Model learning and test generation for Event-B decomposition

Ionut Dinca, Florentin Ipate, 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 method for reliable systems specification and verification, which uses model refinement and decomposition as techniques to scale the design of complex systems. In previous work, we proposed an iterative approach for test generation and state model inference based on a variant of Angluin's learning algorithm, which integrates well with the notion of Event-B refinement. In this paper, we extend the method to work also with the mechanisms of Event-B decomposition. Two types of decomposition, i.e. shared-events and shared-variables, are considered and the generation of a global test suite from the local ones is proposed at the end. The implementation of the method is evaluated on publicly available Event-B decomposed models.

1 Introduction

Event-B [1] is a formal method for reliable systems specification and verification, which was introduced about ten years ago and was tuned up in several industrial-academic projects. Event-B models are a type of abstract state machines in which a set of global variables are changed by so called events. When the guard of an event is satisfied, its action code can executed having an effect on the global variables. The main modeling approach in Event-B relies on the notion of refinement, i.e., the modeler starts with an abstract model which is iteratively enriched and concretized by capturing more and more features of the system to be specified. Each refinement step is accompanied by formal proofs for properties of interest for the system. 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 [2, 3]. There are two main types of decomposition: shared events style [4, 5] and shared variables style [6, 7]. 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.

The current efforts of further developing Event-B are concerted in a large European project, DEPLOY[1], which also includes industrial partners from the embedded and business applications domains (Bosch, Siemens, SAP, SSF). The

[1] European FP7 project (2008-2012): http://www.deploy-project.eu

main platform supporting Event-B, called Rodin [8], is an extensible Eclipse-based tool offering different capabilities such as model refinement, model decomposition, theorem-proving, and model-checking. Complementing the formal verification, *test generation* from Event-B is a recent topic of interest backed by concrete requirements from industry.

Essentially, in order to generate test suites for an Event-B model one has to first construct an equivalent automaton and then apply one of the many finite state based test generation techniques existing in the literature [9, 10]. However, as the states of this equivalent automaton are given by the combinations of the model global variables, this may lead to the well-known state explosion problem. In order to address such issue, in our previous work, we have developed an automata learning and test generation approach [11], implemented in a Rodin plug-in [12], that constructs a finite state *approximation* and an associated test suite for an Event-B model. The core of the method relies on a variant of Angluin's algorithm [13] adapted to finite cover automata [14]. A finite cover automaton (CA)[15] 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. A powerful (conformance) test suite, including appropriate test data, is obtained as a by-product of the learning algorithm. Last but not least, the whole procedure can be applied incrementally, allowing the reuse of the learned model and test cases from the abstract to the more concrete levels of refinement.

The main contribution of this paper is an extension of the above method that integrates not only the Event-B refinement mechanism, but also the different Event-B decomposition styles. More precisely, for decomposition, we investigate the generation of CAs for the sub-models by reusing information via projections from the global model. Also vice-versa, for the recomposition operation we can reuse the information from the CAs of the sub-models for the construction of a CA for the global model. Conformance test suites are also generated alongside. Finally, an integrated approach involving both refinements and (de)compositions in an Event-B development chain is proposed.

The paper is structured as follows. The next section presents prerequisites from formal languages and automata theory. Section 3 shortly recalls the previous work on automata learning for Event-B and Section 4 introduces the extension of this work to Event-B decomposition and recomposition operators. Section 5 provides experiments on publicly available Event-B models, while Section 6 concludes the paper.

2 Preliminaries

In this section we provide theoretical prerequisites on finite automata, cover automata and product automata, together with their accepted languages.

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 $h : Q \times A^* \longrightarrow Q$, where $A^* := \bigcup_{i \geq 0} A^i$. 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, called the *language accepted by* M, is defined by $L_M = \{s \in A^* \mid h(q_0, s) \in F\}$. A DFA M is called *minimal* if any DFA that accepts L_M has at least the same number of states as M. A classic results states that there exists a unique (up to a renaming of the state space) minimal DFA that accepts a given regular language [16].

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. [15], 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 [14]: 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. $L_M \cap A[\ell] = U$, where $A[\ell] := \bigcup_{0 \leq i \leq \ell} A^i$. 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) [14].

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 [14].

Product automata and projections - general concepts. We now provide a couple of definitions and results for product automata and languages. This is a prerequisite for the setting of decomposed Event-B models that we present later on. To simplify the presentation, we only consider the case the two automata, but the definitions and the results hold also for more than two automata.

We start by describing formally the product of two automata synchronizing on their common input symbols. First of all, since the two automata have different input alphabets A_1 and A_2, their transition function is extended to the whole set of symbols $A = A_1 \cup A_2$ using the following definition. Given DFA $M = (B, Q, q_0, F, h)$ and $B \subset A$ we define the DFA $Ext_A(M) = (A, Q, q_0, F, h')$ by: for every $q \in Q$ and $a \in A$, $h'(q, b) = h(q, b)$ if $b \in B$ and $h'(q, a) = q$ if $a \in A \setminus B$.

When the two automata operate on the same input alphabet, their product can be described in a traditional fashion, as follows:

Definition 1. *Let* $M_1 = (A, Q_1, q_{01}, F_1, h_1)$ *and* $M_2 = (A, Q_2, q_{02}, F_2, h_2)$ *be two DFAs. Then we define the DFA* $M_1 \times M_2 = (A, Q, q_0, F, h)$ *by:* $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, $F = F_1 \times F_2$ *and for every* $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in A$, $h((q_1, q_2), a) = (h_1(q_1, a), h_2(q_2, a))$.

Thus, for two DFAs M_1 and M_2 over alphabets A_1 and A_2, we denote by $M_1 \parallel M_2 := Ext_A(M_1) \times Ext_A(M_2)$ the *product automaton* over alphabet $A = A_1 \cup A_2$ capturing the synchronization on common symbols of M_1 and M_2. This is similar to the standard synchronization of labeled transition systems used in the literature (see e.g. [17]).

The languages accepted by product automata are characterized by the so-called *product languages*. For their definition, we first need the notion of *projection*. Given a sequence $s \in A^*$ and $A_1 \subset A$, the projection of s on A_1, denoted by $proj_{A_1}(s)$, is the sequence obtained from s by removing all symbols not in A_1. For a language $L \subseteq A^*$, $proj_{A_1}(L) = \{proj_{A_1}(s) \mid s \in L\}$. Now, we can define the notion of *product language*:

Definition 2. *Let* A_1 *and* A_2 *be two alphabets, not necessarily disjoint, and* $A := A_1 \cup A_2$. *Then, a language* $L \subseteq A^*$ *is called a* product language *(over* A_1 *and* A_2*) if and only if there exist two languages* $L_1 \subseteq A_1^*$ *and* $L_2 \subseteq A_2^*$ *such that*

$$L = \{w \in A^* \mid proj_{A_1}(w) \in L_1 \text{ and } proj_{A_2}(w) \in L_2\}.$$

Moreover, there exist also a useful result (see e.g. [18]) proving that a product language is always the product of its projections, i.e. languages L_1 and L_2 in the previous definition can be replaced by $proj_{A_1}(L)$ and $proj_{A_2}(L)$, respectively. Finally, the expected result relating the languages of product automata with product languages says that:

Proposition 1. *[18] The class of regular product languages coincides with the class of languages accepted by products of DFAs.*

Corollary 1. *For a finite alphabet* $A := A_1 \cup A_2$, *let* $L \subseteq A^*$ *be a regular product language, and* M_1 *and* M_2 *be two DFAs for* $proj_{A_1}(L)$ *and* $proj_{A_2}(L)$, *respectively. Then,* $L = L_{M_1 \parallel M_2}$.

Since any finite language is also a regular language, Corollary 1 holds also when L is a finite product language. Therefore, we can easily derive:

Corollary 2. *For a finite alphabet* $A := A_1 \cup A_2$, *let* $U \subseteq A^*$ *be a finite product language and* ℓ *a positive bound (larger than the size of any word in* U*). If* M_1 *and* M_2 *are two DFCAs w.r.t.* ℓ *for* $proj_{A_1}(U)$ *and* $proj_{A_2}(U)$, *then* $M_1 \parallel M_2$ *is a DFCA w.r.t.* ℓ *for* U.

3 Cover automata based learning and test generation for Event-B

In this section we present the main elements of the approach proposed in [11], that can incrementally construct a series of finite state approximations and corresponding test suites for a series of Event-B refined models. Before that, we need to provide the basic elements of Event-B.

A short introduction to Event-B. Event-B [1] is a formal methodology having its mathematical foundations rooted in set theory and first order logic. A Event-B specification consists of a static part called *context* and a dynamic part called *machine*. A context defines a set of datatypes as carrier sets, constants and axioms that relate the constants to the carrier sets. A machine will be specified by a set of global *variables* and a set of *events*, which are the first-class citizens of the formalism. Moreover, a set of *invariants* captures the properties of the specified system. Proof obligations solved (automatically or manually) by the supporting platform will ensure that the invariants are always true, i.e. both before and after the execution of any event.

An event is an element consisting of a set of *local parameters*, a *guard* and an *action* code. An event *evt* has the following general form:

$$evt \;\widehat{=}\; \textbf{any } t \textbf{ where } G(t,v) \textbf{ then } S(v,t) \textbf{ end}. \tag{1}$$

Above, t is a set of local parameters, v is a set of global variables appearing in the event, G is a predicate over t and v, called the guard, and $S(v,t)$ represents a substitution. If the guard of an event is false, the event cannot occur and is called disabled. The substitution S modifies the values of the global variables in the set v. It can use the old values from v and the parameters from t. For example, an event that adds a number i smaller than 9 to a global variable n, in case n is greater than 15, is modeled as:

$$increment \;\widehat{=}\; \textbf{any } i \textbf{ where } i \in \mathbb{N} \wedge i < 9 \wedge n > 15 \textbf{ then } n := n + i \textbf{ end}.$$

The semantics of an Event-B model is based on the execution of its events. First of all, a special event called *Initialisation*, which does not have a guard, is executed; usually, its action will set initial values to the global variables. Then, in a loop, all the guards of the events are evaluated and the set of enabled events is established. From them, one event is nondeterministically chosen and its action is executed, some of the variables being updated. The process then iterates. Note that the state space of the model is not explicit, but is implicitly given by the evolving values of the 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 a feasible (i.e. executable) path through the Event-B model, or negative, otherwise. The feasibility of a test case implies the existence of appropriate *test data* for the events, i.e. an appropriate initialization of the global variables and suitable values for the local parameters,such that all the guards of the events in the sequence are satisfied. Furthermore, a *test suite* is by definition a collection of test cases.

Given an Event-B model Z having its set of events denoted by E, we can define the language of Z to be the set of feasible sequences over E, i.e.

$$L(Z) := \{w \in E^* \mid w \text{ is feasible in } Z\}.$$

Note that $L(Z)$ is not regular in general, since one can easily simulate a two-counter machine in Event-B, so the formalism is Turing-complete [16]. However,

we can naturally obtain a regular subset by considering only a finite subset of $L(Z)$, namely the sequences of length up to a bound ℓ, i.e. $L(Z, \ell) := L(Z) \cap E[\ell]$.

Finally, the *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) [1].

Incremental model learning based on cover automata. In [11] we present an automata learning and test generation procedure for Event-B: given an Event-B model Z and a positive bound ℓ, we produce a DFCA M for $U := L(Z, \ell)$ and an associated test suite. The procedure can be iteratively used for a series of model refinements.

The core of the procedure is based on a modification of Angluin's learning algorithm [14] that is specialized to finite languages, and that is more efficient than the original Angluin's algorithm, called L^*, for regular languages [13].

In a similar but not trivial way, in [14] we extend Angluin's work by proposing an algorithm, called L^ℓ, for learning a DFCA. Given an unknown finite set $U \subseteq A^*$ 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. Similar to the L^* algorithm, two properties of the observation table are defined: *consistency* and *closedness*.

The algorithm starts with $S = W = \{\epsilon\}$. It periodically checks the consistency and closedness properties and extends the table accordingly using membership queries. When both conditions are met, the DFA $M(S, W, T)$ corresponding to the table is constructed and it is checked whether the language L accepted by $M(S, W, T)$ satisfies $L \cap A[\ell] = U$. If this 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.

The iterative procedure of the algorithm for Event-B is shortly presented below. The technical details can be found in [11]. The main idea is that we evolve the observation table based on previous versions of it, by reusing existing information whenever possible. In particular, for the Event-B refinement, the observation tables of the refined model is not generated from scratch, but from the table of the abstract model that is refined, so unlike the original L^ℓ algorithm, the procedure does not start with empty S, W and T, but with some initial values S_0, W_0 and T_0, which reflect the current knowledge about the DFCA model. An important observation is that, for efficiency reasons, in the recalculation of the observation table only a part of the previous information is sufficient, viz. $S_{min} \subseteq S$ and $W_{min} \subseteq W$, which satisfy certain properties: they are a proper state cover and strong characterization set, respectively (see [11] for definitions).

For the first execution of the procedure, 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 $S_0 = \{\epsilon\}$, $W_0 = \{\epsilon\} \cup E$, where E is the set of events. Note that the alphabet A from L^ℓ above is now the set E. When the procedure has been applied at least once, previous information can be reused. If model is not totally accurate and needs to be improved, we can distinguish the different reasons for that:

- **Case 1:** the Event-B model has been modified or augmented due to changes in the requirements.
- **Case 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.
- **Case 3:** the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). In this case, information from the abstract model can be reused in the computation of the refined model.

A test suite TS can be derived from the observation table as follows:

$$TS := \{t \in E^* \mid t \in ((S \cup SE) \cap E[\ell])W \text{ such that } T(t) = 1\}. \tag{2}$$

Note that we only take positive test cases into account in TS. However, we could also use the existing information about infeasible sequences, i.e. T(t)=0, to generate negative tests, if such a testing requirement exists. Moreover, in (2) we usually take S and W to be the sets S_{min} and W_{min} mentioned above. Furthermore, the test cases from TS are provided with the test data that prove their feasibility. The test data is obtained during the construction of the observation table T, because the membership queries, i.e. feasibility checks, are implemented using a dedicated set-based constraint solver for Event-B, which also returns the values of variables and local parameters for a given feasible sequence. As discussed in [11], TS 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 suite is

more powerful than test suite based on simple 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.

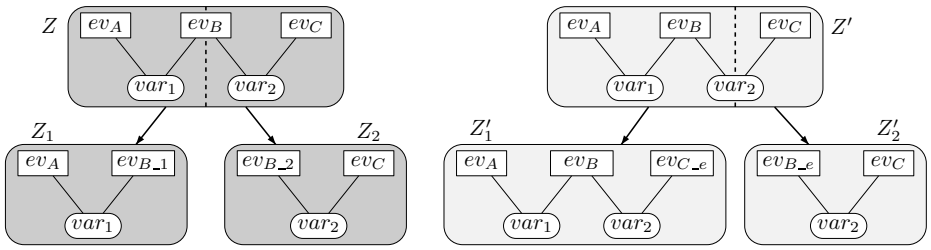
4 Model learning for Event-B decomposition

4.1 Event-B decomposition styles

There are two main decomposition styles in Event-B: shared-events [4, 5] and shared-variables [6, 7]. Other variants like *atomicity decomposition* [4, 19] or *modularization* [20, 3] also exist, but we do not address them in this paper for the following reasons. Since the atomicity decomposition is in fact a special case of refinement, our method in [11] works for it out-of-the-box. On the other hand, modularization defines a different approach to decomposition that reuses a submodel in several other models using interface specifications, so we leave its investigation to the future (moreover, there is some yet to be solved integration issues between the modularization plug-in and the Event-B constraint solver that we use).

Shared events decomposition. In the case of shared events decomposition, an Event-B model is decomposed into several sub-models such that all its events and variables are distributed over the local models. As the name suggests, the local sets of events may have common events (shared events). However, the local sets of variables are disjoint, i.e. the partition of the variables will determine the structure of the decomposition. The left hand side of Fig. 1 presents a minimalistic example of shared events decomposition. At the top, we have a global model Z with three events $\{ev_A, ev_B, ev_C\}$ and two global variables $\{var_1, var_2\}$. The lines between the events and variables suggests the dependencies between them, e.g $ev_A - var_1$ means that var_1 appears in the guard or/and action of ev_A. Assume that the modeler chose to distribute the variables over two sub-models: the first one, denoted Z_1, takes over var_1, and the second, Z_2, takes over var_2. Then, the events are distributed to Z_1 and Z_2 according to the distribution of the variables, so Z_1 has ev_A and ev_B as events (because they depend on var_1) and similarly, Z_2 has ev_B and ev_C as events. In this case, ev_B is a shared event for Z_1 and Z_2.

However, there is a technical issue to be solved for ev_B; the fact that ev_B depends on both var_1 and var_2, while the local models contain only one of the variables. This means that the local events corresponding to ev_B, denoted in Fig. 1 by ev_{B_1} and ev_{B_2}, will only be restricted versions of ev_B that only depend on var_1 and var_2, respectively. So, for the decomposition to be possible, ev_B should have such a form that "separates" the use of var_1 and var_2 in its guards and actions. This is a task for the modeler that should design the Event-B specification in this way as a preparation step for decomposition (refinement may be use in previous modeling steps to achieve this goal). Below, we present ev_B, ev_{B_1}, and ev_{B_2} using the general form of an event in (1):



(a) shared event decomposition (b) shared variable decomposition

Fig. 1. The shared event vs. shared variable decomposition styles

$$
\begin{aligned}
ev_B \;&\widehat{=}\; \textbf{any } t, t_1, t_2 \textbf{ where } G_1(t, t_1, var_1) \wedge G_1(t, t_2, var_2) \\
&\qquad\qquad \textbf{then } S_1(var_1, t, t_1); S_2(var_2, t, t_2) \textbf{ end.} \\
ev_{B\_1} \;&\widehat{=}\; \textbf{any } t, t_1 \textbf{ where } G_1(t, t_1, var_1) \textbf{ then } S_1(var_1, t, t_1) \textbf{ end.} \\
ev_{B\_2} \;&\widehat{=}\; \textbf{any } t, t_2 \textbf{ where } G_2(t, t_2, var_2) \textbf{ then } S_2(var_2, t, t_2) \textbf{ end.}
\end{aligned}
$$

Above, we see that ev_B has a set of local parameters t, t_1, t_2, a guard that is the conjunction of two guards using var_1 and var_2 separately, and also an action that can be split into two actions that do not mix the two global variables. The local events will then only use the parts of the guards and actions that refer to their corresponding global variable. Without going into details, it is also important to observe the existence of the common local parameter t, which can be used for passing data between ev_{B_1} and ev_{B_2}. This makes the shared event decomposition suitable for specifying distributed systems communicating via message-passing [19]. Finally, we mention also the fact that the decomposition mechanism is correct in the sense of Event-B refinement [1], by proving specific proof obligations (e.g. deadlock freedom) and putting restrictions on the subsequent refinements of the shared events in the local sub-models.

The decomposition operation induces the inverse operation of *composition*, for which a dedicated Rodin plug-in exists [21]. It takes a input two models Z_1 and Z_2 that may have events with the same name and constructs a composed model Z (look at Fig. 1 bottom-up). Z is obtained by putting together the variables and events Z_1 and Z_2, taking care that the local shared events are merged by concatenating their guards and actions following the same scheme as for ev_{B_1}, ev_{B_2}, and ev_B above.

Shared variables decomposition. Let us also touch upon the shared variables decomposition, using the exemplification on the right of Fig. 1. In this case, we partition the set of events and then distribute the variables. If we partition the events of Z' into $\{ev_A, ev_B\}$ and $\{ev_C\}$, due to the variables dependences, the sub-models Z'_1 and Z'_2 have the variables $\{var_1, var_2\}$ and $\{var_2\}$, so they share variable var_2. However, since sub-models have in fact two copies of the shared variable, they need to learn the changes made to the shared variable in the other sub-models. This is implemented adding so-called *external* events. For instance, in addition to its "native" event ev_C, Z'_2 will also include an external event

ev_{B_e} that is a restricted version of ev_B, that only simulates its effect on var_2. Note that the shared variables decomposition is suitable for the specification and verification of parallel programs [7].

4.2 Learning and test generation for shared events decomposition

In the rest of the paper, we will present our approach only for the shared events decomposition. We can do this without loss of generality based on the observation that, for our purposes, the shared variables decomposition can be reduced to the shared event decomposition as follows. Suppose Z' is decomposed using shared variables into Z'_1 and Z'_2 and the decomposition is based on the partition of set of events E of Z' into E_1 and E_2 Assume that $E_{11} \subseteq E_1$ is the set of external events for Z_1 and $E_{21} \subseteq E_2$ the set of external events for Z_2. Then, if we duplicate the shared variables and consider each of the two Event-B components to work on its own copy (the definition of the shared variables ensures that they process the two copies identically), the shared variables decomposition can be transformed into a shared events decomposition of Z' into sub-models with set of events $E'_1 = E_1 \cup E_{21}$ and $E'_2 = E_2 \cup E_{12}$, respectively.

Before we proceed, we establish a formal relation between Event-B decomposition and the theory of product languages from Section 2. The proofs of the theoretical results can be found in the long version of our paper [22].

Lemma 1. *Let Z be an Event-B model, which is decomposed into Z_1 and Z_2. Then, for any w sequence of events in Z, w is feasible if and only if, $proj_1(w)$ and $proj_2(w)$ are both feasible in Z_1 and Z_2, respectively.*

Using Lemma 1 and Definition 2 for product languages, we can show that:

Proposition 2. *Let Z be an Event-B model, which can be decomposed into Z_1 and Z_2. Then, the language of Z, $L(Z)$, is a product language over $E := E_1 \cup E_2$, where E_1 are the events of Z_1 and E_2 are the events of Z_2.*

As an immediate corollary, the result hold also when we impose a bound ℓ, i.e. $L(Z, \ell)$ is also a product language, so Corollary 2 can be applied.

Next we now show how our learning and test generation method can be applied to the two operations of decomposition and composition.

Approach for decomposition. Let Z be an Event-B model and E the set of events of Z. We assume that Z is decomposed, using the shared events scheme, into models Z_1 and Z_2 with event sets E_1 and E_2, respectively. Given a bound ℓ, our goal is to obtain DFCAs and associated test suites for Z, Z_1, and Z_2. Although one can apply the method in Section 3 directly and separately on Z, Z_1, and Z_2, we would like to improve the process by reusing information.

We assume that we have a DFCA M and a test suite TS for Z. Then, the DFCA learning procedure for Z_1 will not start with $S_1 = \{\epsilon\}$, as when no previous model is available, but with the set $S_1 = \{proj_1(s) \mid s \in S_{min}\}$, where S_{min} is the proper state cover derived from the DFCA model of Z. The set

W_1 is initialized with $E_1 \cup \{\epsilon\}$. Similarly for Z_2. We could also to start with a projection of the set W obtained for Z (i.e. $W_1 = \{proj_1(s) \mid s \in W_{min}\}$), but, this may not improve performance since W usually contains only singletons [11].

With this input, the learning procedure may not produce a correct DFCA M_1 for Z_1 from the beginning and more iterations may be needed. The reason is that, even though Lemma 1 ensures that a feasible path in Z is projected to a feasible path in Z_1, the projection S_1 may not be rich enough to cover all the states of M_1. This can be understood from the fact that, in general, there is no concrete relation between the sizes of a minimal DFA of a regular language $L \subseteq A^*$ and of the minimal DFA of its projection on a sub-alphabet $A' \subset A$. Thus, the size of a minimal DFA accepting the projection $proj_{A'}(L)$ can be smaller, equal to, or even exponentially larger than the size of the minimal DFA accepting L [23]. The same holds even when L is a finite language. Moreover, in the specific case of Event-B decomposition, the DFCAs of the sub-models may be larger not only because of the effects of the projections just mentioned, but also because there might exist more feasible paths in the projections due to the weakened guards of the shared events, with the effect that the DFCAs for the local sub-models have more states. However, our experiments showed that our choice of S_1 will speed up the learning procedure, generating richer DFCAs in less time compared to the procedure of learning an DFCA for Z_1 from scratch.

Approach for composition. The inverse operation to decomposition is that of composition [21, 5]. Given two models Z_1 and Z_2 with event sets E_1 and E_2, one can construct an Event-B model Z that synchronizes on the shared events.

There are several ways in which we can construct a global DFCA model and/or a test suite for Z from Z_1 and Z_2 or their DFCAs:

1. Construct Z and then apply the techniques of [11] to derive a DFCA and a test suite associated to Z. In this case, there is no reuse of information from Z_1 and Z_2.
2. Construct the two DFCAs M_1 and M_2 for Z_1 and Z_2 and then construct the product $M_1 \parallel M_2$, minimize it and denote it M_{min}. Then, construct a test suite TS from the minimal DFCA M_{min} using a W-method adapted to bounded testing [10]. For every test sequence s for M_{min}, the test data generation process will check if $proj_1(s)$ and $proj_2(s)$ are test sequences for M_1 and M_2, respectively. If this is the case, the test data values for $proj_1(s)$ and $proj_2(s)$ will be reused. This variant is sound due to Corollary 2.
3. Construct only a global test suite TS from the local test suites TS_1 and TS_2 by composing individually the test cases, i.e. $TS := \{t \in E^* \mid proj_1(t) \in TS_1 \text{ and } proj_2(t) \in TS_2\}$. (Optionally, apply a symmetry reduction by only keeping traces in TS that are not equivalent modulo swapping of independent events.)
4. Construct directly a DFCA for the composed model Z without applying the composition of Z_1 and Z_2, nor the product of M_1 and M_2. This is done by applying a learning algorithm for global sequences of events (of length up to ℓ) and answering the global membership queries via answering the local membership queries for the projections (this is sound because of Lemma 1).

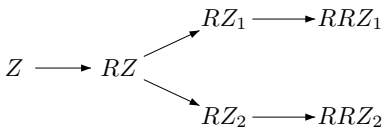


Fig. 2. A sample of decomposition flow

The first two proposals above are correct, i.e. the obtained automata are DFCAs with respect to $L(Z, \ell)$, while the last two are heuristics that in our experiments produced reasonable results, even though they are in general only approximations.

Approach for integrated process. Finally, let us sketch how the above proposals can be integrated in our incremental, refinement based, model learning and test generation strategy presented in [11].

Figure 2 describes a typical incremental development in Event-B involving decomposition. 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. 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. RZ will constitute the basis for the construction of the DFCAs for RZ_1 and RZ_2 starting the learning procedure with the projections as previously explained. 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. These latter models are used to produce a DFCA model and tests for the overall system by one of the methods proposed for the composition operator.

5 Experiments

We implemented the methods for decomposition presented in this paper, extending our Rodin plug-in that previously only addressed refinement [12]. 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.

For the benchmark, we investigated all the publicly available Event-B models involving decomposition from the DEPLOY repository[2]. From the total of eight found models, we could not use two of them because they involved some advanced data types that were not yet supported by the Event-B constraint solver deployed for the membership queries. From the rest of six models, the first three use shared events and the last three use shared variables. Their dimensions are presented in Table 1. The first column gives their name together with a reference. The second column gives the evolution of the models by the operations of refinement and decomposition in a similar fashion to Fig. 2. The '/' symbol represents a refinement step, while '{' depicts a decomposition. For instance, for BepiColombo_SE, there are three refinement steps $m_0/m_1/m_2/m_3$, followed by

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

Table 1. The dimensions of 6 models from DEPLOY repository (development process, no. of events and no. of variables)

Subject	Development process	No. events ($m_0/m_1\ldots$)	No. variables ($m_0/m_1\ldots$)
BepiColombo_SE from [19]	$m_0/m_1/m_2/m_3 \left\{ \begin{matrix} m_4/m_6/m_7 \\ m_5 \end{matrix} \right.$	$6/11/13/17 \left\{ \begin{matrix} 15/19/23 \\ 10 \end{matrix} \right.$	$5/10/12/16 \left\{ \begin{matrix} 12/16/20 \\ 4 \end{matrix} \right.$
UpdateMaster_SE from [3]	$m_0/m_1/m_2 \left\{ \begin{matrix} m_3/m_5/m_7 \\ m_4/m_6/m_8 \end{matrix} \right.$	$5/6/6 \left\{ \begin{matrix} 4/5/5 \\ 4/6/6 \end{matrix} \right.$	$4/5/5 \left\{ \begin{matrix} 4/6/6 \\ 3/8/8 \end{matrix} \right.$
Monitor_SE from [3]	$m_0/m_1/m_2 \left\{ \begin{matrix} m_3/m_6/m_9 \\ m_4/m_7/m_{10} \\ m_5/m_8/m_{11} \end{matrix} \right.$	$7/7/7 \left\{ \begin{matrix} 7/5/5 \\ 4/6/6 \\ 4/6/6 \end{matrix} \right.$	$4/6/6 \left\{ \begin{matrix} 2/4/4 \\ 2/3/3 \\ 2/3/3 \end{matrix} \right.$
Monitor_SV from [3]	$m_0/m_1/m_2 \left\{ \begin{matrix} m_3/m_6/m_9 \\ m_4/m_7 \\ m_5/m_8/m_{10} \end{matrix} \right.$	$7/11/11 \left\{ \begin{matrix} 9/11/11 \\ 10/10 \\ 7/7/9 \end{matrix} \right.$	$4/4/4 \left\{ \begin{matrix} 2/5/5 \\ 3/4 \\ 3/4/6 \end{matrix} \right.$
QResponse_SV	$m_0/m_1/m_2/m_3/m_4 \left\{ \begin{matrix} m_5 \\ m_6 \\ m_7 \end{matrix} \right.$	$2/3/4/5/5 \left\{ \begin{matrix} 3 \\ 5 \\ 3 \end{matrix} \right.$	$2/3/5/7/9 \left\{ \begin{matrix} 4 \\ 7 \\ 4 \end{matrix} \right.$
FindP_SV from [7]	$m_0 \left\{ \begin{matrix} m_1/m_3/m_4/m_5 \\ m_2 \end{matrix} \right.$	$6 \left\{ \begin{matrix} 4/5/6/6 \\ 4 \end{matrix} \right.$	$5 \left\{ \begin{matrix} 3/4/5/6 \\ 3 \end{matrix} \right.$

a decomposition of m_3 into m_4 and m_5; then, m_4 is further refined to m_6 and m_7. The third and forth columns provide the corresponding numbers of events and global variables for the models. For example, BepiColombo starts at m_0 with 6 events and 5 global variables, increases its complexity via refinement to m_3 which exhibits 17 events and 16 variables. and ends up having 23 events and 20 variables for the last refinement m_7 of one of the sub-models.

Note that the search space in BepiColombo case can be very large. E.g. the third refinement m_3 of BepiColombo has 17 events, so for $\ell := 8$ the number of possible sequences or tests of length up to 8 is 17^8 which is almost equal to $7 \cdot 10^9$. Moreover, to this complexity we have to add the computation time for test data for the generated test cases. The constraint solver performing this task need to address a search space implied by 16 global variables of type *Set* and 17 local parameters appearing in the events.

In our experiments, we checked the feasibility of our approach and the scalability of the implementation, by performing the steps for the integrated process at the end of the previous section, i.e. we incrementally construct DFCAs for the refinement and decomposition from abstract model to more concrete levels, combining the (integration) tests at the end using a method for composition. Due to space constraints, we provide the tables with experimental results only in the extended version of our paper [22]. However, we report a successful generation of DFCAs and test suites within reasonable time (max. 6 minutes) for sufficiently high values of ℓ (up to 13 for smaller models). Moreover, the experiments confirmed that the reuse improves the quality of the generated DFCAs (i.e. more states compared to learning from scratch) and reduces the computation time in many cases.

6 Conclusions

In this paper, we presented a method for automata learning and test generation that can be applied along the specification process of Event-B. We focused on the

mechanism of decomposition, because this is an important way of dealing with the large models that may occur in industrial practice. Our approach makes use of the advantages of cover automata and its soundness is based on the theory of product languages. In the future, we will continue to improve the prototype e.g. by a better (UI) integration with decomposition and composition plug-ins [2, 21] and extend its use to the modularization plug-in [20]. We will also investigate the quality of the generated test suites using mutation testing techniques.

In the end, we mention a couple of related papers, even though they solve different problems in different settings. First, we are not aware of any work that generates test cases for Event-B decomposed models, see e.g. [24] and the references therein. An idea of using model projections combined with automata learning for black-box testing of components is presented in [25]. Our relation between learning and conformance test suite is similar to the one presented in [26]. Learning is also used for the generation of communicating automata [27, 28] and for compositional verification of system components [17].

Acknowledgments. This work was supported by project DEPLOY, FP7 EC grant no. 214158, and Romanian National Authority for Scientific Research (CNCS-UEFISCDI) grant no. PN-II-ID-PCE-2011-3-0688 (project MuVet) and grant no. 7/05.08.2010.

References

1. Jean-Raymond Abrial. *Modeling in Event-B – System and Software Engineering.* Cambridge University Press, 2010.
2. Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition tool for Event-B. *Softw., Pract. Exper.*, 41(2):199–208, 2011. Plug-in webpage: `http://wiki.event-b.org/index.php/Event_Model_Decomposition`.
3. Thai Son Hoang, Alexei Iliasov, Renato Silva, and Wei Wei. A survey on Event-B decomposition. *ECEASST*, 46:1–15, 2011.
4. Michael Butler. Decomposition structures for Event-B. In *Proc. of Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.
5. Renato Silva and Michael Butler. Shared event composition/decomposition in Event-B. In *Proc. of FMCO'10*, volume 6957 of *LNCS*, pages 122–141. Springer, 2010.
6. Jean-Raymond Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.
7. Thai Son Hoang and Jean-Raymond Abrial. Event-B decomposition for parallel programs. In *Proc. of ASM'10*, volume 5977 of *LNCS*, pages 319–333. Springer, 2010.
8. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010. Tool available online at: `http://sourceforge.net/projects/rodin-b-sharp`.
9. David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – A survey. *Proc. of the IEEE*, 84(8):1090–1123, 1996.
10. Florentin Ipate. Bounded sequence testing from deterministic finite state machines. *Theoret. Comput. Sci.*, 411(16–18):1770–1784, 2010.

11. Florentin Ipate, Ionut Dinca, and Alin Stefanescu. Model learning and test generation using cover automata. Submitted, 2012.
12. Ionut Dinca, Florentin Ipate, and Alin Stefanescu. Learn and test for Event-B – a Rodin plugin. In *Proc. of ABZ'12*, volume 7316 of *LNCS*, pages 361–364. Springer, 2012. Plug-in webpage: `http://wiki.event-b.org/index.php/MBT_plugin`.
13. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
14. Florentin Ipate. Learning finite cover automata from queries. *Journal of Computer and System Sciences*, 78:221–244, 2012.
15. Cezar Câmpeanu, Nicolae Sântean, and Sheng Yu. Minimal cover-automata for finite languages. *Theoret. Comput. Sci.*, 267(1–2):3–16, 2001.
16. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Ed.)*. Addison-Wesley, 2006.
17. Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L^* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.
18. P.S. Thiagarajan. A trace consistent subset of PTL. In *Proc. of CONCUR'95*, volume 962 of *LNCS*, pages 438–452. Springer, 1995.
19. 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 Symposium (NFM'11)*, volume 6617 of *LNCS*, pages 328–342. Springer, 2011.
20. Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Supporting reuse in Event B development: Modularisation approach. In *Proc. of ASM'10*, volume 5977 of *LNCS*, pages 174–188. Springer, 2010. Plug-in webpage: `http://wiki.event-b.org/index.php/Modularisation_Plug-in`.
21. Michael Poppleton. The composition of Event-B models. In *Proc. of ABZ'08*, volume 5238 of *LNCS*, pages 209–222. Springer, 2008. Plug-in webpage: `http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B`.
22. `http://tinyurl.com/isola12-with-appendix` – extended version of our paper.
23. Galina Jirásková and Tomás Masopust. State complexity of projected languages. In *Proc. of DCFS'11*, volume 6808 of *LNCS*, pages 198–211. Springer, 2011.
24. Jacques Julliand, Nicolas Stouls, Pierre-Christophe Bué, and Pierre-Alain Masson. Syntactic abstraction of B models to generate tests. In *Proc. of TAP'10*, volume 6143 of *LNCS*, pages 151–166. Springer, 2010.
25. Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In *Proc. of TestCom/FATES'07*, volume 4581 of *LNCS*, pages 319–334. Springer, 2007.
26. Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In *Proc. of FASE'05*, volume 3442 of *LNCS*, pages 175–189. Springer, 2005.
27. Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from MSCs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010.
28. Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In *Proc. of ISoLA'10, part 1*, volume 6415 of *LNCS*, pages 658–672. Springer, 2010.