Model-Based Testing using Symbolic Animation and Machine Learning

Extended Abstract

Pierre-Christophe Bué, Frédéric Dadeau and Pierre-Cyrille Héam
LIFC - Université de Franche-Comté - INRIA CASSIS
16 route de Gray - 25030 Besançon, France
Email: {pierre-christophe.bue, frederic.dadeau, pierre-cyrille.heim}@lifc.univ-fcomte.fr

Abstract—We present in this paper a technique based on symbolic animation of models that aims at producing model-based tests. In order to guide the animation of the model, we rely on the use of a deterministic finite automaton (DFA) of the model that is built using a well-known learning machine algorithm from Angluin, considering a complex model as a black-box component that can be animated. Since the DFA obtained in this way may be an over-approximation and, thus, admit traces that were not admitted on the original model, this abstraction is refined using counter-examples made of unfeasible traces. The computation of counter-examples is performed using a systematic coverage of the DFA states and transitions, producing test sequences that are replayed on the model, providing either test cases for offline testing, or counter-examples that aim at refining the abstraction.

Keywords—symbolic animation, constraint solving, machine learning, abstraction refinement

I. INTRODUCTION

In the context of Model-Based Testing [1], various approaches involve the use of a graph-based formalizations of the system based on states and transitions, such as Finite State Machine [2], or (IO)LTS [3]. On the opposite, other approaches consider textual descriptions as a model based on Generalized Substitutions, such as B [4], or more generally pre/postconditions formalisms, such as VDM [5]. In many cases, textual models are first turned into a graph that is then explored using dedicated algorithms, such as the transition tour, or using a test purpose that restricts the possible model executions. When it comes to large systems, the enumeration of reachable states is costly, and requires symbolic techniques [6] or approximation [7] in order to reduce the number of considered states to a tractable value.

Symbolic animation of models makes it possible to explore potentially large state spaces by replacing operation parameters by symbolic variables, that create symbolic states, from which the animation goes on. While efficient for animating given operation sequences, it may hardly be employed for building a state machine since the detection of symbolic state inclusion requires complex constraint solving mechanisms that may dramatically slow the computation.

Abstraction of systems are widely used in order to eliminate irrelevant details and to obtain simple models. However, abstracting systems introduces a loss of precision and may produce false executions (executions that can be run on the abstraction but not on the system). In order to get better/finer abstractions the paradigm of Counter Example Guided Abstraction Refinement was introduced in [8]: from an abstraction and a counter-example, a new finer abstraction refusing this execution is build. This approach was successfully used in many contexts, such as symbolic model-checking, timed automata [9], or rewriting-based analysis [10].

Machine learning is usually employed for extracting a FSM out of a black-box component. The basic idea is to exercise the system, through membership queries using its possible inputs, and infer a Deterministic Finite Automaton (DFA) from the answers provided by a teacher. In these algorithms, the resulting machine may not be correct w.r.t. the behavior of the system, and needs to be refined by taking into account counter-examples that are provided by an oracle, able to determine the equivalence of two machines (i.e. assuming that the oracle knows the SUT).

We propose in this paper to adapt these model learning approaches by combining them with symbolic animation of the model. We aim at inferring a DFA providing an abstraction of a B model, from which tests cases can be computed using classical graph coverage algorithms. The basic idea of the approach is replayed the generated sequences on the model. All test sequences that succeed (i.e. that are feasible when being animated on the model) provide model-based test cases, whereas the failed ones will provide counter-examples that will be used to refine the DFA.

First, we introduce, in Section II, the formalism that is considered in this work, namely the B abstract machines. We present in Section III the principles of symbolic animation that use constraint logic programming. We then present in Section IV machine learning algorithm we consider, introducing the computation of test cases/counter-examples and their exploitation for refining the DFA. We give some experimental data in Section V. Finally, we conclude and present the future works in Section VI.

II. SYMBOLIC ANIMATION OF B ABSTRACT MACHINES

We present in this section the formalism we consider, namely the B abstract machines, and we describe the principles of symbolic animation on this notation.
A. The B Method

The B method [4] is dedicated to the formal development, from high level specification to implementable code. Specifications are based on three formalisms: data are specified using a set theory, properties are first-order predicates and the behavioral part is specified by Generalized Substitutions.

The B method starts by the writing of a formal specification, named abstract machine, that gives a functional view of the system. The machine is then spiced up with invariant properties that represent properties that have to hold at each state of the system execution. It means that (i) the initialization has to establish the invariant, (ii) the operations have to preserve the invariant (meaning that if the invariant is satisfied before the operation, then it also has to be satisfied after the execution of the operation). Operations are written in terms of Generalized Substitutions that are built on basic assignments, composed into more generalized and expressive structures, that may for example, represent conditional substitutions (IF...THEN...ELSE...END) or non-deterministic buildings (CHOICE, ANY).

The B method has been applied for many industrial applications, particularly in the railway domain (e.g. line 14 of the Paris subway called METEOR [11]) and in the context of Java-card application or environment [12].

In this work, we do not take B refinements into account, we only focus on abstract machines. Notice that this is not a restriction, since refinements of machines can be flattened to a single B machine.

B. A Running Example

We introduce here a simple example of a B abstract machine that describes a simple process scheduler. In this example, given in Fig. 1, processes try to access a critical section. The system thus manages a waiting queue for processes ready to be activated. Four operations make it possible to respectively create (new) or delete (delete) a process, make a process request the access to the critical section (ready), or remove the active process, so that an idle process may be activated (swap). The data model is represented by three variables representing the set of existing processes that do not request the critical section (waiting), processes ready to access the critical section as soon as it is released (ready) and active processes (active – containing at most one element).

When a B model is animated, the user chooses which operation he wants to invoke. Depending on the current state of the system and the values of the parameters, different resulting states can be obtained. We now describe the principle of symbolic animation.

III. Symbolic Animation using CLP

The symbolic animation improves the “classical” model animation by giving the possibility to abstract the operation parameters. Once a parameter is abstracted, it is replaced by a symbolic variable that is handled by dedicated constraint solvers. Abstracting all the parameter values turns out to consider each operation as a set of “behaviors”, that are now described.

A. Definition of the Behaviors

The Prolog animation engine of BZ-Testing-Tools [13] that we use relies on a decomposition of the B machines operations into behaviors. Each behavior is defined as a predicate, representing its activation condition, and a substitution that indicates the evolution of the state variables and the instantiation of the return parameters of the operation. These behaviors are computed as the paths in the control flow graph of the considered B operation. They consist in two parts: an activation condition and a substitution.

Example 1 – Computation of behaviors. Consider the ready operation given in Fig. 1. It presents 2 behaviors, that are the following (for each behavior, the symbol is used to separate the activation condition and the substitution):

\[ b_1 : pp \in \text{waiting} \land \text{active} = \emptyset \implies \]
\[ \text{waiting} := \text{waiting} - \{pp\} \land \text{active} := \{pp\} \]
\[ b_2 : pp \in \text{waiting} \land \text{active} \neq \emptyset \implies \]
\[ \text{waiting} := \text{waiting} - \{pp\} \land \text{ready} := \text{ready} \cup \{pp\} \]

The decomposition of operations into behaviors gives equivalence classes in terms of resulting states after the execution of the operation.

B. Use of the Behaviors for the Symbolic Animation

When performing the symbolic animation of a B model, the operation parameters are abstracted and, thus, operations are considered through their behaviors. Each parameter is
replaced by a symbolic variable whose value is managed by a constraint solver.

All state variables that are related to the abstracted parameter (e.g. assigned in a substitution), also become symbolic variables, linked to the corresponding parameter. A system state that contains at least one symbolic state variable is said to be a symbolic state.

**Example 2 – Activation of a single behavior**. Consider behavior $b_1$ from the previous example. Suppose it is invoked (using $\text{ready}(X_1)$) from a system state in which three processes have just been created using three successive invocations of the $\text{new}$ operation ($\text{init}; \text{new}(p1); \text{new}(p2); \text{new}(p3)$):

$$\text{waiting} = \{p1,p2,p3\}, \text{ready} = \text{active} = \emptyset$$

The subsequent activation of $b_1$ results in the following symbolic state:

$$\text{waiting} = X_2, \text{ready} = \emptyset, \text{active} = X_3$$

with the following set of constraints:

$$X_1 \in \{p1,p2,p3\}, X_2 = \{p1,p2,p3\} - \{X_1\}, X_3 = \{X_1\}$$

The symbolic animation process works by exploring the successive behaviors of the considered operations. When two operations have to be chained, this process acts as an exploration of the possible combinations of successive behaviors for each operation.

In practice, the selection of the behaviors to be activated is done in a transparent manner and the enumeration of the possible combinations of behaviors chaining is explored using backtracking mechanisms. For animating B models, we use CLPS-BZ [13], a set-theoretical constraint solver written in SICStus Prolog that is able to handle a large subset of the data structures existing in the B machines (sets, relations, functions, integers, atoms, etc.).

**Example 3 – Combination of behaviors**. Suppose we restart the animation from the symbolic state reached at the end of the previous example, and we invoke $\text{ready}(X_4)$. Behavior $b_1$ will not be activable, since $\text{active}$ is not empty, but the activation $b_2$ will cause the following state to be reached:

$$\text{waiting} = X_2, \text{ready} = X_5, \text{active} = X_3$$

with the following set of constraints:

$$X_1 \in \{p1,p2,p3\}, X_2 = \{p1,p2,p3\} - \{X_1\}, X_3 = \{X_1\}, X_4 \in X_2, X_5 = \{X_4\}$$

Once the sequence has been symbolically played, the remaining symbolic parameters can be instantiated by a simple labelling procedure, that consists in solving the constraints system and produce an instantiation of the symbolic variables. This makes it possible to produce an instantiated test case.

**Example 4 – Test case instantiation**. Suppose we restart the animation from the symbolic state reached at the end of the previous example. The first possible instantiation of the symbolic variables that satisfies the constraints is the following:

$$X_1 = p1, X_2 = \{p2,p3\}, X_3 = \{p1\}, X_4 = p2, X_5 = \{p2\}$$

which produces the following test case:

$$\text{init}; \text{new}(p1); \text{new}(p2); \text{new}(p3); \text{ready}(p1); \text{ready}(p2)$$

The consistency of the resulting constraints system defines the feasibility of the operation sequence; this latter is said to be feasible if and only if there exists at least one solution that assigns a correct value to the variables, according to their related constraints.

**IV. BUILDING AN ABSTRACTION USING A LEARNING ALGORITHM**

We now present the use of a learning algorithm for producing a DFA from which the tests will be computed. We first recall the notions of machine learning before presenting their adaptation using symbolic animation of models.

**A. Machine Learning Algorithms**

The objective of a machine learning algorithm is to infer a finite machine model providing a description in terms of states and transitions of how the considered system behaves. Machine learning algorithms are basically made of three main entities that interact altogether, as depicted in Fig. 2.

The first entity is the learner. He is intended to build the DFA from the informations he receives. The second entity is the one that provides the informations to the learner. It might be a teacher who gives correct and erroneous samples of the possible system execution. It is also possible that the learner directly interacts with the system, so as to ask membership
queries and observe if the proposed executions are accepted or refused by the system. Once a satisfying DFA has been produced, the learner asks the oracle if the inferred machine is equivalent to the actual system or not. The oracle answers by "yes", or "no" and provides a counter-example that can be taken into account so as to improve the DFA and start the process over again.

One of the most famous implementation of this algorithm is $C^*$ proposed by D. Angluin [14], which proposes the inference of a Mealy machine based on the following principles: the learner gathers the executions into observation tables that store the prefixes that have been tried on the system and their corresponding outputs. When two prefixes produce the same outputs in response to all the possible inputs (given by the system alphabet), then the states reached after the execution of these two prefixes are merged into the same state.

The major issue in this process is the existence of an almighty oracle that is able to check the equivalence of the inferred DFA and the actual system. A solution has been proposed by Vasilevski and Chow [15], [2] when the two compared systems are known as Finite State Machines. The basic idea is to compute test sequences from one of the machines. If the other passes the tests, then the two machines are declared as equivalent.

Variants of this algorithm are based on changing the representation of the observations, either by considering a Reduced Observation Table [16], or by considering trees [17] instead of sets to store the observations made from the considered system. This latter solution has been selected in our approach and will now be explained.

B. Building the DFA using Discrimination Trees

We first recall the definition of a Deterministic Finite Automaton before presenting the learning algorithm that aims at building one using symbolic animation of models.

**Definition 1 – Deterministic Finite Automaton.** The Deterministic Finite Automata (DFA) that we consider are quadruplets $⟨Q, \Sigma, \delta, q_0⟩$ in which

- $Q$ is the finite set of states,
- $\Sigma$ is the alphabet, namely, the set of all behaviors,
- $\delta$ is the partial transition function $Q \times \Sigma \rightarrow Q$, and
- $q_0$ is the initial state.

The idea of the learning algorithm that we employ is to explore the model executions using symbolic animation. Each state is, at first, identified by a set of behaviors that can be activated from the point. As a consequence, the exploration of the model executions stops (for the current execution branch) when a state, from which the same set of behaviors is activable as an already visited state, has been reached.

The algorithm is given in Fig. 3. It returns a tree identified by its root node, and a set of merging operations that have to be done to obtain the DFA from the graph. Each merging

```
merging, root \rightarrow \text{procedure} \text{learn( )}
begin
merging \leftarrow \emptyset
visited \leftarrow \emptyset
s_1 \leftarrow \text{init}
visited.push(⟨s_1, \text{root}⟩)
while \text{visited not empty do}
⟨s, \text{node}⟩ \leftarrow \text{visited.pop}
LB \leftarrow \text{get activable behaviors}(s)
if \exists⟨s', \text{node}'⟩ \text{ s.t. } LB = \text{get activable behaviors}(s) \text{ then}
merging \leftarrow \text{merging} \cup \{\text{node} \mapsto \text{node}'\}
else
for each $b$ in LB do
  $s_n \leftarrow \text{activate behavior}(s, b)$
  node$_n \leftarrow \text{node}.\text{newChild}(s_n)$
  visited.push(⟨$s_n$, node$_n$⟩)
end
end
```

Figure 3. Machine Learning Algorithm using Symbolic Animation.

is of the form $s' \mapsto s$ meaning that state $s'$ has to be merged with state $s$. The algorithm keeps track of a stack of visited states from which the exploration has to continue. Symbolic animation procedures are: $\text{get activable behaviors}(s)$ that returns the set of behaviors that can be activated from a given (possibly symbolic) state $s$, and $\text{activate behavior}(s, b)$ that computes the state resulting from the activation of behavior $b$ from state $s$.

Although the algorithm is supposed to terminate (the worst case number of possible states being $2|\Sigma|^1$), it is, in practice, bounded in depth to increase its convergence in presence of large systems. Notice that the exploration of the model execution is done using a depth-first algorithm using Prolog’s internal backtracking mechanism.

Once the tree has been built, it is then explored so as to build the corresponding automaton, as illustrated hereafter.

**Example 5 – From Observation Tree to DFA.** Figure 4 illustrates the transformation of an observation tree into a DFA. In this tree, the nodes that are merged are labelled by the node identifier they are merged with (discriminating behaviors being represented with dotted arrows).
C. Checking the Equivalence with the Original B Model

Similarly to Vasilevski [15] and Chow [2], we propose to build tests from the DFA we obtain, and replay them on the original B model. For each test, if the test passes, then it is kept to be later on concretized to be run on the SUT. If the test fails, i.e. if at some (break)point, the sequence is no more feasible, then it represents a counter-example.

The technique for derivate tests from a DFA is based on the exploration of the graph, using well-known techniques, such as random walks, greedy algorithms or transition tours (e.g. using the chinese postman algorithm [18]).

Example 6 – Test cases generation. Consider the example DFA built from the execution tree depicted in Fig. 4. An example of test sequences that achieve all-transitions coverage is given by at least two sequences, such as \( b_3\cdot b_2\cdot b_1\cdot b_2 \) and \( b_2\cdot b_1\cdot b_2 \).

Tests are replayed on the model, using symbolic animation activating the successive behaviors of the sequence. Once the animation is over, the test case is instantiated, i.e. symbolic parameters are assigned to a value, as explained in Example 4.

D. Processing Counter-Examples

All sequences that are not feasible represent counter-examples, i.e. traces that are accepted by the inferred DFA but not by the original system. In the context of machine learning, these counter-examples are processed in the observation tree.

Basically, the revision procedure works as follows. If the trace is not feasible, then it necessarily covers a node that has been merged by error. The process of the counter-example consists in, first, identifying the “breakpoint” –the node that should not have been merged. Then, we add this trace in the tree from this point until the last activable behavior of the sequence. Each step will thus represent a new state that will not be merged with other existing nodes, but for which the successors have to be computed. The breakpoint is defined as the first merged state that is encountered when monitoring the trace execution on the observation tree.

Example 7 – Counter-example processing. Consider again the example DFA built from the execution tree depicted in Fig. 4. Suppose that test sequence \( b_3\cdot b_2\cdot b_1\cdot b_2 \) is not feasible when being replayed on the original B model, due to an impossibility to activate the last behavior \( b_2 \). The following indicates the position of merged states covered by the trace: \( b_3\cdot b_2\cdot(2)\cdot b_1\cdot(2)\cdot b_2 \).

The first state that has been merged (after \( b_3\cdot b_2 \)) is thus marked as unique, and the rest of the trace is added in the tree, creating new states, from which the learning process has to restart, as shown in Fig. 5.

Once all counter-examples have been processed, the evaluation restarts until no counter-example are found. In practice, counter-example can be considered as test cases if the feasible behavior sequence is of acceptable (user-defined) length. For example, in presence of a test case made of 100 steps that fails at the last step, the 99 first steps are kept as a test case, and no counter-example processing is done on the DFA.

V. Experimental Results

We have first applied our technique on the toy example of the scheduler presented in this paper, before experimenting a realistic case study of an electronic purse, named Demoney. The results are presented in Table I.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Scheduler</th>
<th>Demoney</th>
</tr>
</thead>
<tbody>
<tr>
<td># concrete states</td>
<td>124</td>
<td>&gt; 10^97</td>
</tr>
<tr>
<td># behaviors</td>
<td>6</td>
<td>93</td>
</tr>
<tr>
<td>Abstraction computation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation time</td>
<td>9s</td>
<td>4min 28s</td>
</tr>
<tr>
<td># states/transitions</td>
<td>9/24</td>
<td>18/496</td>
</tr>
<tr>
<td># covered behaviors</td>
<td>6/6 (100%)</td>
<td>??93 (?? %)</td>
</tr>
<tr>
<td># tests Chinese Postman</td>
<td>1 (0 feasible)</td>
<td>5 (4 feasibles)</td>
</tr>
<tr>
<td>Refinement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>states/transitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td># tests Chinese Postman</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I

To improve the results we obtained, we decided to strengthen the merging criterion that we employed in the original algorithm. Instead of computing the activable behaviors, we compute the pairs of successive activable behaviors. Results are given in Table II. Unsurprisingly, the computation time is increased, so is the number of states and transitions that compose the abstraction. Nevertheless, the computed abstractions are both “correct” since the tests produced on them using the Chinese Postman algorithms are all replayable on the original B model, avoiding refinement(s) step(s).

Unfortunately, on the Demoney case study, the abstraction does not present all the behaviors of the original specifica-
tion. Notice that a large majority of them –41– are error behaviors from the same operation.

VI. CONCLUSION AND FUTURE WORKS

We have presented in the paper the use of a learning algorithm inferring a DFA from the symbolic animation of a B model. The learning algorithm is a variant of Angluin algorithm, in which the state merging criterion considers that two states are identical if they display the same set of activable behaviors. This approach makes it possible to design a first abstraction of the B model states that is exercised using graph exploration algorithm building test cases (as a sequence of behaviors) that aim at testing the abstraction. These test cases are then replayed on the model. If a test case execution fails, it is considered as a counter-example that helps improving the abstraction. Otherwise, the test case is kept to be played on an actual System Under Test. The experimentations have shown the potential interest of the approach, when using a merging criterion based on activable transitions pairs.

We are now investing ways to increase the coverage of the behaviors from the original model covered in the abstraction. To do that, we can either strengthen the merging criterion or use external model-based test cases, built from the original model, that can be used as counter-example helping the refinement step.

ACKNOWLEDGMENTS

The authors would like to thank Roland Groz and Musaemmil Shabhaz from Grenoble Universities for fruitful discussions during the TAROT Summer School 2009.

REFERENCES


