Model learning and test generation for Event-B using cover automata

Florentin Ipate, Ionut Dinca, and Alin Stefanescu

University of Pitesti, Department of Computer Science
Str. Targu din Vale 1, 110040 Pitesti, Romania
{name.surname}@upit.ro

Abstract. Event-B is a formal modeling method used for the system-level specification and analysis. Event-B models are abstract state machines, whose states are implicitly derived from the values of the model variables. For large variable domains, the explicit state space of a model may be consequently also very large, or even infinite. Even when an explicit state-based model of reasonable size exist, this may not be available beforehand. This can be problematic for test generation techniques which usually rely on explicit states. To mitigate such issues, in this paper we propose an approach that uses the notion of cover automata to iteratively construct a subset of a state space together with an associated test suite. The iterative nature of the algorithm fits well with the notion of refinement from the Event-B method.

1 Introduction

Software reliability is a topic of major interest given the ubiquitousness of software nowadays. Especially in the safety-critical domains but not only, usage of formal methods is gaining ground due to their capability of checking and ensuring correctness and robustness of software systems. Event-B [1] is a formal method for reliable systems specification and verification, being supported by a software platform called *Rodin*[1]. The platform offers several plugins for the editing, refinement, theorem-proving, composition or model-checking of Event-B models. Complementing these techniques, testing based on Event-B models has recently emerged as an interesting research theme [16, 5].

The concept of state is at the heart of testing and many test generation techniques from (extended) finite state machines exist. Event-B models are essentially abstract state machines. However, their states are not given explicitly; instead they can be implicitly derived from the values of the model variables. This means that, in order to apply the aforementioned techniques, an explicit state model has to be constructed first. Furthermore, such a model may have a very large state space (potentially infinite), thus making the construction of the state model and of the associated test sets unmanageable.

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

In this paper we propose an approach in which both an explicit state model of the system and a test suite are constructed in parallel. The model constructed is based on the concept of *finite cover automaton* of a finite set L, which is a finite automaton which accepts all sequences in L but may also accept sequences that are longer than every sequence in L. In practice, an upper bound ℓ on the length of the considered sequences will be established and the constructed model will have to conform to the original Event-B model for all sequences of length at most ℓ. In this way, by appropriately setting the value of the upper bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be mitigated. Furthermore, the approach allows for a gradual construction of the model, thus permitting the integration of the refinement in this process: an initial model can be derived from an original Event-B model M; this can be later detailed by considering the refined version.

Given an Event-B model and an upper bound ℓ, the proposed approach will incrementally construct finite cover automata that will eventually cover all executable sequences of length less than or equal to ℓ. 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 threefold:

- first, it constructs a successive set of finite approximation models for the set of Event-B executable traces up to a length ℓ. The model construction relies on a variant of Angluin's learning automata [2], adapted to finite cover automata [12]. The construction exploits the restriction given by the bound ℓ to obtain a better computational complexity for the case using cover automata compared to original algorithm. Moreover, the cover automata are minimal by construction.
- second, we instrument the above techniques to 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, Event-B method deploys model refinement as a means to handle modeling complexity. 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.

The paper has the following structure. Section 2 presents the theoretical foundations, including the used algorithm for the cover automata. Sections 3 and 4 describe the adaptation to Event-B and its implementation, 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 [12], 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 (number of symbols) 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 $a = bc$. 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. There is an unique (up to a renaming of the state space) minimal DFA that accepts a given regular language.

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. [4], is a DFA that accepts all sequences in U and possibly other sequences that are longer than any sequence in U. A *minimal* DFCA of U is a DFCA of U having the least number of states.

In this paper we use a slightly more general concept, as defined in [12]: 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) [12].

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 (and more importantly from the point of view of practical applications), 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 [12].

2.2 The L^ℓ algorithm for learning finite cover automata

Learning regular languages from queries was introduced by Angluin in [2]; 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 [12], 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 construct 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 labelled by elements of $(S \cup SA) \cap A[\ell]$ and columns labelled 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 [12]. 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_1a and s_2a are also k-similar for all $a \in A$.

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) (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.

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

Table 1. Example: current observation table

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 (details will be provided later on) and it is checked whether the lan-

Procedure LearnDFCA

Input: S, W and the current observation table T.
Repeat
  \— *Check consistency* —\
  For every $w \in W$, in increasing order of $\|w\| = i$ **do**
    Search for $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \leq \ell - i - 1$ 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 for $a \in A$ such that $sa \not\sim t \ \forall t \in S$ with $\|t\| \leq \|sa\|$.
    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 have been processed
Until $\neg new\_row\_added$
Construct $M(S, W, T)$.
Return $M(S, W, T)$.

Fig. 1. The learning procedure *LearnDFCA*

guage 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. 1 (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:

- 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 incorrect consistency check will be processed itself 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 $\|t\| \leq \|sa\|$; 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 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 2 is the resulting observation table. This is both consistent and closed and so the DFA $M(S, W, T)$ is constructed.

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 2. Example: new observation table

The state set of DFA $M(S, W, T)$ is formed by taking all minimum, mutually dissimilar sequences from S, where the minimum is taken according to the quasi-lexicographical order on A^* [12]. For Table 2, this is $Q = \{\epsilon, a, b, aa\}$ (since $bb \sim aa$) and the corresponding DFA is as represented in Figure 2 (throughout this paper, in graphical representations of DFAs, final states are drawn in double line, whereas non-final states are drawn in single line; the initial states are labelled by $q0$). The formal definition of $M(S, W, T)$ is given in [12], 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 [12].

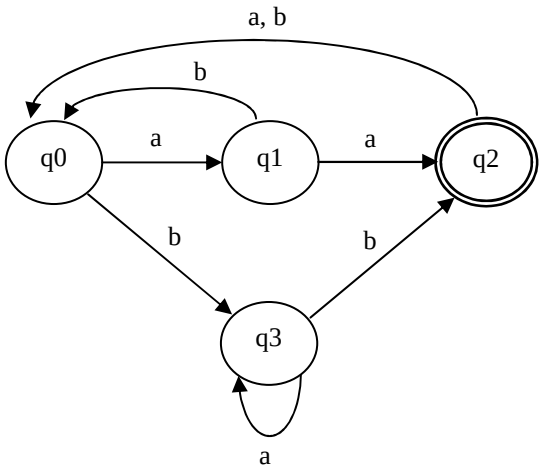


Fig. 2. The DFA corresponding to Table 2

2.3 Black-box testing for finite cover automata

Before proceeding, we briefly outline the W-method for bounded sequences, which we proposed in [11]. 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 language 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, naturally, I is not known; however, it is assumed that the maximum number of states of I can be estimated and 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 k; however, the W-method produced 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: *proper 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 the 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; 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 $X_k = SA[k+1]W \cap A[\ell]$. 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 exists.

The W-method for bounded sequences, as proposed by Ipate [11], is a non-trivial generalization of the W-method for checking functional equivalence (also called the Vasilevski-Chow method) [**?**,**?**]. Note that [11] gives the results for Mealy machines; above, we adapted them for finite state machine acceptors.

3 Model learning and test generation for the Event-B framework

We are now ready to present how we can apply the above theoretical harness. We first describe the Event-B modeling environment. Then we show how we repeatedly apply the learning algorithm to obtain cover automata for Event-B models and how the conformance test suite is obtained. The notion of refinement is also touched upon at the end of the section.

3.1 Event-B framework

Event-B method [1] 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 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 model-checking using the ProB tool [13]. 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, which is a extensible Eclipse-based tool. The current efforts of developing Event-B and Rodin are concerted in a large European project, DEPLOY[2], which also includes industrial partners (Siemens, Bosch, 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 systems is supposed to maintain during its execution. The state of the modeled system is given by the values of variables. 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

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

be executed, and the *action*, which is a set of assignments that change the value of the global variables. Although not standard, a notion of *final state* can be defined in Event-B using logical predicates on the global variables, i.e. a state is considered final if a certain predicate is true. For instance, we can have a special variable called `final_variable` together with a predicate [`final_variable = TRUE`].

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).

Testing for Event-B is a recent topic in DEPLOY project. Since Event-B machines are particular types of abstract state machines (ASMs), testing in Event-B shares commonalities with testing of ASMs [8]. Given an Event-B model, a test case can be defined as a sequence of events that can be executed in the model. Optionally, it could also be required that the state reached after the execution is final. The executability of the 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.

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 by 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 [16]. 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 [5]. 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. In the next section we will propose a technique that provides a way of generating a test suite and thus complementing [5] and furthermore, the state space explosion of [16] is kept under control by incremental model learning putting a certain bound on the lengths of the 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 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 event sequences of length maximum ℓ that end in a final state. Note that 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 (executable paths through

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

Fig. 3. The iterative procedure of constructing the DFCA

the model) and the actual test data to drive these paths. In this subsection we discuss the model learning cycle and in the following subsection 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. 3. Note that, 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, so computational resources will be saved. 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 **vom adauga o noua sectiune sau apendice** 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. Naturally, the ProB model checker will be of great assistance in finding such a counterexample. There are several possibilities to do this:

– interactively using the experience of the human testers that 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.

- 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 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. The 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 observation table, so, if the Event-M 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 or we are satisfied that the DFCA is correct. Note that size of the test set is exponential in k and so using the W method for a large k may be expensive.
- 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 we only consider the first one, where the counterexample is manually provided (which is in fact in the spirit of the original Angluin's algorithm).

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. 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 $S_0 = \{\epsilon\}$, $W_0 = A$. (Since in many cases, states can be distinguished by singleton sequences, initially we consider that W contains 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 is not totally accurate and needs to be improved. This may happen 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 existing Event-B model has been refined and extra detail has been added (using the Event-B refinement)
  - **Subcase 3:** the Event-B has not been changed but the associated DFCA is deemed 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.

In this (second) case, S_0 and W_0 are 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 these some 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 Figure 3. [3] Note that 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. **Comentariu: algoritmul a fost modificat astfel incat sa returneze seturile S si W minimizate.** 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 $(S \cup SA)W \cap A[\ell]$ for which $T = 1$ need to be re-checked (since the guards of the refined events cannot be weaker than the corresponding guards of the original events, sequences that are found not to be feasible in the original model will also be infeasible in the refined model); furthermore, T will also be expanded in the second subcase, by adding the newly introduced events. Finally, in the third subcase, only the sequences in $(S \cup SA)W$ whose length is greater than the previous ℓ need to be re-processed.

Unul sau mai multe exemple care sa ilustreze convingator fiecare dintre cazuri si subcazuri

Example. We illustrate the iterative process of constructing the DFCAs with a system for controlling the cars on a narrow bridge. This example is particularly relevant since it is used to introduce the main Event-B concepts in Abrial's textbook [1].

The modelled 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. There are also some car sensors situated at both ends of the bridge which are used to detect the presence of a car entering or leaving the bridge. 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 Figure 4). The first model M_0 is very simple. The events ML_out and ML_in correspond to

[3] 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.

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. 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. In addition, the events IL_in and IL_out correspond to cars entering and leaving the island respective. 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 make the colors change in a more disciplined way, two more variables ml_pass and il_pass are introduced: $ml\_pass = TRUE$ signifies that at least one car has passed the bridge going to the island since the mainland traffic light last turned green; similarly for $il\_pass = TRUE$.

In the first iteration, we apply the procedure for $\ell = 3$. For each model, the resulted DFCAs (plotted using our plug-in; for simplicity, we removed the sink states, i.e. the states from which no final state can be reached) and the generated test suites are as presented in Figure 5. **Florentin: Testele trebuiesc minimizate** We start the process of generating the DFCA for the abstract machine M_0 with initial sets $S_0 = \{\epsilon\}$ and $W_0 = A$. **Florentin: Ionut, ati facut experimente si pentru cazul $S_0 = A$ and $W_0 = A$? Daca nu produc rezultate mai bune, ar valida intuitia noastra ca e suficient sa luam $S_0 = \{\epsilon\}$.** The obtained DFCA (Figure 5(a)) is correct. For the first refinement M_1, we start the procedure with S and W from the previous iteration. It can be observed that the resulted DFCA (Figure 5(b)) is not totally accurate; for example, the execution path $p_1 = ML\_out, ML\_out, IL\_out, ML\_in$ (of length 4), which is an executable path in the machine M_1, is not included in the generated model. Similarly, the DFCA for M_2 is not correct since, for example, the path $p_2 = ML\_tl\_green, ML\_out1, ML\_out2, IL\_in, IL\_in$ of length 5 is not included in the automaton. Furthermore, the events IL_out1, IL_out2, ML_in and IL_tl_green are not even covered by the associated model (Figure 5(c)).

In an attempt to fix these errors, we increase the value of ℓ by 1. We can observe (Figure 6) that the DFCAs have been improved compared to the initial versions. The execution path p_1 is included now in the DFCA associated to M_1, and moreover, the automaton is correct now (Figure 6(a)). The DFCA of M_2 (Figure 6(b)) has also been improved, but it is still incorrect. The event IL_tl_green is covered now, but the path p_2 is not included and the events IL_out1, IL_out2 and ML_in are still not covered.

Next, we consider the case $\ell = 5$. The DFCA (Figure 7)is improved (for example, the event IL_out2 is covered now), but the path p_2 is still missing. In order to illustrate the iterative construction presented in Figure 3, we choose to

---

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

---

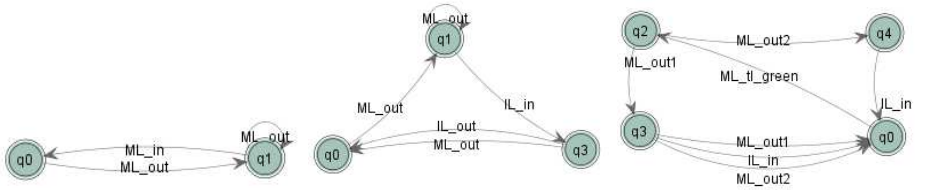
Machine M_2 refines M_1 :
  Variables: $a, b, c, ml\_tl, il\_tl, il\_pass, ml\_pass$
  Event INITIALISATION $\widehat{=}$ begin $(a, b, c := 0, 0, 0)$,
    $(ml\_tl, il\_tl := red, red)$,
    $(il\_pass, ml\_pass := 1, 1)$
    end
  Event *ML_out1* refines *ML_out* $\widehat{=}$ when $ml\_tl = green \ \wedge \ a + b + 1 < d$
    then $a, ml\_pas := a + 1, 1$ end
  Event *ML_out2* refines *ML_out* $\widehat{=}$ when $ml\_tl = green \ \wedge \ a + b + 1 = d$
    then $(a, ml\_pas := a + 1, 1), ml\_tl := red$ end
  Event *IL_out1* refines *IL_out* $\widehat{=}$ when $il\_tl = green \ \wedge \ b > 1$
    then $(b, c := b - 1, c + 1), il\_pas := 1$ end
  Event *IL_out2* refines *IL_out* $\widehat{=}$ when $il\_tl = green \ \wedge \ b = 1$
    then $(b, c := b - 1, c + 1), (il\_tl, il\_pas := red, 1)$
    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

---

Fig. 4. The first two refinements of the "Cars on the bridge" example (from Abrial [1]).

provide the path p_2 as a counterexample for the current DFCA associated to M_2. The resulted DFCA is presented in the Figure 8. It can be observed that the path $q_0 \rightarrow q_2 \rightarrow q_3 \rightarrow q_8 \rightarrow q_4 \rightarrow q_7$ in DFCA corresponds to the path p_2 in M_2.

$(a)M_0$ $(b)M_1$ $(c)M_2$

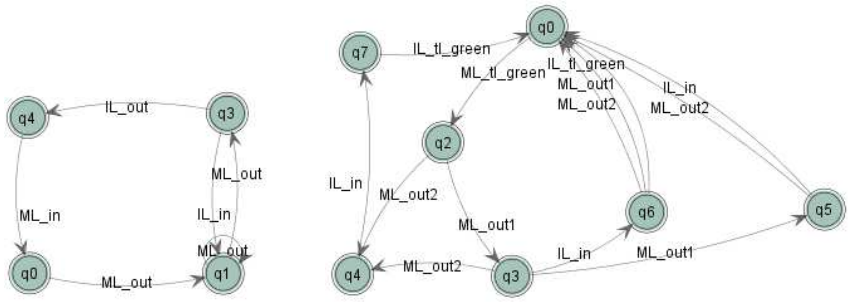


1: ML_out
2: ML_out,ML_out
3: ML_out,ML_in
4: ML_out,ML_out,ML_out
5: ML_out,ML_out,ML_in
6: ML_out,ML_in,ML_out

1: ML_out
2: ML_out,ML_out
3: ML_out,IL_in
4: ML_out,ML_out,ML_out
5: ML_out,ML_out,IL_in
6: ML_out,IL_in,ML_out
7: ML_out,IL_in,IL_out

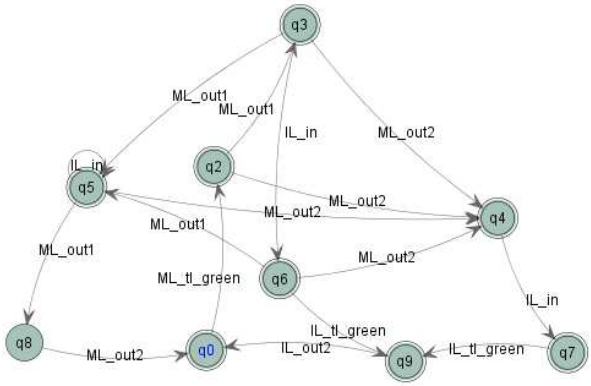
1: ML_tl_green
2: ML_tl_green,ML_out1
3: ML_tl_green,ML_out2
4: ML_tl_green,ML_out1,ML_out1
5: ML_tl_green,ML_out1,ML_out2
6: ML_tl_green,ML_out1,IL_in
7: ML_tl_green,ML_out2,IL_in

Fig. 5. The DFCAs and the generated test suites for $\ell = 3$.



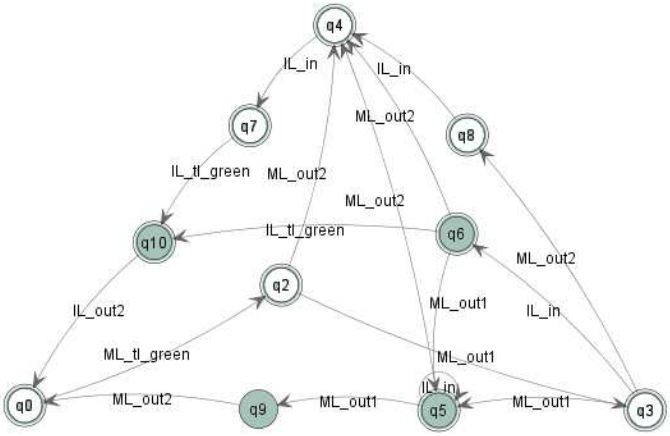
$(a)M_1$ $(b)M_2$

Fig. 6. The DFCAs for $\ell = 4$.



$counterexample:\ ML\_tl\_green, ML\_out1, ML\_out2, IL\_in, IL\_in$

Fig. 7. The DFCA for the machine M_2, $\ell = 5$.



$path:\ q_0 \rightarrow q_2 \rightarrow q_3 \rightarrow q_8 \rightarrow q_4 \rightarrow q_7$

Fig. 8. The DFCA for the machine M_2, $\ell = 5$, after providing the counterexample.

3.3 Test data generation

In order to decide whether a given sequence s, $\|s\| \leq \ell$, is accepted or not by the DFCA (i.e., $s \in L^\ell$ 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 $s \in L^\ell$; otherwise, the path is declared infeasible[4] and so $s \notin L^\ell$. Therefore, deciding whether $s \in L^\ell$ or not reduces to finding test data to execute the corresponding path of the Event-B model. In this way, when the process of constructing the DFCA is completed, a conformance test for the Event-B model has also been obtained: this is the set of test data for all paths in $(S \cup SA)W$ [11].

Having all this mechanisms in place, all it remains to specify are 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 [13]. The second reduces the problem to an optimization problem, which is then solved using search-based techniques (genetic algorithms) [5]. 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, $2^{card(V)}$.

3.4 Test suite construction

As mentioned before, we can use information from the observation table to generate a set of test case that consist of all the sequences $X_0 = (S \cup SA)W \cap A[\ell]$ for which $T(x) = 1$ (we consider only the test cases for which $T(x) = 1$ since test data can only be generated for feasible paths). Again, in the above formula of X_0, only the minimized S and W are used. Following [11], 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). Conformance testing is especially relevant in the embedded systems domain.

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 is an advantage of our approach. Another advantage is the fact that the method is interactive, so the tester can use its 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 by coverage criteria, where the produced tests may not be intuitive or may not cover existing standard testing scenarios in the domain. Regarding coverage criteria, if a very specific

[4] Note that here *infeasible* means only that our tools could not find test data 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.

coverage criteria is sought (cf. [8]), 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 desired coverage. Moreover, if a simpler coverage criteria like event coverage is desired, the obtained test suite can be reduced by choosing a smaller subset of it that satisfies the requirement.

The test data generation for the test cases was already discussed in the previous subsection.

3.5 Relation to Event-B refinement

The incremental approach of L^ℓ allows us to reuse the learned model and test suite of an abstract model AM to the next more concretized model CM. For instance, if the refinement from AM to CM introduces a new event e, the event will be added to W and the learning procedure is triggered, the observation table being extended. A technical discussion about the reuse in case of refinement was already performed in the previous subsection (see Subcase 2 above).

4 Experimental results

We have implemented the previous algorithms and in this section we provide the results of the experiments on a couple of Event-B models. The implementation was done in Java as an Eclipse plugin to Rodin platform. The membership query was implemented using the constraint-solving functionality of ProB.

We chose a benchmark of three Event-B models of small to medium sizes from the publicly available DEPLOY model repository[5]. Two of them are modeled using more than 3 levels of refinements. The models are presented below:

- *PurchaseQuantity*: This is an overly simplified Event-B model inspired from the business domain that has 3 global integer variables and 4 events.
- *GCD*: This is an Event-B model[6] using mathematical and set operators to describe a more sophisticated variant of an algorithm to compute the "Greatest Common Divisor" (GCD). It was used as a running example in [3]. It has 6 refinement levels. It starts on the first level with 3 integer variables and 1 event and ends up with 15 variables (integer and partial function types) and 5 events.
- *BepiColombo*: This is an abstract model[7] of two communication modules in the embedded software on a space craft. The Event-B model was provided by the SSF (Space Systems Finland), an industrial partner in DEPLOY project, who tries to use the Event-B method for formal validation of software parts of BepiColombo mission to Mars[8]. The model has several levels of refinements

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

[6] `http://deploy-eprints.ecs.soton.ac.uk/229/1/gcd.zip`

[7] `http://eprints.ecs.soton.ac.uk/22048/5/Rodin_Space_Craft.zip`

[8] See `http://deploy-eprints.ecs.soton.ac.uk/72/1/BepiColombo_-_Modelling_Approach.pdf` and `http://en.wikipedia.org/wiki/BepiColombo`

Table 3. Number of states of DFCAs

	PurchaseQuantity	GCD					BepiColombo		
	M_0	M_0	M_1	M_2	M_3	M_4	M_0	M_1	M_2
	4ev.	1ev.	1ev.	3ev.	5ev.	5ev.	5ev.	10ev.	12ev.
$\ell = 3$	6	1	1	4	4	4	4	6	6
$\ell = 4$	6	1	1	4	4	4	6	10	11
$\ell = 5$	6	1	1	4	4	4	8	17	19
$\ell = 6$	6	1	1	4	4	4	10	25	28
$\ell = 7$	6	1	1	4	4	4	10	34	38
$\ell = 8$	6	1	1	4	4	4	10	43	45
$\ell = 9$	6	1	1	4	4	4	10	47	47
$\ell = 10$	6	1	1	4	4	4	10	47	47

(combined with model decompositions), but we only considered the first 3 refinement levels. These are described in [7]. The modeling approach starts on the first level with 5 set-type variables and 5 events and ends up with 18 variables and 16 events.

We have applied the model learning and test generation algorithm to the above 3 models gradually at the different refinement levels and increasing ℓ from 3 to 10. Table 3 presents the sizes of the constructed DFCAs for the different refinement levels (together with their corresponding number of events) and different values of ℓ. The times to compute them ranged from a couple of milliseconds to several minutes for the complex models. We only present the refinement levels for which solutions were found within a running time limit of 7 minutes. The generation of the conformance test suite based on the observation table T (as presented in Subsection 3.4) took basically no resources. The most expensive part was of course given by the repeated membership queries on the Event-B machines.

Moreover, in Fig. **??** we provide 3 examples of the generated DFCAs (plotted using the GraphViz package and removing the sink states, i.e. the state not reaching a final state). In the graphical representation, the final states are drawn in double line, whereas non-final states will be drawn in single line; the initial states are labeled by $q0$. We note also that the generated approximated DFCA for the GCD resembles the generated control flow graph presented in [3]. The DFCAs should provide insights to the modelers and testers on the behavioral structure of the Event-B model.

We can conclude that the approach worked relatively well for the given example, even though it showed its limits to some of the most complex model instances.

5 Related work

The correspondence between conformance testing and automata learning is discussed, from a theoretical point of view, by Berg et al. [**?**], which show how

results from one area can be transferred to the other. Such a correspondence is also exploited in our paper; in this case, however, the correspondence is between conformance testing for bounded sequences and cover automata learning. Furthermore, here we propose an adaptive learning and test generation approach, in which the results obtained for a previous (incomplete or inaccurate) model are reused for the current system.

An adaptive approach to model development is proposed by Groce et al. [10], but the emphasis here is on model checking, rather than test generation, as in our approach. Furthermore, a DFA (not a DFCA) model of the system is build. 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 automata learning techniques for test generation is also discussed by Hagerer et al. [**?**] and Hungar et al. [**?**]. As above, both paper 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 provides further support for our approach, which considers bounded sequences.

Bounded model checking (based on SAT methods) [14] 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. [6] and Walkinshaw et al. [15] use automata inference techniques to construct behavior models of software systems. However, there are a number of key differences from our approach. Firstly, the learning techniques used are essentially passive inference methods, in which a set of training data is supplied to the algorithm for model construction. In order to construct an accurate model, training sequences which 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. Secondly, 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 potential to strike the right balance between accuracy and costs. Naturally, at limit (for ℓ sufficiently large), our approach will also produce an exact model of the system.

Grieskamp et al. [9] construct finite automata that under-approximate the global state space of an ASM. They use an algorithm that combine several concrete states into abstract ones using logical formulas to distinguish the abstract

states. The obtained finite automaton is used for test generation. Using a similar idea and extra information on logical dependences between Event-B guards, [3] constructs an abstract over-approximation of the control flow graph of an Event-B model. Compared to our approach based on incremental learning, both these last approaches use a different approach relying on state merging and logical formulas for distinguishability.

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 adapting and applying model learning algorithms for testing purposes to the Event-B method. It 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 for models of medium size. As future plans, we want to extend the experimentation benchmark using existing models from the DEPLOY repository and identify and address the current bottlenecks. Moreover, we plan to implement the language equivalence query which was currently done manually by interactively providing a counterexample.

Acknowledgment This work was partially supported by project DEPLOY, European FP7 grant no. 214158, and Romanian Research Grant CNCS-UEFISCDI no. 7/05.08.2010 at the University of Pitesti.

References

1. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
2. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
3. Jens Bendisposto and Michael Leuschel. Automatic flow analysis for Event-B. In *Proc. of FASE'11*, volume 6603 of *LNCS*, pages 50–64. Springer, 2011.
4. Cezar Câmpeanu, Andrei Paun, and Jason R. Smith. Incremental construction of minimal deterministic finite cover automata. *Theoret. Comput. Sci.*, 363(2):135–148, 2006.
5. Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Test data generation for Event-B models using genetic algorithms. In *Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11)*, volume 181 of *CCIS*, pages 76–90. Springer, 2011.
6. Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1&2):77–115, 2008.
7. Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In *Proc. of 3rd NASA Formal Methods (NFM'11)*, volume 6617 of *LNCS*, pages 328–342. Springer, 2011.

8. Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.
9. Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *Proc. of ISSTA'02*, pages 112–122. ACM, 2002.
10. Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 357–370. Springer, 2002.
11. Florentin Ipate. Bounded sequence testing from deterministic finite state machines. *Theoret. Comput. Sci.*, 411(16-18):1770–1784, 2010.
12. Florentin Ipate. Learning finite cover automata from queries. *Journal of Computer and System Sciences*, 2011. In Press. Online at: http://doi:10.1016/j.jcss.2011.04.002.
13. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008. Tool webpage: `http://www.stups.uni-duesseldorf.de/ProB`.
14. Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.
15. Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proc. of WCRE'07*, pages 209–218. IEEE Computer Society, 2007.
16. Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proc. TESTCOM'09*, volume 5826 of *LNCS*, pages 179–194. Springer, 2009.

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 $W \subseteq A^*$ 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, $r(\epsilon) = \epsilon$.

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. $W_{min} = \{d(s_1, s_2) \mid s_1, s_2 \in S, s_1 \not\sim_l s_2\}$.

Then the following result holds.

Theorem 1. *Suppose that $M(S, W, T)$, the automaton returned by LearnDFCA is a minimal DFCA of U w.r.t. ℓ and let $S_{min} = \{r(s) | s \in S\}$ and $W_{min} = \{d(s_1, s_2) \mid s_1, s_2 \in S, s_1 \not\sim_l s_2\}$. 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 $M(S, W, T)$, 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 [12].

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 and let I be a minimal DFCA of U w.r.t. ℓ. Then the following two results hold.

Theorem 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 $w = a_1 \dots a_i \notin W$ with $a_1, \dots, a_i \in A$, and q_1, q_2 states of I such that q_1 and q_2 are distinguishable by w but indistinguishable by $W \cap A[i]$. Since $\epsilon \in W$, $i \geq 1$. Let j, $1 \leq jlei$, be the largest integer for which $a_j \dots a_i \notin W$. Then we choose one such w for which j has the minimum possible value and q_1 and q_2 as above. Let q_1' and q_2' be the states reached by $a_1 \dots a_{j-1}$ from q_1 and q_2 respectively. Then q_1' and q_2' are distinguishable by $a_j \dots a_i$. Furthemore,q_1' and q_2' are indistinguishable by $W \cap A[\|i - j + 1\|]$. (Assume there exists $z = b_j \dots b_{i'} \in W$ with $b_1 \dots b_{i'} \in A$, $i' \leq i$, which distinguishes between q_1 and q_2. Then q_1 and q_2 are distinguishable by $a_1 \dots a_{j-1} b_j \dots b_{i'}$ but indistinguishable by $W \cap A[i']$. This contradicts the minimality of j.) 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 $w' = a_{j+1} \dots a_k$, $s_1' = s_1 a_1 \dots a_{j-1}$, $s_2' = s_2 a_1 \dots a_{j-1}$ and $k = max\{\|s_1\|, \|s_2\|\} + i + 1$. Then $s_1' \sim_k s_2'$ and $T(s_1' a_j w') \neq T(s_2' a_j w')$. 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 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 $s = a_1 \dots a_i \notin S$ with $a_1, \dots, a_i \in A$, $i \geq 1$, 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, $1 \leq jlei$, 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.