Model learning and test generation using cover automata

FLORENTIN IPATE [1,2,*], ALIN STEFANESCU [1,2,†], IONUT DINCA [2,‡]

[1] *Dept. of Computer Science, University of Bucharest, Romania*
[2] *Dept. of Computer Science, University of Pitesti, Romania*
*Email: *florentin.ipate@ifsoft.ro, †alin@stefanescu.eu, ‡ionut.dinca@upit.ro*

We propose an approach which, given a state-transition model of a system, constructs, in parallel, an *approximate automaton* model and a *test suite* for the system. The approximate model construction relies on a variant of Angluin's automata learning algorithm, adapted to finite cover automata. A *finite cover automaton* represents an approximation of the system which only considers sequences of length up to an established upper bound ℓ. Crucially, the size of the cover automaton, which normally depends on ℓ, can be significantly lower than the size of the exact automaton model. Thus, controlling ℓ, the state explosion problem normally associated with constructing and checking state based models can be mitigated. The proposed approach also allows for a gradual construction of the model and of the associated test suite, with complexity and time savings. Moreover, we provide automation of counterexample search, by a combination of black-box and random testing, and metrics to evaluate the quality of the produced results. The approach is presented and implemented in the context of the Event-B modeling language, but its underlying ideas and principles are much more general and can be applied to any system whose behavior can be suitably described by a state-transition model.

Keywords: Automata learning; Model-based Testing; Formal Methods; Event-B; Angluin's Algorithm

Received 31 May 2013; revised 14 March 2014

1. INTRODUCTION

Testing is a major part of system development and has a great impact on the quality of the delivered product. Model-based test generation involves the use of a system model for selecting test data and offers the potential for automation. The concept of state is at the heart of model-based testing and many test generation techniques from finite state machines (FSMs) exist [1]. However, FSMs are not powerful enough to efficiently model realistic systems and so extended finite state machines (EFSMs), such as Statecharts [2], are used instead; these combine a FSM-like control with suitable data variables and operations for these variables, to offer an intuitive, yet rigorous means for system modeling and analysis. Testing from an EFSM, that can allow strong statements to be made about system correctness, as it is the case with, for example, conformance testing [1], usually involves transforming the EFSM into an equivalent FSM (whose states are given by the state-variable value combinations of the original EFSM) and then applying FSM-based test generation techniques.[1] However, for many systems, the equivalent FSM may have many more states than the length of the tests that can realistically be performed, or, furthermore, the number of states of the resulting FSM may be so large that it is impossible to even construct it. This is the well-known state explosion problem. Despite the existence of numerous techniques for alleviating this problem [1, 3, 4], state explosion remains one of the major obstacles for efficient model-based test generation from state-based models.

In this paper we propose an approach which, given a state-transition model (EFSM) of a system, constructs, in parallel, an *approximate FSM* model and a *test suite* for the system. The (approximate) model construction relies on a variant of Angluin's automata learning algorithm [5], adapted to finite cover automata [6]. A *finite cover automaton* [7, 8] of a finite set U is a finite automaton which accepts all sequences in U but may also accept sequences that are longer than every sequence in U. In practice, an upper bound ℓ on the length of the considered sequences from a regular language L will be established and the constructed model will have to conform to the original model

[1] Note that it is possible to derive tests directly from a EFSM, e.g. randomly or based on representative scenarios, but usually, in such cases, one cannot make theoretically proven claims about the level of correctness achieved through testing.

for all sequences of length at most ℓ. The main advantage of a finite cover automaton is that its size (number of states), which normally depends on ℓ, can be significantly lower than the size of the automaton that accepts exactly the language L [7], as illustrated by examples provided in Section 4. As also shown in Section 4, these smaller automata can be sufficient for testing purposes. In this way, by appropriately setting the value of the bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be kept under control. Note that, in order to take full advantage of the reduced size of this approximation, it is essential that a cover automaton of U, and not the exact automaton that accepts U, is constructed (in some cases, it is possible that the automaton that accepts U is even larger than the automaton that accepts L). Furthermore, test generation from finite cover automata fits very well with common testing practices, that usually require test cases of short to medium length and can also be regarded as a natural complement to Bounded Model Checking (BMC) based on SAT methods [9, 10], which is gaining popularity in the formal verification community.

The proposed approach also allows for a gradual construction of the model and of the associated test suite: the FSM model and test suite for an initial version of the system are reused in the construction of a more elaborate and complex version, with complexity and time savings, but also with improvements in the precision of the obtained models and tests.

The approach is presented and implemented in the context of the Event-B modeling language, but its underlying ideas and principles are much more general and can be applied to any system whose behavior can be suitably described by a state-transition model. As Event-B does not even distinguish between state and data variables, such models are very suitable means for evaluating our approach. Event-B [11], an evolution of the B language, is a formal method for reliable systems specification and verification. The theoretical foundations and associated tooling were developed in several research projects, among which the most notable are two large European research projects: RODIN[2], which produced a first platform for Event-B called *Rodin*[3], and DEPLOY[4], which enhanced the platform based on industrial feedback. The platform offers several plugins for the editing, refinement, theorem-proving, composition or model-checking of Event-B models. Complementing these techniques, model based testing has recently emerged as an interesting research theme for Event-B [12, 13], supported by increasing interest from the industrial partners (in the DEPLOY consortium) like SAP [14].

Given an Event-B model and an upper bound ℓ, the proposed approach will incrementally construct finite cover automata that will eventually accept all executable sequences of length less than or equal to ℓ. A new increment is constructed when (a) the Event-B model has been modified or augmented due to changes in the requirements, (b) the Event-B model has not been changed but the associated DFCA is deemed not to be sufficiently precise, or (c) the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). As a by-product of the automata learning algorithm, a *set of test cases* associated with the cover automata is also maintained and evolved during the iterations. This test suite can be used for conformance testing of the modeled system. The test cases in the test suite are provided together with the associated *test data* that makes them executable on the Event-B model.

The contributions of the paper are fourfold:

- first, it constructs an incremental set of finite approximation models for the set of Event-B executable traces up to a length ℓ. The construction exploits the restriction given by the bound ℓ to obtain models of reduced size (number of states) compared to exact automaton models. Moreover, the cover automata are minimal by construction.
- second, in parallel with automata construction, we incrementally generate conformance test suites for the investigated Event-B models. By construction, the generated test cases satisfy certain minimality properties regarding their lengths. This fits very well with the testing practice that usually requires short test cases.
- third, the Event-B method deploys model refinement as a means to handle modeling complexity. This, along with the two contributions above, can be applied incrementally, allowing the reuse of the learned model and test cases from the abstract to the more concrete levels. Moreover, the approach can also be adapted to Event-B decomposition.
- finally, we provide two solutions to improve the quality of the learned models. First, the search for counterexamples that are used to improve the model is automated using a W-method procedure adapted to bounded sequences. Second, we propose two metrics on the models to guide the exploration depth, i.e. to control the bound ℓ.

We recently published two conference papers [13, 15] with some overlap with the the current material, but their content is largely complementary: [13] is a short tool paper describing the integration of our prototype in the Rodin platform and [15] presents how our learning approach can be used in conjunction with Event-B decomposition. We only touch upon this at the end of Section 3 for completeness.

[2]`http://rodin.cs.ncl.ac.uk` - FP6 project running between 2004-2007

[3]`http://sourceforge.net/projects/rodin-b-sharp`

[4]`http://deploy-project.eu` - FP7 project running between 2008-2012

The paper has the following structure. Section 2 presents the theoretical foundations, including the algorithm used for cover automata. Sections 3 and 4 describe the proposed approach and its empirical evaluation, respectively. Section 5 discusses related work, while Section 6 concludes the paper.

2. THEORETICAL BACKGROUND

This section, which is largely adapted from our previous work [6], presents the L^ℓ algorithm and its automata-related concepts.

Before continuing, we introduce the notations used in the paper. For a finite alphabet A, A^* denotes the set of all finite sequences with members in A. ϵ denotes the empty sequence. For a sequence $a \in A^*$, $\|a\|$ denotes the length of a; in particular $\|\epsilon\| = 0$. For a finite set of sequences $U \subseteq A^*$, $\|U\|$ denotes the length of the longest sequence(s) in U. For $a, b \in A^*$, ab denotes the concatenation of sequences a and b. a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$, $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$, $n \geq 1$. $A[n] = \bigcup_{0 \leq i \leq n} A^i$ denotes the sets of sequences of length less than or equal to n with members in the alphabet A. For a sequence $a \in A^*$, $b \in A^*$ is said to be a *prefix* of a if there exists a sequence $c \in A^*$ such that $a = bc$. The set of all prefixes of a is denoted by $pref(a)$; for $U \subseteq A^*$, $pref(U) = \bigcup_{a \in U} pref(a)$. For a sequence $a \in A^*$, $b \in A^*$ is said to be a *suffix* of a if there exists a sequence $c \in A^*$ such that $a = cb$. For a finite set A, $card(A)$ denotes the number of elements in A.

2.1. Finite automata - general concepts

We start by introducing some classic definitions from automata theory.

A *deterministic finite automaton* (*DFA*) M is a tuple (A, Q, q_0, F, h), where: A is the finite *input alphabet*; Q is the finite *set of states*; $q_0 \in Q$ is the *initial state*; $F \subseteq Q$ is the *set of final states*; h is the *next-state*, $h : Q \times A \longrightarrow Q$. A DFA is usually described by a *state-transition diagram.*

The next-state function h can be naturally extended to a function $h : Q \times A^* \longrightarrow Q$. A state $q \in Q$ is called *reachable* if there exists $s \in A^*$ such that $h(q_0, s) = q$. M is called *reachable* if all states of M are reachable.

Given $q \in Q$, the set L_M^q is defined by $L_M^q = \{s \in A^* \mid h(q, s) \in F\}$. When q is the initial state of M, the set is called the *language accepted by* M and the simpler notation L_M is used. Given $Y \subseteq A^*$, two states $q_1, q_2 \in Q$ are called Y*-equivalent* if $L_M^{q_1} \cap Y = L_M^{q_2} \cap Y$. Otherwise q_1 and q_2 are called Y*-distinguishable.* If $Y = A^*$ then q_1 and q_2 are simply called *equivalent* or *distinguishable*, respectively. Two DFAs are called (Y-)equivalent or (Y-)distinguishable if their initial states are (Y-)equivalent or (Y-)distinguishable, respectively.

A DFA M is called *reduced* if every two distinct states of M are distinguishable. A DFA M is called *minimal* if any DFA that accepts L_M has at least the same number of states as M. A DFA M is minimal if and only if M is reachable and reduced [16]. Furthermore, there is a unique (up to a renaming of the state space) minimal DFA that accepts a given regular language [16].

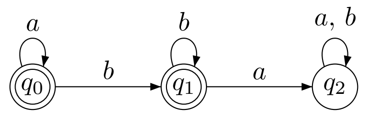


FIGURE 1. A minimal DFCA of $U(k)$ w.r.t. k

Now let us also introduce the concept of *deterministic finite cover automaton* (DFCA). Informally, a DFCA of a finite language U, as defined by Câmpeanu et al. [7], is a DFA that accepts all sequences in U and possibly other sequences that are longer than any sequence in U.

In this paper we use a slightly more general concept, as defined in [6]: given a finite language $U \subseteq A^*$ and a positive integer ℓ that is greater than or equal to the length of the longest sequence(s) in U, a *deterministic finite cover automaton* (*DFCA*) of U w.r.t. ℓ is a DFA M that accepts all sequences in U and possibly other sequences that are longer than ℓ, i.e. $L_M \cap A[\ell] = U$. A DFCA M of U w.r.t. ℓ is called *minimal* if any DFCA of U w.r.t ℓ has at least the same number of states as M. Note that, unlike the case in which the acceptance of the exact language is required, the minimal DFCA is not necessarily unique (up to a renaming of the state space) [6].

Naturally, a DFA that accepts a finite language U is also a DFCA of U w.r.t. any $\ell \geq \|U\|$. Consequently, the number of states of a minimal DFCA of U w.r.t. ℓ will not exceed the number of states of the minimal DFA accepting U. Furthermore, the size of a minimal DFCA of U w.r.t. ℓ can be much smaller than the size of the minimal DFA that accepts U [6]. Consider, for example, $U(k) = \{a^i b^j \mid i \geq 0, j \geq 0, i + j \leq k\}$ for $k \geq 2$. The minimal DFA that accepts $U(k)$ will have $2k + 1$ states: one final state corresponding to the empty sequence, two final states corresponding to every sequence of length i, $1 \leq i \leq k - 1$, (one state for when the sequence is composed only of a's and one state for when the last input read is a b), one final state corresponding to sequences of length k and one non-final state [6]. On the other hand, a DFCA of $U(k)$ w.r.t. $k \geq 2$ may have only 3 states, as shown in Fig. 1.

Furthermore, using the same argument as above, if $L \subseteq A^*$ is a regular language such that $L \cap A[\ell] = U$, the minimal DFA that accepts L will be at least the size of the minimal DFCA of U w.r.t. ℓ.

2.2. The L^ℓ algorithm for learning finite cover automata

Learning regular languages from queries was introduced by Angluin in [5]; the paper also provides a learning algorithm, called L^*. The L^* algorithm infers a regular language, in the form of a DFA from the answers to a finite set of membership queries and equivalence queries. A *membership query* asks whether a certain input sequence is accepted by the system under test or not. In addition to membership queries, L^* uses *equivalence queries* to check whether the learning algorithm is completed.

In a recent paper [6], we extended Angluin's work by proposing an algorithm, called L^ℓ, for learning a DFCA. Given an unknown finite set U and a known integer ℓ that is greater than or equal to the length of the longest sequence(s) in U, the L^ℓ algorithm will construct a minimal DFCA of U w.r.t. ℓ. Analogously to L^*, the L^ℓ algorithm uses membership and language equivalence queries to find the automaton in polynomial time.

The L^ℓ algorithm constructs two sets: S, a non-empty, prefix-closed set of sequences and W, a non-empty, suffix-closed set of sequences. Additionally, S will not contain sequences longer than ℓ and W will not contain sequences longer than $\ell - 1$, i.e. $S \subseteq A[\ell]$ and $W \subseteq A[\ell - 1]$.

The algorithm keeps an *observation table*, which is a mapping T from a set of finite sequences to $\{0, 1, -1\}$. The sequences in the table are formed by concatenating each sequence of length at most ℓ from the set $S \cup SA$ with each sequence from the set W. Thus, the table can be represented by a two-dimensional array with rows labeled by elements of $(S \cup SA) \cap A[\ell]$ and columns labeled by elements of W.

The function $T : ((S\cup SA)\cap A[\ell])W \longrightarrow \{0, 1, -1\}$ is defined by $T(u) = 1$ if $u \in U$, $T(u) = 0$ if $u \in A[\ell] \setminus U$ and $T(u) = -1$ if $u \notin A[\ell]$. The values 0 and 1, respectively, are used to indicate whether a sequence is contained in U or not. However, only sequences of length less than or equal to ℓ are of interest. For the others, an extra value, -1, is used.

In order to compare the rows in the observation table, a relation on these rows, called *similarity*, is used. We say that rows s and t are *k-similar*, $1 \leq k \leq \ell$, and write $s \sim_k t$ if, for every $w \in W$ with $\|w\| \leq k - max\{\|s\|, \|t\|\}$, $T(sw) = T(tw)$. Otherwise, s and t are said to be *k-dissimilar*, written $s \not\sim_k t$. In other words, the table values of rows s and t must coincide for every column w for which the lengths of sw and tw are both less than or equal to k. The relation $\sim_k$ is not an equivalence relation since it is not transitive [6]. When $k = \ell$, we simply say that s and t are *similar* or *dissimilar* and write $s \sim t$ or $s \not\sim t$, respectively. It can be observed that similarity of rows s and t requires all corresponding non-negative values of the two rows to coincide.

Using the similarity relation, two properties of an observation table are defined: consistency and closedness.

The observation table is *consistent* if, for every k, $1 \leq k \leq \ell$, whenever rows $s_1 \in S$ and $s_2 \in S$ are k-similar, rows $s_1 a$ and $s_2 a$ are also k-similar for all $a \in A$.

T	ϵ	a
ϵ	0	0
a	0	1
b	0	0
aa	1	0
bb	1	0
ab	0	0
ba	0	0
aaa	0	-1
aab	0	-1
bba	0	-1
bbb	0	-1

T	ϵ	a	b
ϵ	0	0	0
a	0	1	0
b	0	0	1
aa	1	0	0
bb	1	0	0
ab	0	0	0
ba	0	0	1
aaa	0	-1	-1
aab	0	-1	-1
bba	0	-1	-1
bbb	0	-1	-1

TABLE 1. Observation table for one example (left table) and its updated form that is consistent and closed (right table)

The observation table is *closed* if, for all rows $s \in SA$, there exists row $t \in S$ with $\|t\| \leq \|s\|$, such that $s \sim t$.

Consider, for example, $A = \{a, b\}$, $\ell = 3$ and Table 1 (left hand side) - in which a double horizontal line is used to separate the rows labeled with elements of S from the rows labeled with elements of $SA \setminus S$ - to be the current observation table ($S = \{\epsilon, a, b, aa, bb\}$, $W = \{\epsilon, a\}$). The observation table is not consistent since, for $k = 2$, $s_1 = \epsilon, s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1 \alpha w) \neq T(s_2 \alpha w)$. On the other hand, the observation table is closed.

The algorithm starts with $S = W = \{\epsilon\}$. It periodically checks the consistency and closedness properties and extends the table accordingly. When both conditions are met, the DFA $M(S, W, T)$ corresponding to the table is constructed (see Fig. 3) and it is checked whether the language L accepted by $M(S, W, T)$ satisfies $L \cap A[\ell] = U$ (this is called a "language query"). If the language query fails, a counterexample t is produced, the table is expanded to include t and all its prefixes and the consistency and closedness checks are performed once more. Eventually, the language query will succeed and the algorithm will return a minimal DFCA of U w.r.t. ℓ.

Since in our approach we will separate the construction of the observation table and of the corresponding DFCA (which is the actual processing performed by the algorithm) from the language queries (which represent the user intervention), only the processing performed between two language queries is presented in pseudo-code in Fig. 2 (in what follows this will be referred to as the LearnDFCA procedure).

The *LearnDFCA* procedure starts with the current values of S, W and the current observation table T. It periodically checks whether the consistency and closedness properties are violated and extends the table by adding a new row or a new column to the table, respectively:

Procedure LearnDFCA

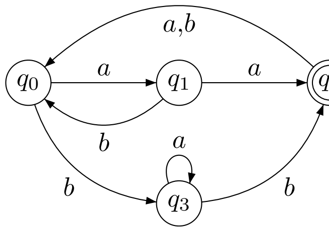
```
Input: Current observation table $T$ (includes $S$ and $W$).
Repeat
  \— Check consistency —\
  For every $w \in W$, in increasing order of $\|w\| = i$ do
    Search for $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \le \ell - i - 1$
    and $a \in A$ such that $s_1 \sim_k s_2$, where
    $k = max\{\|s_1\|, \|s_2\|\} + i + 1$, and $T(s_1aw) \neq T(s_2aw)$.
    If found then
      Add $aw$ to $W$.
      Extend $T$ to $(S \cup SA)W$ using membership queries.
  \— Check closedness —\
  Set $new\_row\_added = false$.
  Repeat for every $s \in S$, in increasing order of $\|s\|$
    Search $a \in A$ such that $sa \not\sim t \ \forall t \in S$ with $\|t\| \le \|sa\|$.
    If found then
      Add $sa$ to $S$.
      Extend $T$ to $(S \cup SA)W$ using membership queries.
      Set $new\_row\_added = true$.
  Until $new\_row\_added$ or all elements of $S$ were processed
Until $\neg new\_row\_added$
Construct $M(S, W, T)$.
Output $M(S, W, T)$.
```

FIGURE 2. The learning procedure *LearnDFCA*

- In order to check consistency, the procedure will search for $w \in W$ and $a \in A$ such that aw will distinguish between two rows s_1 and s_2 that are not distinguished by any sequences in W of length less than or equal to aw; in order to find the shortest such sequence aw, the search will be performed in increasing order of length of w. The search is repeated until all elements of W have been processed; as these are processed in increasing order of their length, any sequence aw that has been added to W as a result of an failed consistency check will be itself processed in the same "For" loop.
- In order to check closedness, the procedure will search for $s \in S$ and $a \in A$ such that sa is dissimilar to any of the current rows t for which $\|t\| \le \|sa\|$; similarly, the search is performed in increasing order of length of s. If such s and a are found, then sa is added to the observation table and the algorithm will check again its consistency.

Consider once again Table 1 (left hand side) as the current observation table. This will fail the consistency check for $i = 0$ and $k = 2$: $s_1 = \epsilon, s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1\alpha w) \neq T(s_2\alpha w)$. Consequently, $\alpha w = b$ is added to W and so Table 1 (right hand side) is the resulting observation table. This is both consistent and closed and so the DFA $M(S, W, T)$ is constructed.

The state set of DFA $M(S, W, T)$ is formed by taking



$$\begin{tikzcd}
q_0 \arrow[r, "a"] \arrow[dr, bend right, "b"'] & q_1 \arrow[r, "a"] \arrow[l, bend left, "b"] & q_2 \arrow[ll, bend right, "{a,b}"'] \\
& q_3 \arrow[loop above, "a"] \arrow[ur, bend right, "b"'] &
\end{tikzcd}$$

FIGURE 3. The DFA corresponding to Table 1 (right hand side)

all minimum, mutually dissimilar sequences from S, where the minimum is taken according to the quasi-lexicographical order on A^* [6]. For Table 1 (right hand side), this is $Q = \{\epsilon, a, b, aa\}$ (since $bb \sim aa$) and the corresponding DFA is as represented in Fig. 3 (in which final states are drawn in double line, whereas non-final states are drawn in single line; the initial state is q_0). The formal definition of $M(S, W, T)$ is given in [6], for simplicity this is not reproduced here. Further details regarding the L^ℓ algorithm, including proofs of correctness and termination and examples which illustrate its functioning, can also be found in [6].

2.3. Black-box testing for finite cover automata

The W-method for bounded sequences [17], is a non-trivial generalization of the W-method for checking functional equivalence (also called the Vasilevski-Chow method) [18, 19]. [5]

Before proceeding, we briefly outline the W-method for bounded sequences, which we proposed in [17]. This is not central to our approach, but may be used for answering language queries, as discussed later.

Given a DFCA model $M = (A, Q, q_0, F, h)$ of a system and an upper bound ℓ, this method generates a set of sequences to check if the implementation under test, which is modeled by an unknown finite automaton I, behaves as defined by M for all sequences of length at most ℓ. In other words, if the languages accepted by M and I are L_M and L_I, respectively, the W-method will construct a finite set of sequences $X \subseteq A[\ell]$ such that $L_M \cap X = L_I \cap X$ implies $L_M \cap A[\ell] = L_I \cap A[\ell]$.

The implementation is a black box and so, obviously, I is not known; however, it is assumed that the maximum number of states of I can be estimated; the difference between this estimated maximum and the number of states of M is denoted by k (if the difference is negative then we take $k = 0$).

Naturally, one can always take X to be the set of all sequences of length up to ℓ; however, the W-method produces a much smaller set, whose size is polynomial in the number of states of M (but exponential in k). The construction of X is based on two sets: a *proper*

[5] [17] gives the results for Mealy machines, whereas here we adapted them for finite state machine acceptors.

state cover S and a *strong characterization set* W of M. S is called a proper state cover of M if it contains sequences of minimum length that reach all states of M, i.e. for every $q \in Q$ there exists $s \in S$ such that: $h(q_0, s) = q$ and $\|s\| \leq \|t\|$ for every $t \in A^*$ for which $h(q_0, t) = q$. W is called a strong characterization set if it contains sequences of minimum length that distinguish between any pair of states of the DFCA, i.e. for every $q_1, q_2 \in Q$, $q_1 \neq q_2$ there exists $w \in W$ such that: w distinguishes between q_1 and q_2 and $\|s\| \leq \|t\|$ for every $t \in A^*$ which distinguishes between q_1 and q_2. Then, for an estimated value of k, the test set has the form $X_k = SA[k+1]W \cap A[\ell]$. [6]

3. MODEL LEARNING AND TEST GENERATION

We are now ready to present how we can apply the above theoretical harness. We first describe the Event-B modeling environment. Then we present our approach for incremental cover automata learning and test generation for Event-B models. The notions of refinement and decomposition are also discussed at the end of the section.

3.1. Event-B framework

Event-B method [11] is a formalism with mathematical foundations based on set theory that is used to model and prove consistency of complex systems. The modeling complexity is addressed using refinement techniques (i.e. the models are being incrementally concretized by adding extra details at each step) and composition-decomposition (i.e. the models may be decomposed in smaller sub-models that can be in the end re-composed). The verification task is performed using theorem proving machinery, i.e. mathematical proofs about the different invariants or properties of the system. Moreover, there is good support for Event-B model-checking using the ProB tool [20]. ProB provides functionality for model simulation, model animation, set-based constraint solvers, etc. The main platform supporting Event-B, integrating the different modeling and formal analysis tools, is called Rodin and is an extensible Eclipse-based tool. The current efforts of developing Event-B and Rodin are concerted in a large European project, DEPLOY, which also includes industrial partners (Bosch, Siemens, SSF, SAP) from the embedded systems and business applications domains.

The Event-B models consist of two main parts: contexts and abstract machines. A *context* provides static information like domain ranges and constants together with axioms. A *machine* describes the dynamic behavior of the system by means of global variables and events, together with invariants specifying the properties that the system is supposed to maintain during its execution. The state of the modeled system is given by the values of variables. (In Event-B there is no distinction between the model control and its data, so the state incorporates both these two aspects.) The system can change its state by executing *events* that are enabled in that state. The main elements of an event are: the local parameters, the *guard*, which is a predicate over the global variables and local parameters that decides when the event can be executed, and the *action*, which is a set of assignments that change the value of the global variables; examples of Event-B will be given in Section 3.2. Usually, in an Event-B model, all states which can be reached by feasible sequences of events are considered to be *final* states. Obviously, in this case only one non-final state is sufficient - a “sink” state which collects all infeasible paths. Although this is the situation we have encountered in all applications considered in this paper, our approach is in no way restricted to this particular case. Furthermore, it would also be possible to differentiate between reachable and final configurations in an Event-B model by using a logical predicate on the global variables, i.e. a state is considered final only if the predicate holds.

Refinement in Event-B is a mechanism of constructing a series of more abstract models before reaching a very detailed one. For instance, in a refinement step, new variables and new events can be introduced and the existing events can be made more concrete with the assumption (that must be formally proved) that the concrete guard is not weaker than the abstract one (i.e. the concrete guard logically implies the abstract one) [11].

Decomposition in Event-B: As the complexity of the model increases, so does the difficulty the proof obligations and verification tasks. One powerful method to address this situation is to decompose a larger model into smaller sub-models which can be further refined and analyzed independently [21, 22]. There are two main types of decomposition: shared events style [23] and shared variables style [24]. As the name suggest, in the former, the communication and consistency between sub-models is realized via shared events, while in the latter this is done via shared variables.

Given an Event-B model, *a test case* can be defined as a sequence of events. This can be either positive, if it corresponds to an executable (feasible) path through the Event-B model, or negative, otherwise. The executability of a test case implies the existence of appropriate test data for the events, i.e. appropriate values for the local parameters that ensure that the guard of the event is true. Finally, a *test suite* is by definition a collection of test cases.

3.2. Incremental model learning

We will apply now the cover automata learning method of Section 2 to the Event-B framework. The input

[6] The model M is assumed to be a minimal DFCA of $L_M \cap A[\ell]$ w.r.t ℓ (if not, it is minimized before the method is applied), so both a proper state cover and a strong characterization set exist.

Procedure IterativeConstructionDFCA

```
Input: $S_0$, $W_0$
Set $S = S_0$ and $W = W_0$
Construct $T$ for $(S \cup SA)W$
LearnDFCA
While the constructed automaton $M(S, W, T)$ is not correct do
        Provide a counterexample $w$
        Add $w$ and all its prefixes to $S$
        Extend the observation table $T$ to $(S \cup SA)W$
        LearnDFCA
Minimize $S$ and $W$
Output: $M(S, W, T)$, (minimized) $S$ and $W$, observation table
and the corresponding test sequences
```

FIGURE 4. The iterative procedure of constructing the DFCA

elements for the procedure were a finite language U and a bound ℓ. For an Event-B model, U will be the set of all executable event sequences of maximum length l. The alphabet of U is the set of events in the model, which we denote by A.

Given the above U and ℓ, our approach gradually constructs both (1) a DFCA for the Event-B model and (2) an associated test set. The test set will be constructed using information from the observation table (paths through the model) and the actual test data to drive the executable paths. In this subsection we discuss the model learning cycle and in Subsections 3.5 and 3.6 the test suite creation.

The proposed procedure consists of a number of steps; at each step, a new DFCA and test set is produced. The outline of this procedure is depicted in Fig. 4. Unlike the original L^ℓ algorithm, the procedure does not start with empty S and W, but with some initial values S_0 and W_0, which reflect the current knowledge about the DFCA model.

In case S_0 and W_0 are carefully chosen by a human with a good insight in the model, the constructed $M(S, W, T)$ will be close to the correct DFCA and so the "while"-loop will be executed fewer times or not at all, saving in this way computational resources. At limit, when either S_0 or W_0 are correctly chosen from the outset, the constructed $M(S, W, T)$ will be correct and the "while"-loop will not be executed at all. (In Appendix B we show that a proper state cover of the Event-B model is such a "correctly chosen" S_0 and a strong characterization set is a "correctly chosen" W_0.)

Otherwise, whenever the DFCA is found to be inaccurate, a counterexample (i.e. a sequence s with $\|s\| \leq \ell$ such that $s \in U$ but s is not accepted by the DFCA or vice versa) must be found and the observation table should be extended accordingly. Practical modalities for finding such a counterexample will be discussed in the next subsection.

Therefore, two main cases can be distinguished:

- **Case 1:** The procedure is executed for the first time. In this case, the initial sets S_0 and W_0 are based on an initial estimation of the states of the model. In the worst case (when no initial estimation is available), we take $S_0 = \{\epsilon\}$, $W_0 = \{\epsilon\} \cup A$. (As it emerged from our empirical evaluation (Section 4), in many cases states can be distinguished by singleton sequences, and so initially we consider W to contain all event names, i.e. A).
- **Case 2:** The procedure has already been applied at least once and, consequently, a DFCA model exists. Suppose that this model needs to be improved for a number of reasons:
  - **Subcase 1:** the Event-B model has been modified or augmented due to changes in the requirements.
  - **Subcase 2:** the Event-B model has not been changed but the associated DFCA is deemed to be insufficient for testing purposes. In this case, the upper bound ℓ is increased according to the existing testing needs and the procedure is executed once more for the new value of ℓ.
  - **Subcase 3:** the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). This subcase will be discussed later in Subsection 3.7.

In this (second) case, S_0 and W_0 are constructed by reusing the values of S and W from the previous iteration. In fact, it is not necessary to reuse the entire sets S and W. As shown in Appendix A, it is sufficient to extract from them two minimal subsets $S_{min} \subseteq S$ and $W_{min} \subseteq W$, where S_{min} is the set of all minimum, mutually dissimilar sequences from S, and W_{min} is the set of minimum sequences from W which distinguish between any two dissimilar sequences of S (the formal definitions are given in Appendix A) - this corresponds to the minimization step in Fig. 4.[7] The construction of S_{min} and W_{min} is not computationally expensive; both these subsets are selected by simply scanning the observation table, so the complexity is linear in its size. Additionally, S_{min} is actually the state set of $M(S, W, T)$, so it is computed by the algorithm anyway. In Subcase 1, the initial values for S and W are precisely those from the previous iteration, i.e. $S_0 = S_{min}$, $W_0 = W_{min}$, whereas in Subcase 2, the alphabet A is added to W, i.e. $S_0 = S_{min}$, $W_0 = W_{min} \cup A$ (this heuristic is consistent with Case 1 and is validated

[7]Intuitively, this is because S_{min} and W_{min} still remain a proper state cover and a strong characterization set of $M(S, W, T)$, so the language equivalence (modulo the upper bound ℓ) against any other automaton with the same number of states is ensured.

by experimental evaluation). The observation table T (corresponding to the minimized S and W) is also partially/totally re-constructed in the next iteration as follows. In the first subcase, since the Event-B model has been changed, the value of T must be re-checked for all sequences in $(S \cup SA)W \cap A[\ell]$. In the second subcase, only the sequences in $(S \cup SA)W$ whose length is greater than the previous ℓ need to be re-processed.

Note that (Case 2 / Subcase 1) also covers the situation in which the procedure has not been applied before but an automaton model of the system exists from other sources (e.g. has been developed during the design phase), but has become obsolete. In this case, S_{min} and W_{min} can also be derived from the existing model, as explained above, so the information contained in the existing model is reused in the construction of the new model.

Example. We illustrate the iterative process of constructing the DFCAs with a system for controlling the cars on a narrow bridge between an island and the mainland. This example is particularly relevant since it is used to introduce the main Event-B concepts in Abrial's textbook [11].

The modeled system is equipped with two traffic lights with two colors: green and red. The traffic lights control the entrance to the bridge at both ends. Cars are not supposed to pass on a red traffic light, only on a green one. The system has two main additional constraints: the number of cars on the bridge and island is limited and the bridge is one-way.

We present here only the first two levels of refinement (see Fig. 5). The first model M_0 is very simple. The events ML_out and ML_in correspond to cars entering and leaving the island-bridge compound, respectively. The context contains a single constant d, which is a natural number denoting the maximum number of cars allowed to be on the island-bridge compound at the same time. In our example, we fix $d = 2$ for all the refinement levels of the model. The single variable n of the machine M_0 denotes the actual number of cars.

In the first refinement, the machine M_1 introduces the bridge. The events ML_out and ML_in correspond now to cars leaving the mainland and entering the bridge or leaving the bridge and entering the mainland, respectively. In addition, the events IL_in and IL_out correspond to cars entering and leaving the island, respectively. The variable *n is now replaced* by three variables: a (the number of cars on the bridge and going to the island), b (the number of cars on the island) and c (the number of cars on the bridge and going to the mainland).

Finally, the second refinement introduces the two traffic lights, named ml_tl and il_tl. The model M_2 has two new events to turn the value of the traffic lights color to green when they are red: ML_tl_green and IL_tl_green. In order to avoid the situation when light colors are changing so rapidly that the drivers can never pass, we force the lights to change only when a car has passed in the other direction. For this, two more variables ml_pass and il_pass are introduced: $ml\_pass = TRUE$ means that at least one car has passed the bridge going to the island since the mainland traffic light last turned green; similarly for $il\_pass = TRUE$.

First, we start the learning process with $\ell = 3$ (*Case 1*); for each of the three models, the procedure is executed with initial values $S_0 = \{\epsilon\}$ and W_0 equal to the corresponding input alphabet plus the empty sequence; the resulted DFCAs (plotted using our plug-in) are presented on the right hand side of Fig. 5 (for simplicity, the sink states are not shown).

Suppose now that we want to improve the DFCA for M_2 by increasing the upper bound ℓ (*Case 2 / Subcase 2*). For $\ell = 6$, we obtain the DFCA in Fig. 6(a), which has more states and transitions (and covers events like ML_in that were not covered for $\ell = 3$).

In order to illustrate the iterative DFCA construction (cf. Fig. 4), we provide a counterexample path for the current DFCA associated to M_2. For instance, the following sequence of length 6:

$$w = ML\_tl\_green, ML\_out1, ML\_out2, IL\_in, IL\_in, IL\_in$$

is not feasible in M_2, but it is accepted by the DFCA in Fig. 6(a). The new DFCA taking into account the counterexample w (see the while-loop of Fig. 4) is presented in Fig. 6(b). It can be observed that w is no longer accepted by the new DFCA. The subsection below shows how to deal with the problem of finding counterexamples.

3.3. Finding counterexamples

Naturally, finding a counterexample is the most difficult part of our approach and we propose three solutions:

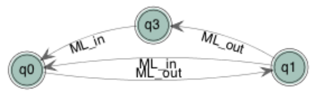
- interactively, using the experience of the human testers who have a good understanding of the model: Testers can use the simulation and animation capabilities of ProB to discover counterexamples, that are fed to the learning algorithm. Moreover, high-priority scenarios that the testers deem as important can be introduced into the learning loop and the associated tests will be covered by the DFCA.
- by testing language equivalence, using the W-method for bounded sequences outlined in Section 2.3: Recall that, given a DFCA model M of a system and an upper bound ℓ, this method generates a test set to check if the implementation under test, modeled by an unknown automaton I, behaves as defined by M for all sequences of length at most ℓ. The test set has the form $X_k = SA[k+1]W \cap A[\ell]$, where S and W are a proper state cover and a strong characterization set of M, respectively, and k is the difference between

```
Machine $M_0$:
  Variables:  $n$
  Event INITIALISATION $\widehat{=}$ begin $n := 0$ end
  Event $ML\_out$ $\widehat{=}$ when $n < d$ then $n := n+1$ end
  Event $ML\_in$ $\widehat{=}$ when $n > 0$ then $n := n-1$ end
```

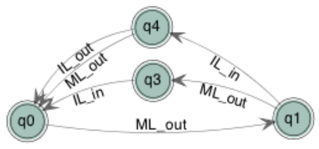
```
Machine $M_1$ refines $M_0$ :
  Variables:  $a, b, c$
  Event INITIALISATION $\widehat{=}$ begin $a, b, c := 0, 0, 0$ end
  Event $ML\_out$ refines $ML\_out$ $\widehat{=}$ when $a + b < d \wedge c = 0$ then $a := a + 1$ end
  Event $ML\_in$ refines $ML\_in$ $\widehat{=}$ when $c > 0$ then $c := c - 1$ end
  Event $IL\_in$ $\widehat{=}$ when $a > 0$ then $a, b := a - 1, b + 1$ end
  Event $IL\_out$ $\widehat{=}$ when $0 < b \wedge a = 0$ then $b, c := b - 1, c + 1$ end
```

```
Machine $M_2$ refines $M_1$ :
  Variables:  $a, b, c, ml\_tl, il\_tl, il\_pass, ml\_pass$
  Event INITIALISATION $\widehat{=}$ begin $(a, b, c := 0, 0, 0)$,
                                     $(ml\_tl, il\_tl := red, red)$,
                                     $(il\_pass, ml\_pass := 1, 1)$
                               end
  Event $ML\_out1$ refines $ML\_out$ $\widehat{=}$ when $ml\_tl = green \wedge a + b + 1 < d$
                                     then $a, ml\_pass := a + 1, 1$ end
  Event $ML\_out2$ refines $ML\_out$ $\widehat{=}$ when $ml\_tl = green \wedge a + b + 1 = d$
                                     then $(a, ml\_pass := a + 1, 1), ml\_tl := red$ end
  Event $IL\_out1$ refines $IL\_out$ $\widehat{=}$ when $il\_tl = green \wedge b > 1$
                                     then $(b, c := b - 1, c + 1), il\_pass := 1$ end
  Event $IL\_out2$ refines $IL\_out$ $\widehat{=}$ when $il\_tl = green \wedge b = 1$
                                     then $(b, c := b - 1, c + 1), (il\_tl, il\_pass := red, 1)$
                                     end
  Event $ML\_tl\_green$ $\widehat{=}$ when $ml\_tl = red \wedge a + b < d \wedge c = 0 \wedge il\_pass = 1$
                         then $(ml\_tl, il\_tl := green, red), ml\_pass := 0$ end
  Event $IL\_tl\_green$ $\widehat{=}$ when $il\_tl = red \wedge 0 < b \wedge a = 0 \wedge ml\_pass = 1$
                         then $(ml\_tl, il\_tl := red, green), il\_pass := 0$ end
  Event $IL\_in$ refines $IL\_in$ $\widehat{=}$ when $a > 0$ then $a, b := a - 1, b + 1$ end
  Event $ML\_in$ refines $ML\_in$ $\widehat{=}$ when $c > 0$ then $c := c - 1$ end
```

M_0: the DFCA for $\ell = 3$



M_1: the DFCA for $\ell = 3$



M_2: the DFCA for $\ell = 3$

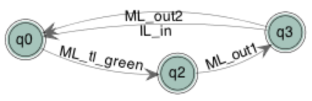
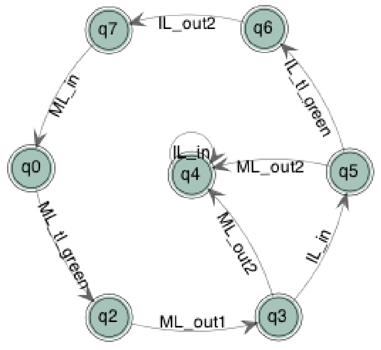
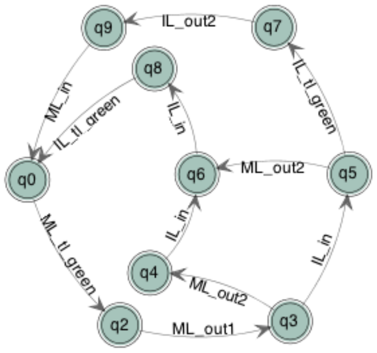


FIGURE 5. The first two refinements of the "Cars on the bridge" example (from Abrial [11]) together with their corresponding learned DFCAs for $\ell = 3$.



(a) DFCA for M_2 and $\ell = 6$



(b) DFCA for M_2 and $\ell = 6$ learning from a given counterexample

FIGURE 6. DFCAs for $\ell = 6$

the estimated maximum number of states of I and the number of states of M. In our case, the model M corresponds to the current DFCA $M(S, W, T)$ and the implementation under test to the Event-B model (more precisely, an approximation of the Event-B model, which contains all set of executable paths of length up to ℓ). Now, the (minimized) sets S and W in the observation table satisfy the definitions of a proper state cover and a strong characterization set of $M(S, W, T)$, respectively. Thus, for $k = 0$, the test set X_0 is actually the set of sequences in the (minimized) observation table, so, if the Event-B model is known to have no more states than the current DFCA, this step is already completed. Otherwise, testing the behavioral equivalence between the current DFCA and the Event-B model corresponds to gradually increasing k until a counterexample is found (a test case produces a different result on the Event-B model compared to the DFCA) or we are satisfied that the DFCA is correct. Note that the size of the test set is exponential in k and so using the W-method for a large k may be expensive.

- by encoding the language equivalence problem into the ProB model checker: For instance the complement of the DFCA is encoded into a CSP process P and ProB will try to run the Event-B machine and P in parallel to find a path that is accepted by Event-B but not by the DFCA. Note, however, that this procedure might be computationally expensive.

From the three options above, in the current version of our implementation we have considered the first two: using the W-method and manually providing the counterexample. As the set $X_0 = SA[1]W \cap A[\ell]$ is already in the observation table, the remaining test set for a given $1 \leq k < \ell$ is $(SA^2W \cup \ldots \cup SA^{k+1}W) \cap A[\ell]$. In order to avoid a blow up in size, the W-method is combined with some randomness as follows: the algorithm generates all sequences in SA^2W and at most r sequences randomly chosen from the union of the sets SA^jW, $2 < j < \ell$, such that the random sequences are equally distributed over the subdomains SA^jW. Since S and W have at most n and $n-1$ elements, respectively, the resulting test set will have at most $p^2n(n-1)+r$ sequences, where n denotes the number of states of the minimal DFCA and p the size of the A. In our experiments we used this option, in which the value of constant r was 3000. Note also that once a counterexample is found, we stop the search and apply the learning procedure.

On the other hand, our tool also allows counterexamples to be manually provided by the user, so the testers can use their intuition to provide relevant sequences to the algorithm to learn and thus more directly influence the result of the test suite. This is in contrast to purely automatic test generation techniques that are driven solely by coverage criteria, where the produced tests may not be intuitive or may not cover existing standard testing scenarios in the domain. This is in the spirit of the original Angluin's algorithm and, in fact, fits well with common practice, in which human knowledge is used to guide model and test design (cf. Fig. 6). For that, as mentioned before, the user can experiment with different paths through ProB simulation. Moreover, ProB model-checking capabilities can be used to search for paths that satisfy certain properties (e.g. an execution containing a certain event or succession of events) - such paths can be identified using the metrics defined in the next section.

3.4. Two metrics on the learned models

Finally, an important issue is how we asses the "quality" of the obtained models and, consequently, when we decide that the upper bound ℓ is sufficiently large for our (testing) purposes. One natural way to measure how well the DFCA produced by this approach approximates the actual system model is by using the notion of *coverage*. Coverage is a widespread means of measuring the quality of a test suite.

In (black box) testing from a finite state machine, two wide spread criteria measure the percentage of edges or pairs of edges covered by the test suite [25]. In particular, event pair coverage (also called switch cover, transition-pair or two-trip) is considered a powerful test coverage criterion and is included in the British Computer Society standard for software component testing [26]. Also note that, since an input symbol in the automaton normally corresponds to a program statement, these two criteria correspond to statement (node) coverage and branch coverage, respectively, in white box testing. These are two of the most widely used criteria in white box testing and, in particular, branch coverage has been proven to produce effective test suites [26].

We have adapted these criteria to measure the quality of the obtained DFCAs as follows: we have counted the number of sequences of events of length i, $1 \leq i \leq 2$, which can be executed by some state of the DFCA; as the DFCA may also under approximate the exact DFA model, we have also counted the number of sequences that are not executable from some state of the DFCA. The resulting metrics are expressed as percentages of the total number of cases ($2 \cdot card(A)$ for $i = 1$ and $2 \cdot card(A)^2$ for $i = 2$). Note that the absolute value of the metrics depends on the domain of the model. However, we are not interested in the absolute value, but we consider the *stabilisation of the metrics* as an indication that the produced DFCA preserved the properties of the original Event-B model (in this case, the sequences of events up to length 2 that can and cannot be executed from each of its states).

During the experiments (see Section 4), we noticed that the chosen metrics performed well in the sense that for many examples, after two steps with no change in the metrics, the values did not modify in the following steps, i.e. they stabilised. So, the stabilisation of metrics is used as a "rule of thumb" for stoping incrementing ℓ. Note that the size of the models may continue to grow as we increase ℓ and search further for counterexamples, but we can decide not to spend the effort of further computations once the coverage as reflected by the metrics is not improved. Even more importantly, as the total length of the resulting test suite is proportional to the third power of the size (number of states) of the corresponding automaton (see Section 3.6), the use of these metrics can drastically reduce the effort involved in the test application process.

Another advantage of the metrics is that they can reveal events or pairs of events that are not covered by the learned automaton. Once they stabilise, we check the uncovered elements and use the ProB model-checker to search for sequences that cover them. Thus, the computationally expensive model-checking procedure is used sparsely only on such concrete problems.

3.5. Test data generation

In order to decide whether a given sequence s, $\|s\| \leq \ell$, is accepted or not by the DFCA M (i.e., $s \in L_M$ or not), the procedure needs to check if s is a feasible path through the Event-B model. This is achieved by effectively constructing (or attempting to construct) test data to drive the given path. If the appropriate test data has been found, then $s \in L_M$; otherwise, the path is declared infeasible[8] and so $s \notin L_M$. Therefore, deciding whether $s \in L_M$ or not reduces to finding test data to execute the corresponding path of the Event-B model.

Then all that remains is to specify the method(s) used to find test data to execute a given path of an Event-B model. So far two such approaches have been proposed and implemented. The first used symbolic execution and reduces this problem to solving a set of constraints [20]. The second reduces the problem to an optimization problem, which is then solved using search-based techniques (genetic algorithms) [12]. Note that the test data generation problem may be complex even for one path, when the guards are complex and the test data domain are large. In particular, the set-theoretic nature of Event-B increases the search space because free set variables v that are subsets of a given carrier set V (i.e. $v \subseteq V$) can take exponentially many values, $2^{card(V)}$. Consequently, most of the time taken by the execution of the procedure is spent on generating the actual test data.

3.6. Test suite construction

When the process of constructing the DFCA is completed, a test suite for the Event-B model has also been obtained; this is precisely the set of sequences in the observation table returned by the procedure, $X_0 = (S \cup SA)W \cap A[\ell]$ (note that the procedure returns the minimized S and W so these are the used in the definition of X_0). The test sequences can be classified into *positive* (for which $T(x) = 1$, which correspond to feasible paths in the Event-B model) and *negative* (for which $T(x) = 0$). Naturally, test data can only be generated for feasible paths and, as explained earlier, test data generation is implicitly included in the DFCA construction procedure. Negative test sequences are also useful for testing the system implementation since they describe erroneous scenarios, which the system cannot perform in normal functioning.

Following [17], the constructed set will constitute a conformance test suite for the Event-B model modulo the bound ℓ (the ℓ-bounded behavior of the model). Such a test is more powerful than a set of tests based on state or transition coverage criteria since it covers all states and all transitions of the equivalent automaton and also checks each state and the initial and destination states of each transition. Conformance testing is especially relevant in the embedded systems domain.

Regarding the size of the produced test suite, the maximum number of test cases (test sequences) is pn^2, where $p = card(A)$ and n is the number of states of the learned automaton [19]. Moreover, the total length of the test suite, i.e. the sum of lengths of all test cases, does not exceed pn^3 [19]. Therefore, the effort involved in the test application process is proportional to pn^3, and so any reduction in the size of the automaton (due to the use of the L^ℓ algorithm instead of L^*) will be translated accordingly into a reduction in the complexity of this process.

Increasing ℓ, longer and more complex tests are generated. However, very complex or long test sequences are usually not the norm, so having the ability to tune the length of the test case using ℓ, guided by the metrics defined in Section 3.4, is an advantage of our approach. On the other hand, if the obtained (conformance) test suite is still too large for the intended purpose, it can be reduced by choosing a smaller subset that satisfies simpler coverage criteria, like transition, state or event coverage. Regarding coverage criteria, if a very specific coverage criteria is sought from the outset (cf. [27]), our method can accommodate this to some extent in that the training set of sequences for the learning algorithm can be chosen according to the given coverage.

3.7. Relation to Event-B refinement

Very often, model design is an iterative process, in which, at each step, the existing (more abstract) model is replaced by a more concretized model through *refinement*. The incremental approach of L^ℓ allows us to reuse the learned model and test suite of the abstract model to the next more concretized model (Case 2, Subcase 3 from Subsection 3.2), as explained below. Again, the approach is presented in the context of Event-B, but the basic ideas can be extended to other languages which provide model refinement as a way to handle complexity.

Suppose we have a refinement from *AM* (abstract model) to *CM* (concrete model). In a refinement step, new events can be introduced and the existing events can be detailed. Let A and A' denote the sets of events of *AM* and *CM*, respectively, and let $E \subseteq A'$

[8]Note that here *infeasible* means only that our tools could not find test data within reasonable time (e.g. 20 seconds for one path) and we stop searching, whereas in reality there might exists such test data. However, since we are working with approximated models, this incompleteness aspect is not very important.

```
Transformation of $S$

Input: $S_{min}$ and the restriction of $T$ to $S_{min}$
$maps_S(\epsilon) = \epsilon$
$Y = S_{min} \setminus \{\epsilon\}$
While $Y \neq \emptyset$ do
  Select $x = sa$, $s \in A^*, a \in A$,
      where $x$ is a sequence in $Y$ of minimum length
  $s' = maps_S(s)$
  If $T(x) = 1$ then
    $found = find\_next(s', a, t)$
    If $found$ then
      $maps_S = maps_S \oplus (x, s't)$
      $Y = Y \setminus \{x\}$
    Else
      If $s = \epsilon$ then
        Return failure
      Else
        $Y = Y \cup (dom(maps_S) \cap \{s\}A[1])$
        $dom(maps_S) = dom(maps_S) \setminus \{s\}A[1]$
  Else
    $maps_S = maps_S \oplus (x, s'a')$ for some $a' \in ref(a)$
    $Y = Y \setminus \{x\}$
Return $S^R_{min} = pref(Im(maps_S))$
```

FIGURE 7. The transformation of S in the case of refinement

be the new events introduced in *CM* that do not refine any abstract event. Every abstract event $a \in A$ from *AM* will correspond to a set (containing one or many concrete events) from *CM*.[9] Let us denote this set $ref(a)$, $ref(a) \subseteq A' \setminus E$. Suppose S_{min} and W_{min} are the (minimized) sets produced by the application of our procedure on the abstract model *AM*. As these sets contain sequences of abstract events, they need to be transformed before they can be reused in the construction of the automaton corresponding to the concrete model *CM*. On the other hand, in Appendix B it is shown that only one of the two sets (S_{min} or W_{min}) is sufficient to correctly determine the corresponding DFCA model (the other set is reconstructed by the algorithm). Furthermore, the set of all feasible paths of an Event-B model is closed under prefixing and so, naturally, a path from *AM* is transformed into a path from *CM* by gradually transforming its prefixes. Such a transformation is natural in the case of S, which is prefix-closed, but is problematic for W, which must be suffix-closed. For these reasons we choose to only transform the set S_{min}. For W, we will use the same type of heuristic as for the case in which the DFCA construction procedure is executed for the first time: W is initialized with the set A' of all events of *CM* (along with the empty sequence).

The transformation of S_{min} is given in pseudocode in Fig. 7; $maps_S$ denotes the mapping between each sequence in S_{min} and the corresponding sequence in the concrete model. Ultimately, the algorithm will return the prefix-closure of the image of $maps_S$, $pref(Im(maps_S))$. Y denotes the set of sequences from S_{min} that remain to be processed. Sequences are processed in increasing order of their length, so, at any time, a sequence of minimum length from Y is selected. As S_{min} is prefix-closed, $maps_S(x)$ is obtained by extending the transformation of its longest prefix s ($x = sa$, $s \in A^*, a \in A$). Two main cases can be distinguished.

- x is a feasible path of *AM*. Assume that $maps_S(x)$ is also a feasible path of *CM* (this assumption is justified and its implications are discussed later). Then $maps_S(x)$ is obtained by extending $s' = maps_S(s)$ with a sequence $t = e_1 \dots e_j a'$, where $e_1, \dots, e_j \in E$, $j \leq k$ (k is a predefined upper bound), and $a' \in ref(a)$. The function *find_next*(s', a, t) searches for such a t in increasing order of j; if found, the function will return *TRUE*, otherwise *FALSE*. *find_next* may be called several times with the same input parameters s and a' during the execution of the algorithm. Each time, it continues the search from where it left off, so each time a different solution is produced. Consequently, *find_next* maintains an internal state, which, for simplicity, is not given in the code. If *find_next* cannot find a (new) solution, the algorithm backtracks: it removes $s \neq \epsilon$ and all sequences which extend s from the domain of $maps_S$ and adds them to Y. Consequently, the algorithm will resume by processing s. If $s = \epsilon$, the algorithm cannot backtrack any further; in this case, it stops and reports failure.
- x is not feasible in *AM*. Then s' can be extended with any $a' \in ref(a)$. [10] Note that S_{min} contains only one such sequence since the corresponding automaton will have only one non-final ("sink") state.

In refinement, an event a from *AM* is replaced by (one or more) events $ref(a)$ in *CM*, describing the system reactions in different circumstances. Furthermore, the applications of the extra events may also condition the event operation and so each application of a in *AM* is replaced by some sequence $e_1 \dots e_j a'$ in *CM*, with $e_1, \dots, e_j \in E$ and $a' \in ref(a)$.

The new events from E cannot be indefinitely enabled [11] and, furthermore, in practice it is reasonable to expect that an upper bound k on the number of times they can be applied in the absence of an event from $A' \setminus E$ can be established, and so $j \leq k$; the upper bound k is then used in the definition of the *find_next*

[9] In general the correspondence may be many-to-many but in the vast majority of the models we have encountered a one-to-many correspondence is sufficient.

[10] Since the concrete guard is not weaker than the abstract one [11], a infeasible path in *AM* can only give rise to infeasible paths in *CM*.

function presented above. Thus, any feasible path $a_1 \dots a_n$ in the abstract model can be mapped (not necessarily in an unique fashion) onto a feasible path $u_1 a'_1 u_2 a'_2 \dots u_n a'_n$ in the concrete model, with $a'_i \in ref(a_i)$ and $u_i \in E[k]$, $1 \leq i \leq n$. This ensures that the transformation procedure will end successfully, so every sequence in S_{min} is refined appropriately. Finally, we need to ensure that S^R_{min} is prefix-closed and so we take the prefix-closure of the refined sequences. Note that the transformation procedure given in Fig. 7 is only guaranteed to terminate successfully if every feasible path in *AM* has a corresponding feasible path in *CM*. This condition is in the spirit of refinement and is satisfied by the vast majority of the applications we have encountered. However, the algorithm can be easily extended so that it terminates successfully even when not all sequences in S_{min} can be refined - for simplicity and due to its reduced practical value, this idea is not pursued here.

Once the set S_{min} has been transformed, the DFCA construction procedure can be executed for the concrete model *CM* with initial values $S_0 = S^R_{min}$ and $W_0 = \{\epsilon\} \cup A'$. The upper bound ℓ^R used for the concrete model *CM* also needs to be established. This will be set by the user, but, naturally, it will be greater than or equal to the length of the longest sequence in S^R_{min} plus one.

The DFCA construction procedure can always be applied directly on the concrete model, but the strategy presented here, which reuses the information from the abstract model in the construction of the concrete model and of its associated test set, presents some advantages, by offering a way of incorporating the human knowledge into the model at the appropriate level of abstraction. Let us consider two Event-B models, *AM* (abstract) and *CM* (concrete), as above. It is likely that the first DFCA for *AM* is not satisfactory, so a richer automaton is obtained by supplying the construction procedure with appropriate counterexamples. In the "reuse" strategy, these counterexamples are propagated to the next level - they are implicitly included in the DFCA for *CM*. On the other hand, when the DFCA for *CM* is produced from scratch, all counterexamples must be produced at this level. Naturally, abstract models are simpler than concrete models, so finding counterexamples (by human intervention or automatically) for the abstract model is simpler. Moreover, if the tester wants to have certain scenarios into the test suite (see end of previous subsection), it is more convenient to do this at an abstract level and have it propagated automatically by the reuse algorithm for refinement.

We also observed on our benchmark that if we do not perform the computationally expensive automatic search for counterexamples from Subsection 3.3, the "reuse" strategy would produce in many cases richer DFCAs than those produced "from scratch" (we do not give the experimental data due to space constraints).

$$\begin{tikzcd}
 & & RZ_1 \arrow[r, "\text{refinement}"] & RRZ_1 \\
Z \arrow[r, "\text{refinement}"] & RZ \arrow[ur] \arrow[dr] \arrow[r, phantom, "\text{decomposition}"] & & \\
 & & RZ_2 \arrow[r] & RRZ_2
\end{tikzcd}$$

FIGURE 8. An example of decomposed model

Last but not least, even if theoretically the algorithm in Fig. 7 seems computationally expensive, in our experiments the transformation times are very reasonable and contributed much less than the learning steps (and also the counterexample search step). This can be seen in Table 4 provided at the end of Appendix C, which presents transformation times and learning times (given in seconds) for a couple of machines and values of the bound ℓ. In most cases, transformation times are much smaller (for the larger models less than 10% and usually less than 5%) than learning times. Note that the expensive counterexample search is not included in the displayed learning time.

3.8. Relation to Event-B decomposition

Fig. 8 describes a typical incremental development in Event-B involving decomposition (of type either shared-events or shared-variables). There, RZ, which is a refinement of Z, is decomposed into RZ_1 and RZ_2, which are further refined into RRZ_1 and RRZ_2, respectively. For this example, our approach will first construct a DFCA model for Z, which will be reused in the construction of a DFCA for RZ as in the previous subsection. RZ will constitute the basis for the construction of the DFCAs for RZ_1 and RZ_2 starting the learning procedure with the projections of the observation tables. The DFCAs for RZ_1 and RZ_2 will, in turn, be reused in the construction of the final models, for RRZ_1 and RRZ_2. If required, these latter models are used to produce a DFCA model and tests for the overall system using certain composition operators. Details and experiments are provided in our paper [15] and are not reproduced here.

4. EXPERIMENTAL RESULTS

In this section we provide experimentation results on a comprehensive benchmark of Event-B models, in order to test the feasibility of our approach. The implementation of our algorithms was done in Java as an Eclipse plugin[11] to the Rodin platform (in its latest version 2.7) and it was evaluated together with the industrial partners in the DEPLOY project. The membership queries were implemented using the constraint-solving functionality of ProB and a timeout

[11]Our tool is freely available. Installation instructions and screenshots can be found at: `http://wiki.event-b.org/index.php/MBT_plugin`

TABLE 2. The complexity dimensions of the ten subjects (number of refinements, number of events and number of variables)

Subject	**No. of refin's**	**No. of events per refin. (M0/M1/...)**	**No. of variables per refin. (M0/M1/...)**
A2A	12	3/4/4/6/8/8/11/13/14/15/15/16/16	2/2/4/8/8/10/11/12/13/15/16/18/17
BepiColombo	3	5/10/12/16	6/10/12/18
CarsOnBridge	3	2/4/8/16	1/3/7/18
Choreography	1	6/12	7/17
MobileAgent	5	4/6/6/7/7/7	3/5/5/7/7/7
PressCtrller	7	4/12/16/16/20/20/20/28	2/6/8/8/10/10/11/15
ResponseCoP	3	4/13/16/16	3/4/5/6
SSFPilot	3	14/20/23/41	5/7/8/14
TrainCtrller	8	7/9/14/19/19/26/37/42/42	5/6/9/14/14/15/27/33/35

TABLE 3. The relation of learning and metrics for four Event-B machines

MobileAgent (M3 / 7 ev.)					**PressCtrller (M2 / 16 ev.)**					**CarsOnBridge (M3 / 16 ev.)**					**SSFPilot (M1 / 20 ev.)**				
ℓ	$\lvert Q\rvert$	**metr1**	**metr2**	**time**	ℓ	$\lvert Q\rvert$	**metr1**	**metr2**	**time**	ℓ	$\lvert Q\rvert$	**metr1**	**metr2**	**time**	ℓ	$\lvert Q\rvert$	**metr1**	**metr2**	**time**
3	8	86%	39%	1.04	3	12	69%	22%	3.02	3	4	19%	1%	3.18	3	5	25%	12%	6.86
4	11	86%	53%	3.46	5	45	94%	49%	18.94	9	22	63%	17%	24.43	5	15	80%	31%	23.29
6	25	100%	76%	8.20	6	72	100%	63%	50.49	13	52	88%	27%	110.20	6	30	100%	41%	3.12
7	30	100%	82%	11.10	9	170	100%	79%	224.79	15	65	94%	29%	131.42	7	59	100%	53%	25.24
8	40	100%	88%	14.84	10	201	100%	82%	224.89	17	70	100%	30%	164.79	15	315	100%	67%	332.32
9	47	100%	88%	32.99	11	225	100%	82%	250.31	23	98	100%	39%	158.99	16	327	100%	67%	277.65
20	207	100%	88%	190.44	15	257	100%	82%	100.61	25	106	100%	39%	284.86	19	360	100%	67%	4479.95
34	394	100%	88%	4912.29	40	257	100%	82%	14.06	40	106	100%	39%	13.14	23	437	100%	67%	6766.67

of 20 seconds per query was imposed. We run ProB with fixed internal parameters, although fine-tuning of ProB parameters may improve results in certain cases. The experiments were conducted on a Windows 7 Professional 64-bit machine with an Intel Core i7 2.80GHz (8 CPUs) processor and 12 GB of RAM.

We used a broad range of models for experimentation, including systems from the embedded systems, transportation and aerospace industries as well as academic and pedagogical Event-B models used in the literature (see Appendix C for short individual descriptions). All the chosen nine Event-B models are publicly available in the model repository of the DEPLOY project[12]. Table 2 presents the models together with their complexity, i.e. number of refinements, number of events for each refinement and number of variables for each refinement. For instance, the third model is our running example "CarsOnBridge" (see Fig. 5). It has 3 refinements (M0/M1/M2/M3); each level (including the initial machine M0) has 3, 5, 9, and 17 events, respectively; and 1, 3, 7, and 18 variables[13], respectively. Note also that many of the models are rather complex, e.g. *TrainCtrller* exhibits 8 levels of refinements and the last level has 43 events and 35 variables.

Table 3 presents how the learning algorithm and the two metrics from Subsection 3.4 performed on four examples from our benchmark[14]. The other five models behave similarly. For each of the four examples, we specify which machine was considered (e.g. M3) and how many events that machines contained (e.g. 7 events). Then, we provide five columns as follows. Let us consider the first model, *MobileAgent*. First column gives eight increasing values of the bound ℓ, from a small value ($\ell = 3$) to a large one ($\ell = 34$). Second column shows the size of the learned DFCA ($|Q|$) for the given ℓ. We see that for a large ℓ, this size can grow to hundreds of states (394 states for $\ell = 34$). Regarding the learning procedure, the DFCAs are obtained by reusing the observation table from previous ℓ and by using automated search for counterexamples from Subsection 3.3. The $|Q|$ given in the table is the one obtained when no more counterexamples were found (there could be zero, one or several counterexamples found in one step). Third and fourth columns present the coverage percentage provided by the two metrics of Subsection 3.4. For instance, for $\ell = 4$, the first metric (metr1) is 86% and the second one (metr2) 53%. The (rounded) percentage of 86% for first metric means that 12 cases out of 14 (2 times 7, the no. of events) were covered by the DFCA with 11 states. Finally, the fifth column logs the time in seconds for computing the DFCA, by adding up the times for learning and counterexample

[12] `http://deploy-eprints.ecs.soton.ac.uk`

[13] Variables might have an integer type, but also more complex types like sets, relations or partial functions, which increase the complexity of the algorithms, especially the membership queries.

[14] Since the counterexample search algorithm involves randomness, we run the experiments several times. Table 3 shows the results for one run, namely a representative one for the average behaviour.

search for a given ℓ (without summing up the times for smaller ℓ's).

Looking at the *MobileAgent* example, we see that the metrics stabilise at $\ell = 8$, but as ℓ grows, the number of states $|Q|$ also increases. However, from a testing point of view we could stop at $\ell = 8$ as the achieved coverage is reasonable. The same behavior is observed for *PressCtrller*, where both metrics stabilise at $\ell = 8$. The difference to the previous example is that the size of the automaton also stabilises at $\ell = 15$: at that point $|Q| = 257$ and remains so independent of the increase of ℓ up to 40. This is an indication that the learned automaton is pretty close e.g. to the one given by L^* algorithm which does not consider bounds during learning. The next model, *CarsOnBridge*, behaves similarly to *PressCtrller*: the first metric reaches 100% for $\ell = 17$ and the second metric stabilises at $\ell = 23$. Moreover, for values of ℓ greater than 25, the size of the automaton remains constant at 106 (we run the algorithm until $\ell = 40$ – note also that the computation time for $\ell = 40$ is small; this is because no new states are added, so no learning is needed). As for the last considered example, *SSFPilot*, which is a model from industry, we could stop at $\ell = 15$, when both metrics stabilise, but, due to the large resulting state space, one can also choose to control the increase of ℓ only by the first metric, which expresses a 100% coverage at $\ell = 6$. Thus, the human tester can make informed decisions on when to stop, based on the results of the metrics.

Regarding the generated tests presented in Subsection 3.6, we provide in Table 5 in the Appendix the sizes of the test suites (number of tests) for all the models of the benchmark, but only with selected values of ℓ. Here, we discuss about the sizes of the test suites for the *SSFPilot* model for machine $M1$ and the bounds used in Table 3 (see also the size of the corresponding DFCAs in Table 3). So, we have: for $\ell = 3$, 270 tests; for $\ell = 5$, 1,510 tests; $\ell = 6$, 4,612 tests; $\ell = 7$, 10,371 tests; $\ell = 15$, 139,162 tests; $\ell = 16$, 155,465 tests; $\ell = 19$, 193,829 tests; and $\ell = 23$, 278,218 tests. Notice the difference in size between the test suites for $\ell = 6$ (when the first metric stabilises) and the test suite for $\ell = 23$. For $\ell = 15$ (when the second metric stabilises), the number of produced tests may still seem high, but this is because conformance testing is a strong form of testing, heavily exercising the system. However, more compact test suites can be obtained according to weaker coverage criteria. For instance, test suites for state and transition coverage are readily available from the learning procedure, from the (minimized) sets S and $S \cup SA$, respectively. Moreover, we have also implemented different test suite optimization algorithms that produce significantly smaller test suites. As the quality of the test suite depends on the model, it is better to base the test generation on a richer model and then optimise the test suite afterwards, if needed. Finally, regarding the quality of the produced tests, since the implementations of the Event-B models were not available, fault-seeding was not considered relevant because this technique is aimed at uncovering faults in the actual system, not the model. However, we used in our evaluation coverage based criteria, widely accepted and considered to be closely associated with fault detection, as previously discussed.

5. RELATED WORK

The correspondence between conformance testing and automata learning is discussed, from a theoretical point of view, by Berg et al. [28], who show how results from one area can be transferred to the other. Such a correspondence is also exploited in our paper; in our case, however, the correspondence is between conformance testing for bounded sequences and cover automata learning. In this bounded case, the algorithm produces tests of minimal length, unlike the tests produced using the original L^* method. Furthermore, here we propose an incremental learning and test generation approach, in which the results obtained for a previous model are reused in the next iteration.

An adaptive approach to model development is proposed by Groce et al. [29], but the emphasis there is on model checking, rather than test generation, as in our approach. Furthermore, a DFA (not a DFCA) model of the system is built. Our approach can gradually build an appropriate model by suitably setting the value of the upper bound ℓ or through the (Event-B) refinement mechanism.

The use of Angluin-style automata learning techniques for test generation is also discussed by Hagerer et al. [30] and Hungar et al. [31]. As above, both papers refer to the case of unbounded sequences. Furthermore, in both papers the focus is on language learning and test generation is only mentioned as an addendum: once the model is constructed, it can be used as basis for automatic test generation. Hungar et al. present a technique for optimizing complex system learning. Hagerer et al. present a technique, called regular extrapolation, for model generation from knowledge accumulated from different sources and expert knowledge. The final model is only an approximation of the real system and so the test cases derived from it may not attain the desired level of coverage. Interestingly, testing is also used to validate the obtained model and the authors state that "most errors show up in short sequences". This supports the view of our approach, which considers bounded sequences. Alternatively, [32] provides an incremental learning algorithm for Mealy automata, which is used for test generation. However, the learning algorithm is not Angluin-based, but uses a so-called Congruence Generator Extension.

Bounded model checking (based on SAT methods) [9, 10] is also gaining popularity in the formal verification community, the general consensus being that it works particularly well on large designs where bugs need to be searched at shallow to medium depths.

Dupont et al. [33] and Walkinshaw et al. [34] use automata inference techniques to construct behavior models of software systems. However, there are a number of key differences from our approach. First, an initial set of training data is supplied to the algorithm for model construction. In order to construct an accurate model, training sequences that satisfy an appropriate level of coverage must be supplied. Therefore, these approaches rely on the existence of good test sets, rather than assist in producing such sets. In subsequent work, Walkinkshow et al. [35] produce these test sets using model-based testing. Second, the behavior of all (unbounded) sequences is considered. While the model produced by this approach may be exact, it may be too complex and so the whole process may be too expensive to have a practical value. By appropriately setting the upper bound ℓ, our approach has the potential to strike the right balance between accuracy and costs. Naturally, at limit (i.e. for ℓ sufficiently large), our approach will also produce an exact model of the system.

Grieskamp et al. [36] construct finite automata that under-approximate the global state space of an ASM. They use an algorithm that combines several concrete states into abstract ones using logical formulas to distinguish the abstract states. The obtained finite automaton is used for test generation. Note however that such a merging approach may introduce non-determinism, which is usually not desired in test models. Using a similar idea and extra information on logical dependences between Event-B guards, Bendisposto and Leuschel [37] construct an abstract over-approximation of the control flow graph of an Event-B model. Compared to our approach based on incremental learning, both [36] and [37] use a different strategy, relying on state merging and logical formulas for distinguishability.

Cho et al. [38] learn an abstract model, which is then used for vulnerability detection, by abstracting the system inputs and outputs. The output abstraction function is assumed to be known beforehand and provides the basis for merging a cluster of concrete states into an abstract state. On the other hand, the input abstraction is unknown and a subset of concrete inputs are selected by the algorithm and used in the model. The model learning is iterative: after each iteration, dynamic symbolic execution is used to explore the surroundings of each concrete state and select new inputs, which are used in the next iteration. This approach is quite different from ours, in which an abstract (input) alphabet (e.g. the set of events of Event-B specifications) replaces the input and output abstractions. This abstract alphabet is part of the considered specification and, consequently, is known beforehand; this is the case for Event-B models but also fits very well with usual programming paradigms (e.g. procedural, object-oriented), in which the program units (e.g. functions, procedures, methods) are also known. Furthermore, our approach fits well with the usual testing experience, which shows that most faults are detected using short sequences. Note also that, unlike [38], our approach uses functional testing (i.e., the W-method adapted to bounded sequences) to restrict the sampled domain and therefore to provide additional guidance for counterexample generation. On the other hand, the two approaches are orthogonal (one limits the size of the model by using alphabet abstraction, the other by placing a bound on the sequences length) and could even be combined to limit state explosion. There are also a number of approaches which use alphabet refinement in model learning [39, 40]: similarly to [38], the evolution of the learned model is determined by an expansion of an initial abstract alphabet. Finally, the algorithm proposed by Chen et al. [41] solves a similar but more general problem than ours, however at a much increased computational cost (on the particular problem addressed by our method) induced not only by a significantly increased number of language queries, but also by calls to an additional NP-complete minimization procedure.

There is little existing work done for the testing of Event-B models. The interest in testing was spurred by the idea of using the formal Event-B models not only for analysis but also for test generation. Driven by the industrial partner SAP who intends to generate test cases for enterprise applications, an approach using the explicit model checker ProB was devised [14]. However, the classical state space explosion problem hampers the applicability of the method to the larger models, especially for data-intensive models. A recent approach to tackle the test data generation using evolutionary methods was proposed in [12]. However, in there only the test data for one given event sequence is searched for and the method must be complemented by algorithms that generate the whole test suite. Our approach provides a way of generating a test suite and thus complementing [12] and furthermore, the state space explosion of [14] is kept under control by incremental model learning putting a certain bound on the lengths of the test cases. Moreover, [42] presents a method of incrementally concretizing test scenarios using the information from the Event-B refinements. However, the paper only addresses one scenario at a time, it does not discuss the test data generation, and the control flow graphs are constructed manually. Finally, to the best our knowledge, this is the first attempt to use grammatical inference techniques for Event-B models.

6. CONCLUSIONS

In this paper, we presented a novel approach of using model learning for testing purposes and its application to the Event-B method, being successfully implemented for Event-B in a very large European project called DEPLOY. The method is based on sound theoretical automata theory foundations and has an incremental and interactive nature that makes it fit the testing practice requirements. The prototype implementation showed that the method works well on a comprehensive benchmark of realistic models, of medium or even fairly large size. Since the complexity of the large specifications is tackled not only by successive refinements but also by model decompositions, we integrated these features in our learning and test generation method. Moreover, we provide automation of counterexample search, by a combination of black-box and random testing, and metrics to evaluate the quality of the produced results. Based on these metrics, test suites can be generated from the approximate automata models constructed by our algorithm rather than from the exact automaton model. Since the exact automaton can be significantly larger, or even prohibitive to construct, test sets of the same level of coverage may require a substantial increased computational effort or, at worst, may be even infeasible to construct.

As future plans, we intend to further investigate the scalability of our approach and implementation on even larger models together with an industrial partner from the DEPLOY project. We also plan to implement further different optimizations on our prototype, especially on membership queries and counterexample search, which constitute the most expensive parts of the procedure. For instance, we could compute batches of membership queries in parallel (on a multi-core/multi-processor architecture) or evaluate partial-order reductions exploiting the independence of events similar to [31] as well as improve the random part of counterexample search with ideas from [43] and [44]. Moreover, it would be useful to integrate our learning algorithms into existing automata learning frameworks like LearnLib[15] and LibAlf[16] and thus contribute to the community efforts in bringing automata learning forward.

Funding

This work was supported by project DEPLOY, European FP7 grant no. 214158, and by Romanian National Authority for Scientific Research (CNCS-UEFISCDI) grant no. PN-II-ID-PCE-2011-3-0688 (project MuVeT).

REFERENCES

[1] Lee, D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, pp. 1090–1126.

[2] Harel, D. and Politi, M. (1998) *Modeling reactive systems with statecharts: the STATEMATE approach.* McGraw-Hill.

[3] Mouchawrab, S., Briand, L. C., Labiche, Y., and Penta, M. D. (2010) Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Trans. on Software Engineering*, **37**, 161–187.

[4] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A. M., and Ernst, M. D. (2010) Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. on Software Engineering*, **36**, 474–494.

[5] Angluin, D. (1987) Learning regular sets from queries and counterexamples. *Inf. Comput.*, **75**, 87–106.

[6] Ipate, F. (2012) Learning finite cover automata from queries. *Journal of Computer and System Sciences*, **78**, 221–244.

[7] Câmpeanu, C., Sântean, N., and Yu, S. (2001) Minimal cover-automata for finite languages. *Theoret. Comput. Sci.*, **267**, 3–16.

[8] Câmpeanu, C., Paun, A., and Smith, J. R. (2006) Incremental construction of minimal deterministic finite cover automata. *Theoret. Comput. Sci.*, **363**, 135–148.

[9] Prasad, M. R., Biere, A., and Gupta, A. (2005) A survey of recent advances in SAT-based formal verification. *STTT*, **7**, 156–173.

[10] Cordeiro, L., Fischer, B., and Marques-Silva, J. (2012) SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. on Software Engineering*, **38**, 957–974.

[11] Abrial, J.-R. (2010) *Modeling in Event-B - System and Software Engineering.* Cambridge University Press.

[12] Dinca, I., Stefanescu, A., Ipate, F., Lefticaru, R., and Tudose, C. (2011) Test data generation for Event-B models using genetic algorithms. *Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11)*, CCIS, **181**, pp. 76–90. Springer.

[13] Dinca, I., Ipate, F., Mierla, L., and Stefanescu, A. (2012) Learn and test for Event-B – a Rodin plugin. *Proc. of ABZ'12*, LNCS, **7316**, pp. 361–364. Springer.

[14] Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., and Schieferdecker, I. (2009) Applying model checking to generate model-based integration tests from choreography models. *Proc. TESTCOM'09*, LNCS, **5826**, pp. 179–194. Springer.

[15] Dinca, I., Ipate, F., and Stefanescu, A. (2012) Model learning and test generation for Event-B decomposition. *Proc. of ISoLA'12*, LNCS, **7609**, pp. 539–553. Springer. Extended version online at: `http://tinyurl.com/isola12-with-appendix`.

[16] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006) *Introduction to Automata Theory, Languages, and Computation (3rd Ed.).* Addison-Wesley.

[17] Ipate, F. (2010) Bounded sequence testing from deterministic finite state machines. *Theoret. Comput. Sci.*, **411**, 1770–1784.

[18] Vasilevski, M. P. (1973) Failure diagnosis of automata. *Kibernetika*, **4**, 98–108.

[19] Chow, T. S. (1978) Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, **4**, 178–187.

[20] Leuschel, M. and Butler, M. J. (2008) ProB: an automated analysis toolset for the B method. *Int. J. Softw.*

[15] `http://ls5-www.cs.tu-dortmund.de/projects/learnlib`
[16] `http://libalf.informatik.rwth-aachen.de`

Tools Technol. Transf., **10**, 185–203. Tool webpage: `http://www.stups.uni-duesseldorf.de/ProB`.

[21] Hoang, T. S., Iliasov, A., Silva, R., and Wei, W. (2011) A survey on Event-B decomposition. *ECEASST*, **46**, 1–15.

[22] Silva, R., Pascal, C., Hoang, T. S., and Butler, M. (2011) Decomposition tool for Event-B. *Softw., Pract. Exper.*, **41**, 199–208. Plug-in webpage: `http://wiki.event-b.org/index.php/Event_Model_Decomposition`.

[23] Silva, R. and Butler, M. (2010) Shared event composition/decomposition in Event-B. *Proc. of FMCO'10*, LNCS, **6957**, pp. 122–141. Springer.

[24] Abrial, J.-R. (2009) Event model decomposition. Technical Report 626. ETH Zurich.

[25] Utting, M. and Legeard, B. (2007) *Practical model-based testing - a tools approach.* Morgan Kaufmann.

[26] Ammann, P. and Offutt, J. (2008) *Introduction to software testing.* Cambridge University Press.

[27] Gargantini, A. and Riccobene, E. (2001) ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, **7**, 1050–1067.

[28] Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., and Steffen, B. (2005) On the correspondence between conformance testing and regular inference. *Proc. of FASE'05*, LNCS, **3442**, pp. 175–189. Springer.

[29] Groce, A., Peled, D., and Yannakakis, M. (2002) Adaptive model checking. *Proc. of TACAS'02*, LNCS, **2280**, pp. 357–370. Springer.

[30] Hagerer, A., Hungar, H., Niese, O., and Steffen, B. (2002) Model generation by moderated regular extrapolation. *Proc. of FASE'02*, LNCS, **2306**, pp. 80–95. Springer.

[31] Hungar, H., Niese, O., and Steffen, B. (2003) Domain-specific optimization in automata learning. *Proc. of CAV'03*, LNCS, **2725**, pp. 315–327. Springer.

[32] Meinke, K. and Niu, F. (2012) An incremental learning algorithm for extended Mealy automata. *Proc. of ISoLA'12 - part I*, LNCS, **7609**, pp. 488–504. Springer.

[33] Dupont, P., Lambeau, B., Damas, C., and van Lamsweerde, A. (2008) The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, **22**, 77–115.

[34] Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S. (2007) Reverse engineering state machines by interactive grammar inference. *Proc. of WCRE'07*, pp. 209–218. IEEE Computer Society.

[35] Walkinshaw, N., Derrick, J., and Guo, Q. (2009) Iterative refinement of reverse-engineered models by model-based testing. *Proc. of FM'09*, LNCS, **5850**, pp. 305–320. Springer.

[36] Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M. (2002) Generating finite state machines from abstract state machines. *Proc. of ISSTA'02*, pp. 112–122. ACM.

[37] Bendisposto, J. and Leuschel, M. (2011) Automatic flow analysis for Event-B. *Proc. of FASE'11*, LNCS, **6603**, pp. 50–64. Springer.

[38] Cho, C. Y., Babic, D., Poosankam, P., Chen, K. Z., Wu, E. X., and Song, D. (2011) MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. *USENIX Security Symposium*, pp. 139–154. USENIX Association.

[39] Pasareanu, C. S., Giannakopoulou, D., Bobaru, M. G., Cobleigh, J. M., and Barringer, H. (2008) Learning to divide and conquer: applying the L^* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, **32**, 175–205.

[40] Howar, F., Steffen, B., and Merten, M. (2011) Automata learning with automated alphabet abstraction refinement. *Proc. of VMCAI'07*, LNCS, **6538**, pp. 263–277. Springer.

[41] Chen, Y.-F., Farzan, A., Clarke, E., Tsay, Y.-K., and Wang, B.-Y. (2009) Learning minimal separating DFA's for compositional verification. *Proc. of TACAS'09*, LNCS, **5505**, pp. 31–45. Springer.

[42] Malik, Q., Lilius, J., and Laibinis, L. (2009) Model-based testing using scenarios and Event-B refinements. *Methods, Models and Tools for Fault Tolerance*, LNCS, **5454**, pp. 177–195. Springer.

[43] Shahbaz, M. and Groz, R. (2009) Inferring Mealy machines. *Proc. of FM'09*, LNCS, **5850**, pp. 207–222. Springer.

[44] Howar, F., Steffen, B., and Merten, M. (2010) From ZULU to RERS - lessons learned in the ZULU challenge. *Proc. of ISoLA'10*, LNCS, **6415**, pp. 687–704. Springer.

[45] Fathabadi, A. S., Rezazadeh, A., and Butler, M. (2011) Applying atomicity and model decomposition to a space craft system in Event-B. *Proc. of 3rd NASA Formal Methods (NFM'11)*, LNCS, **6617**, pp. 328–342. Springer.

APPENDIX A

Let $A = \{a_1, \ldots, a_n\}$ be an ordered set, $n > 0$. Then the quasi-lexicographical order on A^*, denoted $<$, is defined by: $x < y$ if $\|x\| < \|y\|$ or $\|x\| = \|y\|$ and $x = za_iv$, $y = za_ju$, for some $z, u, v \in A^*$ and $1 \leq i < j \leq n$. $x \leq y$ is used to denote that $x < y$ or $x = y$.

Let $U \subseteq A^*$ be a finite set and ℓ an integer that is greater than or equal to the length of the longest sequence(s) in U. Let $S \subseteq A^*$ and $W \subseteq A^*$ be the current sets processed by LearnDFCA.

For $s \in S \cup SA$, we define $r(s)$ to be the minimum sequence $t \in S$ such that $s \sim t$, where the minimum is taken according to the quasi-lexicographical order on A^*. In particular, $r(\epsilon) = \epsilon$. Then we define $S_{min} = \{r(s) \mid s \in S\}$.

For every two dissimilar sequences $s_1, s_2 \in S$, we denote by $d(s_1, s_2)$ the minimum element (according to the quasi-lexicographical order) of W that distinguishes between s_1 and s_2. Then we define W_{min} as the set of all such sequences, i.e. $W_{min} = \{d(s_1, s_2) \mid s_1, s_2 \in S, s_1 \not\sim_\ell s_2\}$.

Then the following result holds.

Theorem 6.1. *Suppose that $M(S, W, T)$, the automaton returned by LearnDFCA, is a minimal DFCA of U w.r.t. ℓ. Then the execution of LearnDFCA for inputs $S_0 = S_{min}$ and $W_0 = W_{min}$ will pass both the consistency and closedness checks and the returned DFCA $M(S_{min}, W_{min}, T_{min})$ is isomorphic to $M(S, W, T)$.*

Proof. We prove by contradiction that the procedure passes the consistency check. Otherwise, there exist $w \in W_{min}$, $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \leq \ell - \|w\| - 1$ and $a \in A$ such that $s_1 \sim_k s_2$, where $k = max\{\|s_1\|, \|s_2\|\} + \|w\| + 1$, and $T(s_1aw) \neq T(s_2aw)$. Thus aw distinguishes between s_1 and s_2, but s_1 and s_2 cannot be distinguished by any element in W_{min} of length $\|w\| + 1$. This contradicts the fact that W_{min} contains $d(s_1, s_2)$.

Similarly, if the procedure fails the closedness check, then there exist $s \in S_{min}$, $a \in A$ such that $sa \not\sim t$ $\forall t \in S_{min}$ with $\|t\| \leq \|sa\|$. On the other hand $r(sa) \in S_{min}$ and $\|r(sa)\| \leq \|sa\|$. This provides a contradiction, as required.

Since the state set of $M(S, W, T)$ is S_{min}, the fact that $M(S_{min}, W_{min}, T_{min})$ and $M(S, W, T)$ are isomorphic follows directly from the definition of the DFCA returned by LearnDFCA [6]. □

APPENDIX B

Let A be a finite alphabet, $U \subseteq A^*$ a finite set and ℓ an integer that is greater than or equal to the length of the longest sequence(s) in U; let I be a minimal DFCA of U w.r.t. ℓ. Then the following two results hold.

Theorem 6.2. *If $S_0 \subseteq A^*$ is a proper state cover of I, then LearnDFCA returns a minimal DFCA of U w.r.t. ℓ for inputs S_0 and $W_0 = \{\epsilon\}$.*

Proof. Let $M(S, W, T)$ be the DFCA returned by LearnDFCA. First we prove that W is a strong characterization set of I. We provide a proof by contradiction. If we assume otherwise, then there exist q_1, q_2 states of I and $w = a_1 \ldots a_i \notin W$ with $a_1, \ldots, a_i \in A$, such that q_1 and q_2 are distinguishable by w but indistinguishable by $W \cap A[i]$. Furthermore, w is chosen such that i is the lowest possible value with the above property.. Since $\epsilon \in W$, $i \geq 1$. Let j, $1 \leq j \leq i$, be the largest integer for which $a_j \ldots a_i \notin W$. Let q_1' and q_2' be the states reached by $a_1 \ldots a_{j-1}$ from q_1 and q_2 respectively. Then q_1' and q_2' are distinguishable by $a_j \ldots a_i$. Furthermore, q_1' and q_2' are indistinguishable by $W \cap A[i - j + 1]$. (Assume there exists $z = b_j \ldots b_{i'} \in W$ with $b_j, \ldots, b_{i'} \in A$, $i' \leq i$, which distinguishes between q_1' and q_2'. Then q_1 and q_2 are distinguishable by $a_1 \ldots a_{j-1}b_j \ldots b_{i'}$ but indistinguishable by $W \cap A[i']$. This contradicts the minimality of i.) Let $s_1 \in S$ and $s_2 \in S$ be sequences of minimum length which reach q_1 and q_2, respectively (since S is a proper state cover, such sequences exist). Let $w' = a_{j+1} \ldots a_i$, $s_1' = s_1a_1 \ldots a_{j-1}$, $s_2' = s_2a_1 \ldots a_{j-1}$ and $k = max\{\|s_1\|, \|s_2\|\} + i$. (Note that, since $M(S, W, T)$ is a minimal DFCA, q_1 and q_2 are distinguished by a sequence of length less than or equal to $\ell - max\{\|s_1\|, \|s_2\|\}$; from the minimality of i, it follows that $k \leq \ell$.) Then $s_1' \sim_k s_2'$ and $T(s_1'a_jw') \neq T(s_2'a_jw')$. Since $w' \in W$ (as j is the largest integer for which $a_j \ldots a_i \notin W$), this provides a contradiction as the final observation table must be consistent.

Since I and $M(S, W, T)$ have the same number of states (equal to the number of elements of S_0), the result follows from the W-method for bounded sequences. □

Theorem 6.3. *If $W_0 \subseteq A^*$ is a strong characterization set of I, then LearnDFCA returns a minimal DFCA of U w.r.t. ℓ for inputs $S_0 = \{\epsilon\}$ and W_0.*

Proof. We prove by contradiction that S is a proper state cover of I. If we assume otherwise, then there exist $s = a_1 \ldots a_i \notin S$ with $a_1, \ldots, a_i \in A$, $i \geq 1$, and q a state of I such that q is reached by s but cannot be reached by any sequence in $S \cap A[i]$. Let j, $1 \leq j \leq i$, be the smallest integer for which $a_1 \ldots a_j \notin S$. Then we choose one such s for which j has the minimum possible value and q as above. Let q' be the state reached by $a_1 \ldots a_j$ from the initial state of I. Then q' cannot be reached by any sequence in $S \cap A[j]$. (If there exists $z = b_1 \ldots b_{j'} \in S$ with $b_1 \ldots b_{j'} \in A$, $j' \leq j$, which reaches q' then q is reached by $b_1 \ldots b_{j'}a_{j+1} \ldots a_i$ but unreachable by any sequence in $S \cap A[i]$. This contradicts the minimality of j.) Let $s' = a_1 \ldots a_{j-1}$. Since W is a strong characterization set of I, $s'a_j \not\sim t$ $\forall t \in S$ with $\|t\| \leq \|s'a_j\|$. This provides a contradiction. As above, the final result follows from the W-method for bounded sequences. □

APPENDIX C

We provide a short description of the Event-B models in Table 2 including pointers where they can be retrieved:

1. *A2A* - `http://deploy-eprints.ecs.soton.ac.uk/129/` - Business domain: A model of the Order and Supply Chain A2A Communication using a pattern approach from ETH Zurich.
2. *BepiColombo* - `http://deploy-eprints.ecs.soton.ac.uk/72/` - Aerospace domain: A model of two communication modules in the embedded software on a space craft. The model was constructed by researchers in Southampton [45] based on the feedback from SSF (Space Systems Finland), an industrial partner in the DEPLOY project, who investigates the Event-B method for formal validation of software parts of BepiColombo mission to Mars[17].
3. *CarsOnBridge* - `http://deploy-eprints.ecs.soton.ac.uk/112/` - Pedagogical example: Described in Section 3.2 and [11, Chapter 2]. Note that in our experiments we fixed the constant d (representing the capacity of the bridge) to 2.
4. *Choreography* - see [14] - Business domain: A model of service choreography for enterprise component communication.
5. *MobileAgent* - `http://deploy-eprints.ecs.soton.ac.uk/120/` - Distributed systems domain: A model for distributed computing communication: a routing algorithm for sending messages to a mobile phone, see also [11, Chapter 12].
6. *PressCtrller* - `http://deploy-eprints.ecs.soton.ac.uk/113/` - Embedded control domain: A model of a mechanical press controller adapted from a real system at INRST (Institut National de la Recherche sur la Sécurité du Travail), see also [11, Chapter 3].
7. *ResponseCoP* - `http://deploy-eprints.ecs.soton.ac.uk/301/` - Information flow domain: A model for analysis of information flow policies for Dynamic Virtual Organizations (DVOs), commonly referred to as the Bronze/Silver/Gold structure that frequently arises in multi-agency response to emergencies.
8. *SSFPilot* - `http://deploy-eprints.ecs.soton.ac.uk/58/` - Aerospace domain: A model of a pilot for a complex on-board satellite mode-rich system: Attitude and Orbit Control System (AOCS).
9. *TrainCtrller* - `http://deploy-eprints.ecs.soton.ac.uk/316/` - Automotive Domain: The model specifies a controller that detects the driving mode wished by the train driver. A large number of requirements are taken into account, so a large number of variables and events are needed.

[17]`http://en.wikipedia.org/wiki/BepiColombo`

TABLE 4. Comparing the transformation time (*trans_t*) with the learning time (*learn_t*) for the refinement reuse procedure (in seconds)

Subject	**machine/ℓ**	***trans_t***	***learn_t***
A2A	M3/10	0.22	0.47
	M4/10	0.47	1.33
	M6/10	1.72	4.46
BepiColombo	M1/11	1.44	18.18
	M2/9	2.84	32.26
	M3/12	20.51	223.11
CarsOnBridge	M2/16	5.25	61.08
	M3/12	6.05	43.65
Choreography	M1/10	1.40	4.79
	M1/11	1.56	4.66
MobileAgent	M2/11	1.02	6.36
	M3/12	4.31	57.33
PressCtrler	M1/10	0.42	9.03
	M2/9	4.78	69.18
ResponseCoP	M1/9	3.94	32.23
	M1/12	5.69	243.2
SSFPilot	M1/6	0.59	27.19
	M1/8	0.42	132.34
TrainCtrller	M4/8	0.65	6.88
	M5/8	0.99	15.74
	M7/9	5.17	151.58
	M8/8	5.03	128.92

TABLE 5. Test suite sizes for the benchmark with selected bounds ℓ (note that these are the sizes of the test suites after incremental learning, but before counterexample search)

Subject	**machine/ℓ**	$\|Q\|$	**test suite size**
A2A	M4/11	8	445
	M6/11	14	1518
BepiColombo	M2/11	53	6249
	M3/15	145	34115
CarsOnBridge	M2/13	8	455
	M3/14	53	13569
Choreography	M0/13	5	120
	M1/13	10	957
MobileAgent	M2/12	17	777
	M4/12	33	1964
PressCtrler	M1/8	46	5628
	M2/8	136	26385
ResponseCoP	M1/8	43	5091
	M1/9	65	9539
SSFPilot	M0/10	54	7541
	M1/11	107	25370
TrainCtrller	M6/13	12	4437
	M8/13	20	11203