Test Generation Based on Abstraction and Test Purposes to Complement Structural Tests

F. Bouquet, P.-C. Bué, J. Julliand, P.-A. Masson
LIFC - Université de Franche-Comté - 16, route de Gray F-25030 Besançon Cedex, France
Email: {bouquet, bue, julliand, masson}@lifc.univ-fcomte.fr

***Abstract*—This paper presents a computer aided model-based test generation method. We propose this approach as a complement to the LTG (Leirios Test Generator) method, which extracts functional tests out of a formal behavioral model M by means of static (or structural) selection criteria. Our method computes additional tests by applying dynamic (or behavioral) selection criteria (test purposes called TP). Applying TP directly to M is usually not possible for industrial applications due to the huge (possibly infinite) size of their state space. We compute an abstraction A of M by predicate abstraction. We propose a method to define a set of abstraction predicates from information of TP. We generate symbolic tests from A by using TP as a dynamic selection criterion. Then we instantiate them on M, which allows us play the tests on the implementation the same way as we play the functional ones. Our experimental results show that our tests are complementary to the structural ones.**

***Keywords*-Model-Based Testing; Abstraction; Test Purpose;**

I. Introduction

Model-Based Testing (MBT) is a test generation approach that automatically computes, from a behavioral model of a system and selection criteria, a set of tests to be executed on an implementation of that system (the implementation under test, IUT). There are model based test generation suites available as commercial products, such as LTG [1] and TD[1], Conformiq[2], SpecExplorer [2], or as research prototypes, such as TGV [3] and STG [4], AGATHA [5], etc.

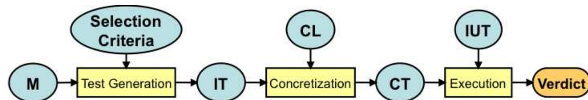

Fig. 1. Model-Based Testing Approach

A "standard" MBT approach is illustrated by means of Fig. 1. A model M describes the behavior of the system by means of a set of parameterized actions. The test generation tool, depending on selection criteria, instantiates from M a set of tests IT (instantiated tests). The tests in IT are sequences of action calls where the input parameters and the corresponding outputs are instantiated. The tests of IT are as abstract as the model, so a concretization layer CL translates them into concrete tests (CT), that are executed on the IUT. The verdict of success or failure of a test is delivered by comparing the results returned by the IUT with the ones predicted by M.

[1]LTG and TD are from Smartesting: `http://www.smartesting.com`
[2]Conformiq is from Qtronic : http://www.conformiq.com

One advantage of MBT is to have with the behavioral model an oracle to predict the outputs of the IUT. Another one is the possibility to achieve different kinds of coverage of the model (such as state coverage, decision coverage, path coverage) by varying the selection criteria. But writing the behavioral model and the concretization layer is costly, and ideally a large variety of test cases, i.e. achieving a large spectrum of coverage, should be computed from them.

An MBT tool-supported method, the LTG technology [1], has been developed in our research team and is now commercialized by the Smartesting company. It ensures structural coverage of the model by means of static (i.e. structural) selection criteria. Additionally, we intend to generate test cases issued from dynamic selection criteria [6], [7]. A test engineer (the *tester*) specifies a set of executions that he finds interesting to test by means of a *test purpose* (TP). The TP is specified in a language [7] that allows the tester to describe which actions to call and which states to reach. Some of the action calls can be left unspecified, which provides flexibility to the tester for expressing his test intention, but requires later these calls to be made explicit. This is performed by a research amongst the possible instantiations allowed by the model of the unspecified action calls, and it can possibly blow up with huge size models.

We propose in this paper to define, based on information extracted from the TP, a set of predicates from which we compute an abstraction A (of small size) of a behavioral model M. We first compute its synchronized product SP with the TP, and then a set of traces that cover every transition of SP. Each trace is a symbolic test for which we search an instantiation on M. This results in a new (w.r.t. the structural ones) set of instantiated tests IT, that can be executed on the IUT through the same concretization layer.

Our contributions are to define the set of abstraction predicates based on information of the TP. This makes the resulting abstraction related to what the tester intends to observe. Also, M provides an oracle for our new tests, and it is reused as well as the concretization layer to execute the tests on the IUT. We have an experimental validation of the method.

We detail our process for generating tests from abstractions and test purposes in Sec. II. Section III presents the "Qui-Donc" example that illustrates our approach. We define labelled transition systems (LTSs, for short) in Sec. IV and give the syntax of our behavioral models, whose semantics are LTSs. The abstractions and the test purposes are also defined as LTSs in Sec. V, where their synchronized product is defined.

The definition from the TP of a set of abstraction predicates is detailed in Sec. VI. This set is the basis for the computation of the abstraction, as explained in Sec. VII, where we also explain how to generate symbolic tests out of it. Experimental results are presented and discussed in Sec. VIII. Section IX concludes the paper, compares our approach to related works and gives some future research directions.

II. Test Generation Process from Dynamic Selection Criteria and Abstractions

Let us present our motivations, and our process for generating tests from dynamic selection criteria and abstractions.

A. *Problem and Motivations*

In our approach, the validation engineer describes by means of an hand-written test purpose TP how he intends to test the system, according to his know-how. We have proposed in [7] a language based on regular expressions, to describe a TP as a sequence of actions to fire and states to reach (targeted by these actions). The states are described as state predicates. The actions can be explicitly called by their name, or left unspecified by means of a generic name. This genericity allows the tester to describe a situation to reach without specifying *how* it should be reached. A regular expression repetition operator can be applied to the unspecified action call. This means in the tester's mind: leaving all possibilities to the system to reach a target state. The drawback is a combinatorial explosion of the number of executions of M in which to look for an instantiation of the action calls that reach the target state. The search should be guided to avoid that problem.

B. *Test Generation Process by Abstraction*

A small size state-transition graph, that symbolically models the executions of the system, can be obtained by computing an abstraction A of M. We propose to perform a synchronized product between A and the automaton that is the semantics of the TP, and to use paths of the synchronized product as guides for the instantiation of the TP on M.

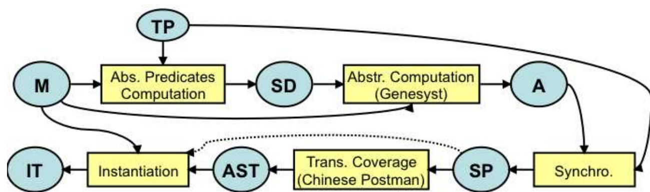

Fig. 2. Generating Tests from Test Purpose by Abstraction

Our approach is depicted in Fig. 2. From a behavioral model M and the state variables of a TP, we define a set SD of abstraction predicates. SD stands for sub-domains (of the state variables). The details of this definition are given in Sec. VI. The *GeneSyst* tool [8] produces an abstraction A of M based on the predicates of SD. The synchronized product, defined in Sec. V-B, of A with the TP results in a model SP. The executions of SP are the executions of A that match the TP. An implementation [9] of the Chinese Postman algorithm is applied to SP to cover its transitions. The result is a set of abstract symbolic tests AST. These tests are instantiated from M, which allows for re-using the same test execution environment (i.e. the concretization layer CL and the test execution tools) as of Fig. 1: the new instantiated tests IT (of Fig. 2) complement the ones of Fig. 1. Notice the dashed arrow in Fig. 2 from SP to the "instantiation" box. This is because, to instantiate the abstract symbolic tests, we use the reflexive transitions of SP that were ignored to generate AST. This point is explained in Sec. VII-B.

$$
\begin{array}{lcl}
X & \widehat{=} & \{\mathsf{HandSet}, \mathsf{TryCounter}, \mathsf{State}\} \\
I & \widehat{=} & \mathsf{HandSet} \in \{\mathsf{hang}, \mathsf{unhook}\} \wedge \\
 & & \mathsf{TryCounter} \in 0..2 \wedge \\
 & & \mathsf{State} \in \{\mathsf{welcome}, \mathsf{enter_num}, \mathsf{find_urgency}, \\
 & & \mathsf{find_num}, \mathsf{put_down}, \mathsf{busy}\} \wedge \\
 & & (\mathsf{State} = \mathsf{put_down} \Leftrightarrow \mathsf{HandSet} = \mathsf{hang}) \wedge \\
 & & (\mathsf{State} = \mathsf{put_down} \Rightarrow \mathsf{TryCounter} = 0) \\
\mathit{init} & \widehat{=} & \mathsf{HandSet} = \mathsf{hang} \wedge \mathsf{TryCounter} = 0 \wedge \\
 & & \mathsf{State} = \mathsf{put_down} \\
\mathsf{Call} & \widehat{=} & \mathsf{HandSet} = \mathsf{hang} \wedge \\
 & & \quad \mathsf{HandSet}' = \mathsf{unhook} \wedge \\
 & & \quad \mathsf{TryCounter}' = \mathsf{TryCounter} \wedge \\
 & & \quad \mathsf{State}' = \mathsf{welcome} \\
\mathsf{HangUp} & \widehat{=} & (\mathsf{HandSet} = \mathsf{unhook} \vee \mathsf{State} = \mathsf{busy}) \wedge \\
 & & \quad (\mathsf{HandSet}' = \mathsf{hang} \wedge \mathsf{TryCounter}' = 0 \\
 & & \quad \wedge\ \mathsf{State}' = \mathsf{put_down})
\end{array}
$$

Fig. 3. A fragment of the Behavioral Model of the Qui-Donc System

III. The Qui-Donc Example

Our approach is illustrated by means of the Qui-Donc example [10]. It is a reverse phone book service. When the user contacts the service, he gets a welcome message followed by an invitation to enter the number he searches for. When he has done so, the service possibly answers that the number is an emergency one, or is invalid, or is unknown, otherwise the answer is the name and address of the owner of the number. In case the user forgets to provide an input, or provides an unexpected one (such as an unauthorized key), the service invites up to twice the user to provide its input again and finally closes the communication if needed.

We have designed a behavioral model of the Qui-Donc. Let us introduce its variables and some actions, which appear in a forthcoming example of test purpose. The Call and HangUp actions allow respectively to call the service and to close the communication. The Delay action simulates that the user remains inactive during a fixed delay. The $\mathsf{HandSet} \in \{\mathsf{hang}, \mathsf{unhook}\}$ variable stands for the state of the handset. The $\mathsf{TryCounter} \in \{0, 1, 2\}$ variable counts how many times the user has provided an unexpected (or void) input. The State variable indicates in which state the service is. Amongst the six possible State values are welcome (when the service is ready to listen to a new input from the user) and put_down (when the communication has been closed). A fragment of its model is expressed w.r.t. the syntax of Def. 4 in Fig. 3.

IV. Behavioral Models and their Semantics

We define Labelled Transition Systems (LTSs) and present a syntax, inspired from the guarded commands [11], [12], to

describe the behavioral models.

A. Labelled Transition Systems

The state transition graph of the behavioral models, their abstractions as well as the test purposes are formalized as LTSs (see Def. 1). Notice that we only consider LTSs with finite state spaces in this paper, which is usual with models for the test.

Definition 1 (Labelled Transition System): An LTS is defined by a tuple $\langle O, Q, Q_0, \Delta, AP, L, Q_f \rangle$, where

- O is a finite set of action names,
- Q is a finite set of states,
- $Q_0 (\subseteq Q)$ is a set of initial states,
- $\Delta (\in Q \times O \times Q)$ is a labelled transition relation,
- AP is the set of atomic propositions,
- $L (\in Q \rightarrow 2^{AP})$ is a state labelling function which maps each state to the set of atomic propositions that hold in this state,
- $Q_f (\subseteq Q)$ is a set of final states.

We write $q \xrightarrow{o} q'$ instead of $(q, o, q') \in \Delta$ for the sake of simplicity. An execution σ of an LTS is a finite sequence of transitions represented by a sequence of pairs $\sigma = (\perp, q_0), (o_1, q_1), \ldots (o_n, q_n)$ where o_i is an action name and q_i a state. σ is such that q_0 is an initial state of Q_0 and for every i in [0, n-1], $q_i \xrightarrow{o_{i+1}} q_{i+1}$ and $q_n \in Q_f$.

The set of atomic propositions AP is defined over a set of state variables X and their domains by relational operators. Any $x \in X$ has a domain denoted by $D(x)$. Let v be a value of $D(x)$ and V be a sub-domain of $D(x)$ ($V \subseteq D(x)$). An atomic proposition is either in the shape of $x = v$ in a concrete LTS where the states are valuated, or in the shape of $x \in V$ in an abstract LTS where the states are symbolic. Notice that we unify the notations and denote $x = v$ by $x \in \{v\}$. When necessary, we use an upper-script notation to indicate the name of the LTS to which we refer.

An LTS A is compatible with an LTS M (see Def. 2) if its action names are in M, its set of state variables is included into that of M, and the atomic propositions of AP^A partition the domains of the variables of the subset of state variables of M that also are state variables of A into several sub-domains.

Definition 2 (Compatible LTSs): Let M and A be two LTSs. A is compatible with M if:

- $O^A \subseteq O^M$ (A uses only actions of M),
- the set of variables X^A on which is defined AP^A is a subset of X^M on which is defined AP^M,
- for any atomic proposition $x \in V^M$ in AP^M such that x is in X^A, there exists a proposition $x \in V^A$ in AP^A such that $V^M \subseteq V^A$.

Two states q^A and q^M are compatible (see Def. 3) if q^A is an abstract state that includes the more concrete state q^M.

Definition 3 (Compatible States): Let A be an LTS compatible with an LTS M. A state q^A is compatible with a state q^M if for any atomic proposition $x \in V^A$ in $L^A(q^A)$ there exists a proposition $x \in V^M$ in $L^M(q^M)$ with $V^M \subseteq V^A$.

We say that $(q^A,\ q^M)$ is a *compatible pair* when q^A is compatible with q^M.

B. Model syntax and semantics

Our work does not necessitate that we specify a particular modelling syntax. We only consider a model as being defined on a set of variables, and as being specified by means of an initial condition and a transition relation. They are specified syntactically as first-order logic predicates. The relation symbols form the atomic predicates. The expressions are defined in the data set theory, as it is for example the case in B [13]. The main data structures are sets, functions and relations. Some relation and function symbols may have fixed interpretations, such as $=, \neq, \leq, \in, +$, etc.

Definition 4 (Behavioral Model): A behavioral model is defined by a tuple $\langle X, I, \mathit{init}, O, OP \rangle$where:

- X is a finite non-empty set of state variables. Each variable $x \in X$ has a finite or infinite domain of values, denoted by $D(x)$,
- I is an invariant specified as a predicate on X,
- *init* is an initial condition specified as a predicate on X such that $\mathit{init} \Rightarrow I$,
- O is a set of guarded action labels,
- OP is the transition relation, specified by the definition of every guarded action (labelled by o) by an equation $o \mathrel{\widehat{=}} T_o(X, X')$, where $T_o(X, X')$ is a before-after predicate on $X \cup X'$. X' is a set of "next-state" variables that is in 1-1 correspondence with X. It is such that $I \wedge T_o(X, X') \Rightarrow I'$, where I' denotes the invariant in which the variables of X' replace the ones of X.

The semantics of a behavioral model is an LTS. Q is the subset of the cartesian product of the domains of the variables X that satisfy the invariant condition I. The value of a predicate e in a state q is denoted by $e(q)$. It can be defined by induction on the syntax of the predicates. A state q is an initial state in Q_0 iff $init(q) = true$. The transition relation is defined as follows: there is a transition $q \xrightarrow{o} q'$ iff $T_o(q, q')$ is true. The state labelling function L is defined as follows: for each atomic predicate $x = v$, we have that $x \in \{v\}$ is in $L(q)$ iff v is the value of x in q. Any state is a final state: $Q_f = Q$.

Figure 3 models a fragment of the Qui-Donc example[3]. Each action in a behavioral model is made of one or many elementary guarded actions (EGA) in the shape of $G \wedge X' = f(X)$, that assign all the state variables of X when a guard G is true. Each of the two actions Call and HangUp in Fig. 3 is made of only one EGA, but there can be several of them in an operation, as it is the case for the Delay operation. We denote by $T_{o_i}(X, X')$ the ith EGA of an action o.

V. Abstraction, Test Purpose, Synchronization

In this section, we define the abstractions, the test purposes and their synchronization as LTSs.

A. Abstraction, Test Purpose

An abstraction A of a behavioral model M is an LTS defined on a subset of the variables of M, $X^A \subseteq X^M$. It is compatible with the semantics of M. It uses the same set of action names

[3]Its LTS can be seen on: http://lifc.univ-fcomte.fr/∼testAndAbs/index.html

as M. Each variable of X^A has an abstracted domain of values which is a partition of its domain in M. The state space Q^A is the subset of the cartesian product of the set of the abstracted domains of the variables X^A that satisfy the invariant condition I^M. Any state is a final state. AP^A is the set of atomic predicates defined on X^A and their abstracted domains. An atomic proposition on A is denoted by $x \in V^A$ where V^A is one of the elements of $D^A(x)$ which is a sub-domain of $D^M(x)$. Figure 5 shows an example of abstraction of the Qui-Donc model.

A test purpose TP defined w.r.t. a behavioral model M (whose abstraction is A) is an LTS, where there is only one initial state in Q_0^P. This LTS is defined as follows. The labels of the transitions are actions names of M. With L^P, any state of the set Q^P is labelled by a set of atomic predicates defined w.r.t. a subset of the variables of X^M. The set of final states Q_f^P is a subset of Q^P which is defined by the tester. We assume that the tester defines test purposes that are compatible (see Def. 2) with the LTS semantics of M.

In our context, the sets of atomic predicates of a TP compatible with the semantics of M are subsets of atomic predicates of the abstraction A of M. By construction in our method, the set of symbolic states of A is a partition of the set of symbolic states of TP (see definition of SD in Sec. VI).

Figure 4 shows the automaton representation of a TP for the Qui-Donc. Its aim is to test that the try counter, once incremented, gets back to 0 in case of a correct entry from the user. For readability purposes, in the graphical representations of the LTSs of Fig. 4 and 5, some transitions are labelled with a set of labels: this means that there are as many transitions in the LTS as there are labels in the set. Notice that the tester does not have to draw the automaton to express a TP: he would rather use the language of [7]. The automaton would be its semantics. In our example, TP is compatible both with the semantics of M and with A. By construction, A is also compatible with the semantics of M. The sets of action names satisfy the conditions $O^P \subseteq O^A$ and $O^A = O^M$. The set of state variables $X^M \mathrel{\widehat{=}} \{\mathsf{HandSet}, \mathsf{TryCounter}, \mathsf{State}\}$, $X^A \mathrel{\widehat{=}} \{\mathsf{TryCounter}, \mathsf{State}\}$ and $X^P \mathrel{\widehat{=}} \{\mathsf{TryCounter}\}$ are such that $X^P \subseteq X^A \subseteq X^M$. The set of atomic propositions $AP^P \mathrel{\widehat{=}} \{\mathsf{TryCounter} \in \{0\}, \mathsf{TryCounter} \in 1..2\}$ is included in $AP^A \mathrel{\widehat{=}} \{\mathsf{TryCounter} \in \{0\}, \mathsf{TryCounter} \in 1..2, \mathsf{State} \in \{\mathsf{welcome}\}, \mathsf{State} \in \{\mathsf{put_down}\}, \mathsf{State} \notin \{\mathsf{welcome}, \mathsf{put_down}\}\}$. Thus TP is compatible with A and A is compatible with M.

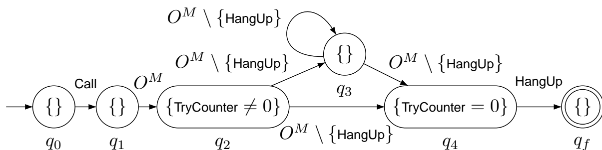

Fig. 4. A Test Purpose for Qui-Donc

B. Synchronization of an Abstraction and a Test Purpose

We synchronize two compatible LTSs as in [14]: an abstraction A and a test purpose TP. This synchronization is a particular synchronized product of LTSs (see Def. 5), in which

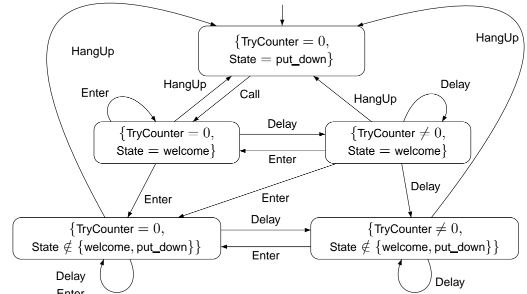

Fig. 5. An Abstraction of the Qui-Donc System

the transitions that have the same label are synchronized when their source and target states are compatible and when they are not reflexive in the abstraction. The reflexive transitions of A are not considered for the synchronized product because they do not help in progressing towards a target state of the TP. But they are essential for the instantiation of the abstract tests. Consequently, the input SP of the instantiation function of Fig. 2 is defined as in Def. 5, except that this time the reflexive transitions of A are taken into account. It is defined from Def. 5 by suppressing the condition $q^A \neq q'^A$ of the fourth item.

Definition 5 (Synchronized Product of two compatible LTSs): The synchronized product between a test purpose $\langle O^M, Q^P, Q_0^P, \Delta^P, AP^P, L^P, Q_f^P \rangle$ with a compatible LTS $\langle O^M, Q^A, Q_0^A, \Delta^A, AP^A, L^A, Q_f^A \rangle$ of an abstraction is an LTS $\langle O^M, Q, Q_0, \Delta, AP^A, L, Q_f \rangle$, where:

- $Q (\subseteq Q^P \times Q^A)$ is the subset of compatible pairs (q^P, q^A) of the cartesian product $Q^P \times Q^A$,
- $Q_0 (\subseteq Q_0^P \times Q_0^A)$ is the subset of compatible pairs of the cartesian product $Q_0^P \times Q_0^A$,
- for any state $(q^P, q^A) \in Q$, $L((q^P, q^A)) = L^A(q^A)$,
- for any pair of pairs of compatible states $((q^P, q^A), (q'^P, q'^A))$, $(q^P, q^A) \xrightarrow{o} (q'^P, q'^A)$ if $q^P \xrightarrow{o} q'^P$, $q^A \xrightarrow{o} q'^A$ and $q^A \neq q'^A$,
- $Q_f = \{(q^P, q^A) \mid (q^P, q^A) \in Q \wedge q^P \in Q_f^P \wedge q^A \in Q_f^A\}$.

Notice that $Q_0^P = \{q_0^P\}$. The state q_0^P is always compatible with any state of Q_0^A because $L^P(q_0^P) = \{\}$.

VI. Set of Abstraction Predicates Definition

In this section, we present how we define a set of abstraction predicates from a test purpose TP and a behavioral model M. To compute the abstraction, we use *GeneSyst* [8]. It requires that a set of symbolic states is defined by a set of predicates. This is a kind of predicate abstraction [15], [16], [12] that uses a first-order theorem prover on the set theory.

We consider in this section a test purpose TP defined as an LTS and a behavioral model M defined as a guarded action system according to Def. 4. We call state predicate of TP a set of atomic propositions that labels a state of TP.

The set of abstraction predicates is defined w.r.t. the state predicates and the actions appearing in TP. We propose to define this set of in two steps:

- extraction of the subset X^A of the variables of X^M that are used in the state predicates of TP or modified by the actions explicitly fired in TP,
- partition of the domains of these variables according to their use in the state predicates and in the actions of TP.

The set of abstraction predicates is defined from a TP that is an LTS compatible with the semantics of M. In our example, the action names in the TP of Fig. 4 are Call and HangUp. The state predicates are about the state variable TryCounter.

A. Extraction of the Variable Names

We define $O^P (\subseteq O^M)$ as the set of the actions explicitly fired in TP, i.e. the ones that appear at least once in the labels of the graphical representation of TP without being subtracted from O^M. We have $O^P \mathrel{\hat=} \{\mathsf{HangUp}, \mathsf{Call}\}$ with Fig. 4.

The set of variables $X^A (\mathrel{\hat=} X^A_{sp} \cup X^A_{op})$ of the abstraction A is the union of the set of variables of M that are used in the state predicates of TP, denoted by X^A_{sp}, and the set of variables modified by all the actions of TP, denoted by X^A_{op}:

- $X^A_{sp} \mathrel{\hat=} \bigcup_{q \in Q^{\mathsf{P}}} \{x \mid x \in V \text{ is in } L^{\mathsf{P}}(q)\}$,
- $X^A_{op} \mathrel{\hat=} \{x|$ for any o in O^{P} the definition $T^M_o(X^M, X'^M)$ contains a predicate $x' = e$ where $e \neq x\}$.

For the example of Fig. 4, the set of variables abstracting the Qui-Donc is $X^A \mathrel{\hat=} \{\mathsf{TryCounter}, \mathsf{State}\}$. TryCounter $(\in X^A_{sp})$ is used in the state predicates and State $(\in X^A_{op})$ is modified by both Call and HangUp (see Fig. 3) in the TP of Fig. 4.

B. Partition of the Variable Domains

To define the set of sub-domains of the variables of X^A, we define (i) the set of sub-domains issued from the state predicates of TP and (ii) the sub-domains of the symbolic states targeted by the actions of TP. Then we split these sub-domains into parts so as to realize the smallest partition of domains of each variable of X^A.

The set SD^x_{sp} for item (i) is the union of the atomic predicates, on the variable x, that label the states of TP:

$$SD^x_{sp} \mathrel{\hat=} \bigcup_{q \in Q^P} \{x \in V \mid x \in V \text{ is in } L^P(q)\}.$$

We obtain for example the two following sub-domains for the variable TryCounter of the TP in Fig. 4[4]:

$SD^{\mathsf{TryCounter}}_{sp} \mathrel{\hat=} \{\mathsf{TryCounter} = 0, \mathsf{TryCounter} \neq 0\}$.

Let X be a set of variables. To distinguish between a variable x and the others in X, we define $Z = X \setminus \{x\}$.

The set of sub-domains issued from the actions (item ii) for a variable x is the set of strongest postconditions of every EGA a of every action used in TP ($\in O^P$) from the precondition $p \mathrel{\hat=} x \in D^M(x)$. It is denoted by $sp(p, a)$ with $sp(x \in D^M(x), a) \mathrel{\hat=} x' \in V$ such that

$$x' \in V \Leftrightarrow \exists x \cdot \exists Z \cdot \exists Z' \cdot \\ (\bigwedge_{z \in Z} (z \in D^M(z) \wedge z' \in D^M(z)) \wedge a \wedge x \in D^M(x)).$$

[4]Notice that we have simplified the writing of the predicates that should have been denoted by TryCounter $\in \{0\}$ and TryCounter $\in 1..2$.

Let n_o be the number of EGA of the action o. The set of sub-domains SD^x_{op} is defined as:

$$SD^x_{op} \mathrel{\hat=} \bigcup_{o \in O^P, i \in 1..n_o} \{x \in V \mid x' \in V = \\ sp(x \in D^M(x), T^M_{o_i}(X^M, X'^M))\}.$$

For example, the strongest postcondition for the precondition State $\in D^M(\mathsf{State})$ of the only EGA of Call is the predicate $\mathsf{State}' \in \{\mathsf{welcome}\}$.

So we have $SD^x_{sp} \cup SD^x_{op} = \{x \in V_1, x \in V_2, \ldots, x \in V_n\}$. When the sub-domains V_i intersect, we partition them. If the set of sub-domains do not overlap the whole domain of x, we add as the last sub-domain the complement of the union of the sub-domains. So the set of sub-domains SD^x of any variable x of X^A is the smallest partition of $D^M(x)$ w.r.t. the set $SD^x_{sp} \cup SD^x_{op}$ whose cardinal is n. Let $\mathbb{F}_1(1..n)$ be the set of non empty finite parts of the set $1..n$ and $\complement_J$ be the complementary set of J in $1..n$. The partition is defined as:

$$SD^x \mathrel{\hat=} \bigcup_{J \in \mathbb{F}_1(1..n)} \{x \in (\bigcap_{j \in J} V_j \setminus \bigcup_{i \in \complement_J} V_i)\} \cup \\ \{x \in (D^M(x) \setminus \bigcup_{j \in 1..n} V_j)\}.$$

Finally, the set of sub-domains is defined as:

$$SD \mathrel{\hat=} \bigcup_{x \in X^A} SD^x$$

For example, the State variable has three sub-domains: State $\in \{\mathsf{welcome}\}$, State $\in \{\mathsf{put_down}\}$, State $\in \{\mathsf{enter_num}, \mathsf{find_urgency}, \mathsf{find_num}, \mathsf{busy}\}$ (shortly State $\notin \{\mathsf{welcome}, \mathsf{put_down}\}$). They are obtained as follows. The variable State is not used in the state predicates ($SD^{\mathsf{State}}_{sp} = \{\}$). The variable State is modified by every action appearing in TP (Call, HangUp). State $\in \{\mathsf{welcome}\}$ is the strongest postcondition of the action Call and State $\in \{\mathsf{put_down}\}$ is the strongest postcondition of the action HangUp. State $\notin \{\mathsf{welcome}, \mathsf{put_down}\}$ is the complement of the domain of the variable State.

We can obtain the predicates that define the sub-domains by constraint solving. This requires all the domains of the variables to be defined as finite sets, which is usual in a model for the test.

The abstraction predicates define a set of symbolic states as the cartesian product of the sets of the abstract domains of any variable of X^A. We obtain for our example the set of abstraction predicates shown in the following table. This defines six symbolic states. Only five of them are reachable and appear in the abstraction computed by *GeneSyst* and shown in Fig. 5.

Variable	*Sub-domains*
TryCounter	TryCounter $= 0$, TryCounter $\neq 0$
State	State = welcome, State = put_down, State $\notin \{\mathsf{welcome}, \mathsf{put_down}\}$

In Proposition 1, we prove that with the assumption that the tester designs a TP compatible with M, this definition of the set

of abstraction predicates SD makes that the TP is compatible with an abstraction A computed from them.

Proposition 1: Consider a TP compatible with M and an abstraction A computed from the set of abstraction predicates SD. Then the TP is compatible with A.

Proof: By assumption, the test purpose TP is compatible with M. Hence the three following conditions hold:

- $O^{\mathsf{P}} \subseteq O^M$,
- $X^{\mathsf{P}} \subseteq X^M$,
- for any x in X^{P} and for any atomic proposition $x \in \{v\}$ in AP^M, there exists an atomic proposition $x \in V$ in AP^{P} such that $v \in V$.

By definition of X^A in this section, $X^{\mathsf{P}} \subseteq X^A$ and $X^A \subseteq X^M$. By definition of SD in this section, it is a partition of the domains of the variables of X^A and $SD = AP^A$. Moreover $O^A = O^M$. Let SD_{sp} be the union of SD^x_{sp} for any x in X^{P}. By definition of SD_{sp}, $AP^{\mathsf{P}} = SD_{sp}$. Then, by definition of SD, the predicates of SD_{sp} are redefined in SD from some predicate of SD_{op} in such a way that the predicates of SD define a domain partition of any variable of X^{P}. Hence for any predicate $x \in V^A$ in AP^A that concerns a variable x of X^{P}, there exists a predicate $x \in V^{\mathsf{P}}$ in AP^{P} such that $V^A \subseteq V^{\mathsf{P}}$. Therefore TP is compatible with A. ∎

VII. Abstraction and Test Generation

A. Generation of the Abstraction

We use *GeneSyst* to generate an abstraction from a behavioral model M and a set of symbolic states. This abstraction is an LTS that is an over-approximation of M: it simulates all the executions of M, but adds new ones. *GeneSyst* tries to prove automatically the feasibility or not of transitions between the symbolic states. It proceeds by weakest precondition computations and satisfiability evaluations over first order logical formulas [8]. *GeneSyst* takes B specifications [13] as input. The weakest precondition of a statement S that leads to the abstract state a' is defined by the B substitution calculus. It is denoted by $[S]a'$. A transition from an abstract state a to a' is feasible if $a \wedge [S]a'$ is satisfiable. If $a \Rightarrow \neg[S]a'$ is valid then the transition is not feasible. On the contrary, when the proof of $\exists X \cdot (a \wedge [S]a')$ succeeds, the transition $a \rightarrow a'$ is added to the LTS. It is also added when this proof is inconclusive, although it is possibly infeasible. This makes the abstraction more over-approximated.

Thus, some of the symbolic tests that we generate from the abstraction may not be possible to instantiate as executions of the behavioral model. This would result in a bad coverage of the abstraction by the instantiated tests. It is possible to use an interactive prover to try to get rid of the proof failures. We have chosen another alternative: using constraint solving techniques makes it possible to automatically check the feasibility of the unproved transitions when the state space is finite. The applicability of this technique depends on the size of the domains, as it proceeds by partial consistency checking and domain enumeration. We will assess the practical impact of the instantiation problem on some examples in Sec. VIII.

B. Generation and Instantiation of the Symbolic Abstract Tests

We compute the symbolic abstract tests as selected executions of the abstraction, by running an implementation [9] of the chinese postman algorithm on the synchronized product SP of the abstraction with the TP (see Fig. 2). This provides a set of paths such that every transition of SP is covered at least once. Every path is a symbolic abstract test that terminates in a final state of SP. It is a sequence of non parameterized action calls. We still have to instantiate the tests, i.e. to find parameter values that make these sequencings of actions possible according to the behavioral model M. We proceed by a symbolic animation of the tests on M. It is possible that a sequence can not be instantiated as it is: an action might not be enabled on a given instance of a symbolic state. Thus we will use a version of SP augmented with its reflexive transitions to complete the instantiation. Indeed, these transitions may lead to another instance of the same symbolic state, from which the action could be enabled. As a result, we insert bounded sub-sequences of (reflexive) action calls into the original sequence. We have implemented this instantiation procedure. Although naive and incomplete (invoking reflexive transitions is not always sufficient, sometimes cycles are necessary), our algorithm gave satisfactory instantiation results on our case studies, as shown in Sec. VIII.

VIII. Experimental Results

We have applied our method to three various cases of reactive systems: a reverse phone book service (Qui-Donc [10]), an automatic conveying system (Robot [17]) and an electronic purse (DeMoney [18]). Tests have been generated for each of them from two TPs, on a 2.8GHz Pentium with 1GB of RAM.

The first two parts of Table I show the size of the behavioral models and the TPs. The symbol "♯" means *number of*, "Act." stands for *Actions*, and "Trans." stands for *Transitions*. For example, the TP of Fig. 4 (in bold font in Table I) is made of 6 states, 8 transitions, 2 explicit action calls and two non empty sets of atomic predicates. The behavioral model of the Qui-Donc is made of 4 actions, with a total of 20 EGAs. It is 122 lines long and it defines 13 states from three variables whose average domain size is 3.66.

A. Abstraction Generation and Synchronous Product

The last two parts of Table I are about the abstraction generation and the synchronous product (SP) computing. *GeneSyst* failed to prove from 66% up to 90% of the transitions. We have checked, with success on our examples, the satisfiability of the unproved POs by means of a constraint solver [19] (see the *Filter.* columns in Table I). The *filtered transitions* are the ones really unsatisfiable. There are only few of them (from 3% up to 18%), with the exception of DeMoney. Filtering is almost immediate for Qui-Donc and the Robot, modelled with variables enumerated on small domains. But it takes one hour for DeMoney, whose model has some numerical variables.

Case studies	Behavioral Model						Test purpose				Abstraction generation							SP		
	♯Act.	♯EGA	♯Enum. Var.	♯Num. Var.	♯Lines	♯States	♯States	♯Trans.	♯Act.	♯State Pred.	♯States	♯Trans.	♯PO	Computing time	♯Unproved Trans.	♯Filtered Trans.	Filtering time	♯States	♯Trans.	♯Filtered Trans.
Qui-Donc	**4**	**20**	**3 (3.66)**	**0**	**122**	**13**	3	3	2	0	5	24	180	2 min.	16 (66.7 %)	2 (12.5 %)	$\leq$ 1 sec.	9	36	3
							6	**8**	**2**	**2**	5	18	147	1.5 min.	12 (66.7 %)	2 (17.7 %)		12	54	8
Robot	9	10	6 (3)	0	107	372	6	7	3	2	6	37	452	5 min.	32 (86.5 %)	1 (3.1 %)		17	106	3
							7	14	0	4	8	50	557	8 min.	37 (74 %)	0 (0 %)		23	214	0
DeMoney	11	42	3 (9)	6	530	10^{30}	3	4	2	0	3	148	1138	35 min.	133 (89.8 %)	49 (36.8 %)	1 h.	7	332	111
							4	5	2	2	7	228	9690	1 h. 30 min.	188 (82.4 %)	56 (29.8 %)		18	422	90

TABLE I

METRICS ON BEHAVIORAL MODEL, ABSTRACTION GENERATION AND SIZE OF SYNCHRONOUS PRODUCT

Model	LTG Test Generation			Test generation from M		Instantiation of the SP tests without Filtering						Instantiation of the SP tests with Filtering						
	♯Tests	Average length	Standard deviation	♯Tests	Average length	♯Symbolic tests	Average length	♯Instant. tests	Average length	♯Instant. tests / ♯Symbolic tests	Time	♯Symbolic tests	Average length	♯Filtered Trans.	♯Instant. tests	Average length	♯Instant. tests / ♯Symbolic tests	Time
Qui-Donc	20	3.65	1.02	23	9.3	10	5.3	4	3	4/10 (40%)	$\leq$ 1 sec.	10	4.9	2	6	3.8	6/10 (60%)	$\leq$ 1 sec.
	9	2.66	0.81	26	8.3	15	8	5	5.6	5/15 (33%)	$\leq$ 1 sec.	11	8.2	2	6	7.8	6/11 (55%)	$\leq$ 1 sec.
Robot	19	4.10	1.36	71	149.3	12	11.1	7	21	7/12 (58%)	3 min.	11	11.18	1	7	23.7	7/11 (64%)	2 min.
	24	4.25	1.21	808	32.2	23	9.1	8	23.1	8/23 (35%)	5 min.	23	9.1	0	8	23.1	8/23 (35%)	5 min.
DeMoney	26	1.57	0.53	-		32	10.88	0	-	0/32 (0%)	2 h.	19	10.78	49	0	-	0/19 (0%)	2 h.
	13	1.07	0.14			42	10.2	0	-	0/32 (0%)	2 h.	18	10.1	56	17	12.48	17/18 (94%)	1 h.

TABLE II

TEST GENERATION FROM STATIC AND DYNAMIC CRITERIA, TESTS INSTANTIATION STATISTICS

B. Symbolic Test Generation and Complementarity

We have presented in [6] a technique to build a new behavioral model synchronized with a TP, from which LTG generates tests. The results in the first part of Table II have been obtained with this technique. We also have tried (see the second part of Table II) to first synchronize directly the LTS semantics of the behavioral models with the TPs, without using abstractions, and then to generate the tests with a chinese postman walk. Although it has been possible to do so with Qui-Donc and the Robot, it failed with DeMoney due to the size of the resulting state space. This shows the utility of using abstractions to apply this test generation method. The complementarity with the LTG tests has been studied in [6]. It also appears in our results. LTG covers every EGA and the chinese postman walk covers every transition. For example with the first TP of the Robot, 80 test steps have been generated by LTG to cover every EGA, and more than 10500 tests have been generated to cover every transition.

The tests in the last two parts of Table II have been obtained (in less than one second) by a chinese postman walk on the synchronized product SP between an abstraction and a TP. The tests of the third part have been obtained without filtering the unfeasible transitions by constraint solving, whereas in the fourth part they have been filtered. Although the transitions of the without filtering version of the abstraction are a superset of the ones of the with filtering version, the “without filtering” tests are not a superset of the “with filtering” ones. They cannot be compared as the chinese postman algorithm does not necessarily covers the same traces of the two abstractions.

C. Test Instantiation

Using abstractions require the generated tests to be instantiated from the concrete model. But our abstractions are over-approximations, and they contain unproved transitions potentially infeasible. The results of the “without Filtering” part in Table II are for abstractions where every unproved transition has been kept. From 35% up to 58% of the tests have been instantiated, except with the two TPs of DeMoney for which no test have been instantiated. In contrast, for the results of the “with filtering” part of Table II, we have removed the filtered transitions from the abstraction. From 35% up to 94% of tests have been instantiated, except with one TP of DeMoney. There are two reasons why with filtering, the ratio of instantiated tests grows. There are less transitions, thus less symbolic tests. Moreover, the former symbolic tests going through infeasible transitions are replaced by other tests, possibly instantiable. For example, with the second TP of DeMoney, the first transition of every symbolic test generated without filtering is infeasible, so that no test is instantiable. In contrast, with filtering, this transition is replaced by a feasible one and 94% of the tests become instantiable. Notice that even without unproved POs, the abstraction computed by *GeneSyst* is still an over-approximation and may produce uninstantiable tests. Our study shows that except in one case, this has a limited practical impact on our examples.

IX. CONCLUSION, RELATED AND FURTHER WORKS

We have presented in this paper an MBT process based on the use of an abstraction A of a model M. A is generated from a set of abstraction predicates that are defined based on the state variables involved and modified by a test purpose TP. A and TP are synchronized into a model SP made of all the sequences of A that match TP. By covering all the states and transitions of SP we get a set of symbolic abstract tests. Since A is an abstraction that models more executions than M, whose states and transitions are over-approximated, it is possible that at the end, the instantiated test suites do not cover all the states and all the transitions of SP. But Sec. VIII shows that the ratio of instantiated tests is satisfactory in practice, although we use a naive and incomplete algorithm.

Our method helps in finding executions of M that match TP. SP brings the following information to help the search for TP executions that exist in M: the predicates that define the symbolic states and the instantiation of the generic action calls. Our first experimental results indicate that the method should provide a valuable help in practice. We have shown in [6] the complementarity of our tests with the LTG ones.

Other works are about the generation of tests from abstraction. We make a predicate abstraction as in [16]. T. Ball uses it to generate structural tests of a program. He computes

a boolean abstraction from a set of predicates which is the set of all the control-flow conditions of the program. We use an abstraction to generate functional tests in an MBT approach. We compute abstractions from a set of predicates defined from a test purpose as presented in Sec. VI. These two abstraction methods are based on weakest precondition and strongest postcondition computations, but by considering a model instead of a program, our method benefits from invariant properties to obtain a finer abstraction. Moreover, as we are in an MBT context, we are able to compute an oracle. Both methods aim at maximizing the ratio of the number of instantiated tests to the number of symbolic tests. We compute an over-approximation and we experimentally show that this ratio is between 35% and 94% on our case studies (except in a particular case that is explained in Sec. VIII-C) using an incomplete algorithm. In [16], Ball presents a method to compute a lower bound of the set of reachable states. Then he only generates instantiable tests which cover this set. Our LTS should be extended in Tri-Modal Transition System [20] to make this lower bound computation possible.

The method implemented in the tool Agatha [5] also computes an abstraction from a model, but by applying a symbolic execution technique. This abstraction approach on data of models is very different from predicate abstractions. In [5], Agatha does not make use of test purposes.

The methods in [21] implemented in STG [4] use an abstraction defined by the user and modelled by an IOSTS (Input Output Symbolic Transition System). These approaches use test purposes synchronized with abstractions, both defined as IOSTS. Then the synchronized product allows for generating tests after an optimization step, which consists of a pruning of unreachable states by abstract interpretation. Our approach is very similar in that we also use test purposes and abstractions, as well as synchronization and constraint solving techniques to instantiate the symbolic tests. The two following points explain the differences. First, our abstractions are computed from a set of predicates that are defined from the test purposes, whereas the abstractions used by STG are given by the user. Second, the optimization is performed by the abstraction computation. It consists of using the invariant properties (that do not exist in an IOSTS) of the models in the weakest precondition computation for the minimization of the symbolic state space and of the feasible transitions.

The approach in our paper differs from the one of [22], where the test purposes are generic and aim at achieving a static coverage of the instructions and the conditions. Our method intends to become integrated into LTG, that already achieves such a static coverage, to complete it with tests issued from dynamic criteria. Another difference with [22] is that our approach relies on abstractions.

As in [23] and [24], it is also possible to describe test purposes by means of LTL properties, that specify particular state sequencings. Our language allows for specifying action calls in addition to states descriptions.

This work has to be pursued to better assess the contribution of the method, from a practical point of view. It seems necessary to reduce the time for generating the abstraction, for example by using syntactic abstraction as defined in [12]. Also, a finer combination of the proof and constraint solving techniques may significantly help to remove the transitions that really cannot be instantiated from SP. Our valuation algorithm also needs to be improved. This will provide a better generation of the instantiated tests.

REFERENCES

[1] E. Jaffuel and B. Legeard, "LEIRIOS Test Generator: Automated test generation from B models," in *B'2007, Tool Session*, ser. LNCS, vol. 4355. Springer, 2007, pp. 277–280.

[2] M. Barnett, K. Leino, and W. Schulte, "The spec# programming system: An overview," in *CASSIS'04*, ser. LNCS, vol. 3362. Springer, 2005, pp. 49–69.

[3] C. Jard and T. Jéron, "TGV: theory, principles and algorithms," *Software Tools for Technology Transfer*, vol. 7, no. 1, pp. 297–315, 2005.

[4] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva, "Symbolic test selection based on approximate analysis," in *TACAS'05*, ser. LNCS, vol. 3440, 2005.

[5] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois, "Behavioral unfolding of formal specifications based on communicating extended automata," in *ATVA'03, Automated Technology for Verification and Analysis*, 2003.

[6] J. Julliand, P.-A. Masson, R. Tissot, and P.-C. Bué, "Generating tests from B specifications and dynamic selection criteria," *FAC, Formal Aspects of Computing*, 2009, to appear (accepted manuscript). Revised and extended version of a paper from the ABZ'08 conference.

[7] J. Julliand, P.-A. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in *AST'08*. ACM Press, 2008, pp. 41–44.

[8] D. Bert, M.-L. Potet, and N. Stouls, "Genesyst: a tool to reason about behavioral aspects of B event specifications," in *ZB'05*, ser. LNCS, vol. 3455, 2005.

[9] H. Thimbleby, "The directed chinese postman problem," *Software: Practice and Experience*, vol. 33, no. 11, pp. 1081–1096, 2003.

[10] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann, 2006.

[11] E. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs," *C. ACM*, vol. 18, 1975.

[12] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *CAV'00*, ser. LNCS, vol. 1855, 2000, pp. 435–449.

[13] J.-R. Abrial, *The B Book*. Cambridge Univ. Press, 1996.

[14] T. Jéron and P. Morel, "Test generation derived from model-checking," in *CAV*, 1999, pp. 108–121.

[15] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *CAV'97*, ser. LNCS, vol. 1254, 1997, pp. 72–83.

[16] T. Ball, "A theory of predicate-complete test coverage and generation," in *FMCO'04*, ser. LNCS, vol. 3657, 20005, pp. 1–22.

[17] F. Bouquet, P.-C. Bué, J. Julliand, and P.-A. Masson, "Génération de tests à partir de critères dynamiques de sélection et par abstraction," in *AFADL'09*, Toulouse, France, Jan. 2009, pp. 161–176.

[18] R. Marlet and C. Mesnil, "Demoney: A demonstrative electronic purse – card specification," Trusted Logic, Tech. Rep. SECSAFE-TL-007, 2002.

[19] F. Bouquet, B. Legeard, and F. Peureux, "CLPS-B: A constraint solver to animate a B specification," *Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 143–157, 2004.

[20] P. Godefroid and R. Jagadeesan, "On the expressiveness of 3-valued models," in *VMCAI'03*, ser. LNCS, vol. 2575, 2003, pp. 206–222.

[21] J. Calamé, N. Ioustinova, and J. van de Pol, "Automatic model-based generation of parameterized test cases using data abstraction," *ENTCS*, vol. 191, pp. 25–48, 2007.

[22] G. Fraser, M. Weiglhofer, and F. Wotawa, "Coverage based testing with test purposes," in *QSIC'08*, 2008, pp. 199–208.

[23] P. Ammann, P. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM'98*. IEEE, 1998, pp. 46–54.

[24] H. Hong, I. Lee, O. Sokolsky, and H. Ural, "A temporal logic based theory of test coverage and generation," in *TACAS'02*. Springer, 2002, pp. 327–341.