN*, *HML* (Hennessy-Milner Logic) and *Constraints* (see figure 2). Informally, the purpose of each layer is the following:

- *CO-OPN:* to describe input/output pairs over a CO-OPN specification of the SUT. In an input/output pair the input corresponds to a CO-OPN method call and the output corresponds to the message resulting from that method call;
- *HML:* this elementary temporal logic allows the expression of graphs of events. For our testing purposes these events will consist of input/output pairs over a CO-OPN specification. A test can be seen as a ground HML formula;
- *Constraints:* using the Constraints layer we are able to specify high level hypotheses about the functioning of the SUT. These hypotheses are expressed as constraints over HML formulas.

In the following paragraphs we will describe the abstract syntax and the semantics of *HML* and *Constraints*. These semantics will be provided in an informal fashion.

In what concerns the *CO-OPN* level we will also mention that method calls as well as the results produced by them can include parameters. Our language includes variables over an events' input, output or their respective parameters.

3.1. HML

HML_{SP} stands for the language of HML formulas over a given CO-OPN specification SP. In the following definition of the abstract syntax of HML_{SP}, T represents the always true constant and $Event(SP)$ is the set containing all the input/output pairs over SP.

- $T \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\neg f) \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (f \wedge g) \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (< e > f) \in HML_{SP}$
  where $e \in Event_{SP}$

The semantics of HML_{SP} can be defined in terms of the satisfaction relation between HML_{SP} formulas and the transition system denoted by specification SP. This transition system is a quadruple $\langle Q, Event(SP), \rightarrow, i\rangle$ where Q is the set of all states in SP, $\rightarrow$ is a function[1]with signature $Q \times Event_{SP} \rightarrow Q$ and i is an the initial state of the transition system. Given a state $q \in Q$:

- T is always satisfied by specification SP starting from state q;
- $(\neg f)$ is satisfied by specification SP starting from state q, if f is not satisfied by SP starting from state q;

[1]We define $\rightarrow$ as a function in order to avoid internal non-determinism not directly observable through the events.

- $f \wedge g$ is satisfied by specification SP starting from state q if f is satisfied by SP starting from state q and g is satisfied by SP starting from state q;

- $< e > f$ (read event e followed by HML formula f) is satisfied by SP if starting for a state $q \in Q$ there is an event $e \in Event_{SP}$ leading to $q' \in Q$ and f is satisfied by SP starting from state q'.

In the concrete syntax all HML formulas will be preceded by the keyword "HML" and the symbols "$\neg$" and "$\wedge$" will be replaced by "not" and "and", respectively. An example of an HML formula over an hypothetical CO-OPN specification for an e-banking SUT would be:

```
HML <userName("smith"),loginOK>
       <password("my_pass"),passOK> T
```

This HML formula is a test that consists of a simple sequence of two events. In the first one the user inserts a login *"smith"* and gets a message saying the login is correct. In the second the user introduces a password *"my_pass"* and the system replies saying it is the correct password.

Operators $\neg$ and $\wedge$ are used for discriminating differences in non-deterministic SUTs, although we will not explore this topic in this paper. Again, the interested reader is pointed to [7].

3.2. Constraints

In our theory of testing, hypotheses about the functioning of the SUT may be described as constraints over the *exaustive test set*. Since we describe tests in terms of HML formulas we need a language with which we can constraint HML formulas. From here on in this paper we will employ the term *constraint* to mean a constraint over HML formulas.

Figure 3 depicts the high level concrete syntax of our constraint language. Following the philosophy of the CO-OPN specification language, we have decided to define the constraint language at two syntactic levels: a high level including all the sections necessary to define a constraint module – a *ConstraintSet*; a low level where the constraints themselves are built. We start by describing the purpose of the sections of the high level syntax:

- *Interface:* the interface defines the constraints that are exported from the module and that can be used (composed with others) to build test sets. It includes only one section *Constraints* where the names of the exported constraints are declared;

- *Body:* the body declares the properties necessary to the construction of constraints. It includes five sections:

```
ConstraintSet Name;
   Interface
      Constraints
         ...
   Body
      Constraints
         ...
      Use
         ...
      Axioms
         ...
      Variables
         ...
      External
         ...
End Name;
```

Figure 3. High level structure of a ConstraintSet module

  - *Contraints:* declares the constraints defined locally to help in the construction of the exported constraints. They are not exported from the module;
  - *Use:* declares constraints that are imported from other *ConstraintSet* modules;
  - *Axioms:* declares axioms and rules that establish constraints as sets of HML formulas;
  - *Variables:* establishes the type of the variables used in the definitions of the *Axioms* section;
  - *External:* declares functions used in HML formulas during testing time (as opposed to test generation time). The purpose of these functions is to calculate values over of non-deterministic outputs of the SUT.

In figure 4 we present the grammar that produces the abstract syntax of the *Axioms* subsection of a *ConstraintSet* module. Non-terminal symbols are written in normal font and terminal ones are written in bold face.

From the remaining sections of a *ConstraintSet* module all but the *Variables* one are composed of lists of names. The *Variables* section consists of a set of variable declarations of the form:

```
v1, v2, ... vn : type
```

The language in figure 4 allows us to express sets of *axioms* or *rules* (production *axiomSet* of the grammar) which define constraints – in other words, sets of HML formulas. Each axiom or rule has the form:

```
axiomSet ::= axiom axiomSet
           | axiom
           | ϵ

axiom ::= condition => assignement
        | assignement

assignement ::= name in formulaSet
              | [] in formulaSet

formulaSet ::= hmlFormula . formulaSet
             | name . formulaSet
             | hmlFormula
             | name

condition ::= atom & condition
            | atom
            | ϵ

atom ::= assignement
       | logicalOperation
       | uniformity(name)
       | subuniformity(name)

logicalOperation ::= expression = expression
                   | expression != expression
                   | ... repeated for <,>,<=,>=

expression ::= term + expression
             | term - expression
             | ... repeated for *,/
             | term

term ::= boolean
       | name
       | nbEvent(formulaSet)
       | depth(formulaSet)
       | nbOccurence(formulaSet,name)
       | onlyConstructor(formulaSet)
       | onlyMutator(formulaSet)
       | onlyObserver(formulaSet)
       | sequence(formulaSet)
       | positive(formulaSet)
       | trace(formulaSet)

boolean ::= true | false

name ::= ... any alphanumeric string started
         by a letter ...
```

Figure 4. Abstract syntax of subsection Axioms

```
[ condition => ] assignment
```

An axiom corresponds to an *assignment*. An assignment allows including an HML formula into a constraint. For example, assuming f is declared in the *Variables* section as being of type HML and *EnterLogin* is declared as a constraint is the *Constraints* section, the following assignment (or axiom) would be valid:

```
f in EnterLogin
```

A *rule* includes a *condition* and an *assignment*. The condition consists of a conjunction of *atoms* which shape the HML formulas that are used on the *assignment* part of the rule. Axioms and rules allow recursive definitions of constraints. Consider for example:

```
[] in LoginWrong
f in LoginWrong =>
    f.HML⟨login("undef_user"),LoginError⟩T ∈ LoginFail
```

With the axiom in the first line of the definition and the rule spanning over the second and third lines we are able to establish recursively the *LoginWrong* set of all HML formulas which are sequences of the event ⟨`login("undef_user"),LoginError`⟩. The [] symbol represents the empty HML formula.

As can be seen in figure 4 a *condition* is made out of *atoms* which can be *assignements*, *logicalOperations* , emphuniformity or *subuniformity* predicates. *Uniformity* predicates act on variables defined at the CO-OPN level, as parameters of the input part of an event. Consider the following rule:

```
uniformity(x) => ⟨login(x),LoginOK⟩ in TestLogin
```

x is a variable defined at the CO-OPN level and corresponds to any username that is available in the specification while generating tests. The *uniformity* predicate selects only one value from the domain of the variable, which means that the *TestLogin* constraint will contain only one HML formula where the value for x is chosen randomly from the available values in the specification. In terms of hypotheses about the SUT this means that we rely on the fact that testing the success of the *login* operation with only one user is enough. Consider now the rule:

```
subuniformity(x) => ⟨login(x),y⟩ in TestLogin
```

The rule is similar to the one before, but we have also defined a new CO-OPN variable y corresponding to any output of the login operation (in our specification either *LoginOK* or *LoginWrong*). In this case the *subuniformity* predicate will decompose the behavior of the *login* operation according to its possible outputs and will instantiate x to a set of values, one (randomly chosen) per possible output. This said, the rule would put in the *TestLogin* constraint two HML formulas, one with a username that exists and a *LoginOK* output and a second one with a username that does not exist and a *LoginError* output.

Regarding *logicalOperations*, they allow imposing additional constraints on HML formulas. For example, the following:

```
[] in LoginWrong
f in LoginWrong & nbEvent(f) <= 5 =>
    f.HML⟨login("undef_user"),LoginError⟩T ∈ LoginFail
```

would generate in *LoginWrong* all sequences of ⟨`login("undef_user"),LoginError`⟩ events with at most five events.

An *expression* can be of type *Natural* (set of all numbers) or *Boolean* (true or false). For example:

```
f in OldConstraint & positive(f)=false =>
                        f in NewConstraint
```

chooses only formulas from the previously defined constraint *OldConstraint* which are negative (meaning they contain ”not”). The following predicates are also available in our language:

- *depth:* number of levels of imbricated ”and” or ”not” operators in an HML formula;
- *nbOcurrence:* number of ocurrences of a given event in an HML formula;
- *onlyConstructor:* true if all method calls in an HML formula are class constructors;
- *onlyMutator:* true if all methods calls in an HML formula modify classes attributes;
- *onlyObserver:* true if all method calls in an HML formula do not modify classes attributes;
- *sequence:* true if there are no ”and” operators in an HML formula;
- *trace:* true if there are no ”and” and ”or” operators in an HML formula.

As we have mentioned previously, the *Variables* section of a *ConstraintSet* defines the types of the variables used in the declarations of the *Axioms* section. The following types are available:

- *Boolean:* true or false;
- *Natural:* set of natural numbers;
- *CO-OPN:* generic type meaning either a parameter or an output of a method call;
- *HML:* HML formulas.

When calculating a constraint defined in a *ConstraintSet* module, all variables in the axioms and rules are instantiated to their possible values so that all HML formulas corresponding to that constraint can be calculated. However, variables of type CO-OPN that are parameters of functions declared *external* are not instantiated and remain free in the produced constraints. The purpose of these semantics is made clear by the example in section 4 of this paper.

4. Case study - the e-banking login system

Again, consider as an example SUT a system that allows a user to login into an e-banking system. The login will be performed in three steps:

1. the user inserts his/her username that is validated against a user database. If the username is correct the system asks for the password;
2. the user inserts his/her password. If correct the system asks for a challenge which is a coordinate in a card the bank has provided the user;
3. the user inserts the string corresponding to the requested challenge and becomes logged into the system.

If a user fails three times the password step or five times the challenge step the e-banking system becomes blocked for 24 hours for that user.

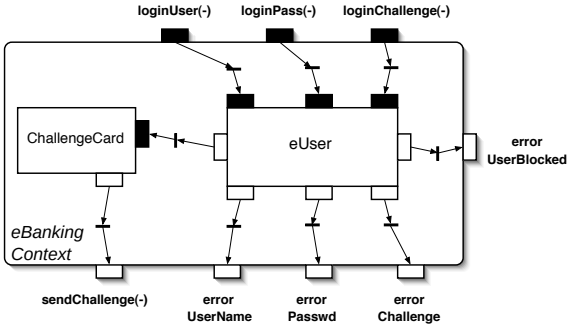

Figure 5. CO-OPN specification for the eBanking sytem

Figure 5 depicts a CO-OPN specification of the SUT. Three methods (the small black boxes on top) are available from the main *e-banking* context – *loginUser* with a username parameter, *loginPass* with a password parameter and *loginChallenge* with a challenge parameter. The system outputs five possible answers though gates (the four small white boxes below): *errorUserName*, *errorPasswd*, *errorChallenge*, *errorUserBlocked* or a *sendChallenge* message with a random challenge parameter. The internal contexts *ChallengeCard* and *eUser* are not used by the test generation mechanism, although they represent components that could be tested individually.

There are several tests we would like to perform on the e-banking login system:

- Is the *loginUser* operation implemented correctly?

- Does the SUT propose a challenge when the password is correct? Does it block after three unsuccessful *loginPass* operations?
- Does the SUT allow the user access to the e-banking system when the reply to the proposed challenge is correct? Does it block after five unsuccessful *loginChallenge* operations?

```
ConstraintSet e-banking;
   Interface
      Constraints
         login
         badPass
         badChal
         challenge
   Body
      Constraints
         nWrongPass
         nWrongChal
      Axioms
1        HML<loginUser(u),r>T in login;
2        []in nWrongPass;
3        f in nWrongPass & nb_event(f)<5 =>
             HML<loginPass("wrong",r)>T in nWrongPass;
4        []in nWrongChal;
5        f in nWrongChal & nb_event(f)<7 =>
             HML<loginChal("wrong",r)>T in nWrongChal;
6        f in login & g in nWrongPass => f.g in badPass
7        f in login & g in nWrongPass
                    & h in nWrongChal =>
             f.g.HML<(loginPass(p),sendChallenge(c)>T in badChal;
8        f in login & g in nWrongPass
                    & h in nWrongChal =>
             f.g.HML<(loginPass(p),sendChallenge(c)>T.
             HML<(loginChal(calc_challenge(c),r> in challenge;
      Variables
         u,r,c,p : CO-OPN
         f,g,h : HML
      External
         calc_challenge
End Name;
```

Figure 6. ConstraintSet module for the e-banking system

Figure 6 shows an e-banking *ConstraintSet*. We have defined four constraints that this module exports:

- *login:* test all behaviors of the *loginUser* operation;
- *badPass:* test behaviors of the *loginPass* operation when wrong passwords are introduced after a correct username. Eventually, the user may become blocked;
- *badChal:* test behaviors of the *loginChallenge* operation when wrong challenges are introduced after a correct username and a correct password. Here also the user may become blocked;
- *challenge:* test the success behavior of a *loginChallenge* operation after a correct username and a correct password. Notice that the *calc_challenge* function is declared as external, which means that its parameters will not be instantiated in the HML formulas produced by the constraint. Since it is not possible to calculate in advance the challenge that the SUT is going to propose, we keep the variable holding it uninstantiated (variable ***c*** in rule 8 of figure 6). We can then apply in testing time the *calc_challenge* function the the actual challenge we find.

5. Conclusions

We have presented a test selection language for CO-OPN specifications by defining its syntax and semantics. We think the language is expressive enough to cope with the main test intentions of a test engineer, while remaining at a level of abstraction that makes it applicable to a wide range of SUTs.

This paper represents a first step in developing this language and we are aware more work needs to done to bring it to a mature state. In particular the problem of test selection reusability needs to be further addressed as well as the connection between our language and the activity of prototyping. Also, we envisage applying our test selection language to more extensive SUTs in order to better understand the language's advantages and limits.

References

[1] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.

[2] S. Chachkov and D. Buchs. From formal specifications to ready-to-use software components: The concurrent object oriented petri net approach. pages 99–110, Newcastle, 2001.

[3] R.-K. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. volume 3(2), pages 101–130, 1994.

[4] M.-C. Gaudel G. Bernot and B. Marre. Software testing based on formal specifications: a theory and a tool. volume 6(6), pages 387–405, 1991.

[5] L.Pedro L.Lucio and D. Buchs. A methodology an a framework for model-based testing. pages 52–63, Luxembourg, 2004.

[6] Levi Lucio and Marko Samer. Technology of test case generation. To be published in LNCS.

[7] Cecile Peraire. *Formal testing of object-oriented software: from the method to the tool*. PhD thesis, EPFL - Switzerland, 1998.