Symbolic Execution of UML-RT State Machines

Technical Report 2011-578

Karolina Zurowska and Juergen Dingel

Modeling And Analysis in Software Engineering
School of Computing
Queen's University
Kingston, Canada

June, 2011

Copyright ©2011 Karolina Zurowska, Juergen Dingel

Abstract

One of the languages used in the industrial practice of the model-driven development (MDD) is UML-RT. The language is a proper profile of UML 2 and it targets especially development of embedded systems. In UML-RT, UML-RT State Machines are used to model behavior. This paper presents a technique for a symbolic execution of these machines, which introduces modular treatment of action code. This feature clearly separates the symbolic execution of the state machine itself from the symbolic execution of its action code and thus facilitates support of different action languages. The separation is achieved via a formalization of UML-RT State Machines in which functions are used to represent the result of the symbolic execution of the action code. Key parts of the technique are formalized, an implementation is presented and an example is used to illustrate the symbolic execution itself and how it can be used for different purposes including reachability analysis, invariant checking, output analysis and test case generation. The evaluation of our tool on two case studies is also discussed.

1 Introduction

Symbolic execution of programs was proposed more than 30 years ago by King. In [21], symbolic values serve as placeholders for input parameters. These symbolic values are used in a traversal of the control-flow graph (CFG), which results in a symbolic execution tree. The tree contains all execution paths and the constraints the symbolic values have to satisfy such that a particular path in the CFG is followed. Symbolic execution can be used to solve many problems related to program analysis, verification and testing [11]. Recently, a lot of research has been devoted to the development of novel extensions to and applications of symbolic execution [27]. Most notably, combining symbolic and concrete execution has been very fruitful and led to testing techniques that are now in serious industrial use (e.g., [17]).

Model-driven development (MDD) advocates the use of models as primary artifacts for software development (rather than code): models are iteratively refined until code can be generated from them and model-level analyses provide early feedback regarding the quality of the models. MDD has been used successfully for the development of, e.g., telecommunication, avionics, military, and automotive systems and is supported by several academic and commercial tools including Scade Suite [4], Motorola Mousetrap [35], IBM Rational Rhapsody®[1] [2], IBM Rational®RoseRT and IBM Rational®Software Architect Real Time Edition (IBM RSA-RTE) [3]. RoseRT and RSA-RTE use UML-RT [31], a modeling language and proper profile of UML 2 [5]. UML-RT features capsules as active objects with their behavior described with state machines. UML-RT State Machines are a special case of UML 2 state machines with some added constraints (e.g., no orthogonal regions, a.k.a, "and"-states) and refinements (for executability) and are fully consistent with UML 2 State Machines. Just like state machines in UML 2, UML-RT State Machines typically contain action code, i.e., code written in some programming language (IBM RSA-RTE supports C++, Java, and Alf [26]) that can be associated with transitions or states (as entry actions, do actions, or exit actions). Although it was recognized long ago that UML-RT lends itself to formal analysis [36], current UML-RT tools still do not take advantage of this. The work presented in this paper is inspired by the recent success of symbolic execution of source code and aims at, ultimately, making its benefits available to the industrial practice of MDD in general and MDD using UML-RT in particular.

To this end, this paper makes the following contributions:
1) An approach to the symbolic execution of UML-RT State Machines (see Figure 1) is presented. The approach is based on the traversal of the state space of an internal representation of the state machine, called Functional Finite State Machine (FFSM). The FFSM is built "on-the-fly" during the traversal (procedure `UMLRT2FFSM` in Figure 1) and any action code encountered in the UML-RT State Machine is incorporated into the FFSM in a modular way in terms of "functions". This modularity is a key distinguishing feature of our approach; the clear separation of the symbolic

[1]IBM, Rational and Rhapsody are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide

execution of the state machine itself from the action code not only facilitates implementation and support for different action languages (an external symbolic execution engine for the action code can be used in a "pluggable" way), but also optimization via reuse similar to the work in [16, 6, 34]. We formally define FFSMs and provide algorithms for their symbolic execution (Section 3) and construction from a UML-RT State Machine (Section 4).
2) The implementation of the approach is described. The implementation consists of Eclipse plugins which can be integrated into IBM RSA-RTE, thus allowing further experimentation with the symbolic execution of UML-RT, including, e.g., variants that combine symbolic and concrete execution. The use of the implementation is illustrated on a running example and a model we obtained from our industrial partner (Section 5).
3) A demonstration of the utility of symbolic execution for different analyses of models of reactive systems including reachability analysis of state machine locations, invariant checking, output analysis, and test case generation (Section 5) is given.

The literature on the analysis of state machines (expressed in, e.g., UML 2, UML-RT, or forms of labeled transition systems) is rich. However, as we will argue in Section 6, to the best of our knowledge, none of the existing approaches combines all the features listed above.

2 Symbolic execution of state machines

The goal of our work is to provide a technique for symbolic execution of UML-RT State Machines and to enable their subsequent analysis. Two key challenges are the reactive nature of UML-RT State Machines and their complex behavioral semantics based on action code [31]. In order to deal with semantics we propose a formalization of UML-RT State Machines with functional finite state machines (FFSMs). To incorporate reactive features in the analysis of UML-RT State Machines we define the symbolic execution of FFSMs. Both of the above elements are introduced below and presented in detail in the following sections.

An example of an FFSM (defined in Section 3) is shown in Figure 2. FFSMs have *input/output actions* and variables, which are divided into *machine variables*, common to all states, and variables that store input and output values called *input action variables* and *output action variables*, respectively. Additionally, transitions are labeled with three types of functions: *guard*, *update* and *output*. The first type is used to guard a transition and represents a condition defined over machine variables and input variables of the input action. The other two types of functions are defined over the same set of variables, but they represent updates of machine variables and sequences of output actions, respectively. Both of these functions, as justified in Section 3, are defined using cases and represented as pairs with the first element being a condition that must be satisfied in order to apply a given update or to produce an output sequence.

FFSMs capture the most important characteristics of UML-RT State Machines such as states (called locations), transitions, events and variables. Moreover, functions represent the results of the symbolic execution of action code. The separation of action code analysis, along with the generality of FFSMs, make it possible to use them to represent other kinds of state machines, for

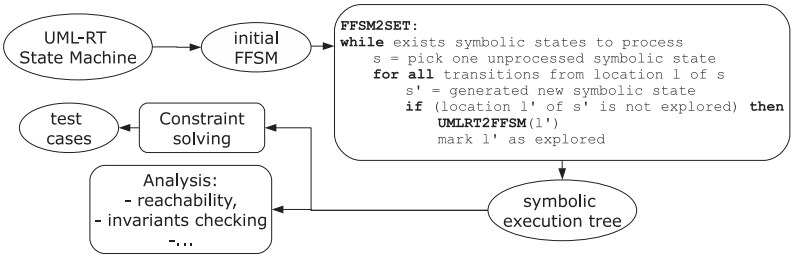

Figure 1: Symbolic execution of UML-RT State Machines.

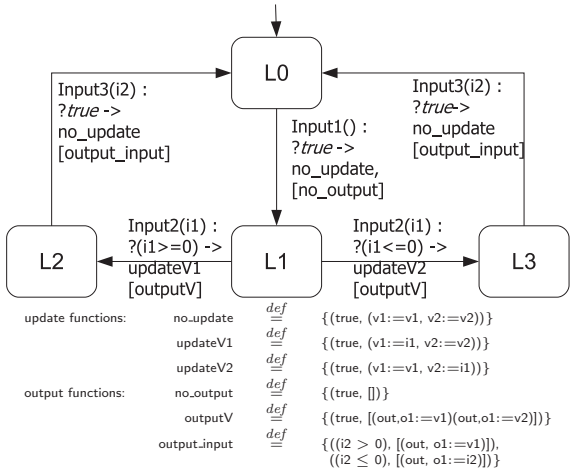

Figure 2: An example of an FFSM with machine variables `v1, v2`, input action variables: `i1, i2` and output action variables `o1` (labels of transitions are of the form `input_action(input_action_variables)? (guard) -> update [output]`).

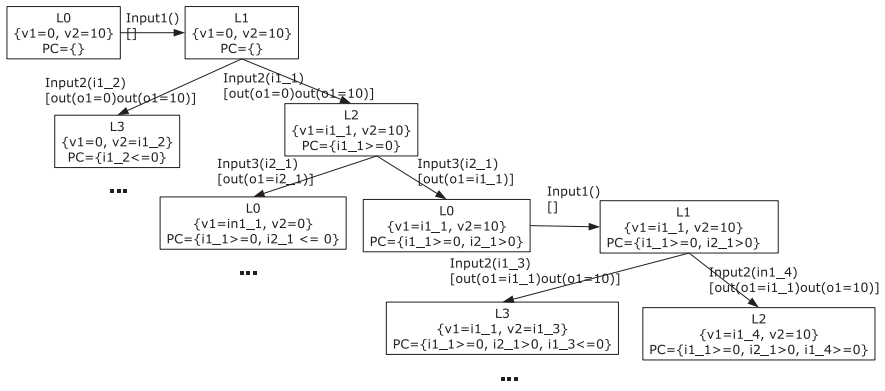

Figure 3: An initial part of a symbolic execution tree for the FFSM in Figure 2

example the more general UML 2 State Machines [5].

Figure 3 presents the first states of the symbolic execution tree of the FFSM in Figure 2. The initial symbolic state is `L0` and the initial values of the two machine variables are 0 for `v1` and 10 for `v2`. In FFSMs, machine variables are always initialized. This assumption is motivated by the fact that machine variables represent attributes in UML-RT models (as presented in detail in Section 4), and attributes always have initial values. From the initial symbolic state, the transition to `L1` is executed. Next, there are two possible transitions from `L1` triggered by the action `Input2` with input variable `i1`. To take the transition to `L2` the input variable is given a symbolic value $i1_1$ that must be greater or equal to 0. This constraint is included in the path constraint of the next symbolic state. The evaluation of the update function `updateV1` yields an update of `v1` to the symbolic value $i1_1$. Symbolic processing of the output function `outputV` generates a sequence of output actions using the values of both machine variables.

The symbolic execution of FFSMs conveniently represents the reactive nature of UML-RT State Machines, because inputs and outputs are part of every transition in the symbolic execution tree. Additionally, variables of enabled input actions receive new symbolic values.

3 Functional finite state machines and their symbolic execution

Functional finite state machines (FFSMs) are inspired by X-machines [18] and by Input-Output Symbolic Transition Systems (IOSTSs) [13]. As in X-machines, transitions in FFSMs are labeled with functions, but their definition is based on first-order logic structures as in IOSTSs (although in our case inputs and outputs are on the same transition). The explicit specification of functions capturing how the machine processes variables is crucial for the formulation of a symbolic execution algorithm. Also, as we will see in Section 4, FFSMs are defined in such a way that they can express UML-RT State Machines, which then enables their symbolic execution.

3.1 Functional Finite State Machines

To represent variables, their values and operations on them, the notion of a domain is used. Definition 1 introduces domains, which are similar to quantifier-free first-order logic structures with functions and relations [32].

Definition 1. A *domain* is a tuple $D = (U, F, R, X)$, where:

- U is called *universe* and is a possibly infinite set of elements,
- F is a set of *functions* and each function $f \in F$ has an arity $n \in \mathbb{N}$ and is defined as $f : U^n \to U$. Given $x_1, ..., x_n \in U$, $f(x_1, ..., x_n)$ denotes an element of U represented by a function f. If the arity of a function is 0, then the function represents a *constant*,
- R is a set of *relations*: a relation $r \in R$ has an arity $n \in \mathbb{N}$ and is $r \subseteq U^n$,
- X is a set of *variables*.

We define *terms* and *formulas* in the usual manner. Terms are variables, constants or functions applied to other terms. We use $\Sigma(D)$ to denote a set of all terms over a domain D. Formulas are either relations over terms or are composed with the standard boolean connectives $\neg, \vee, \wedge, \Rightarrow$. The set of all formulas over a domain D is denoted with $\Phi(D)$. We also use an operation *vars* : $(\Sigma(D) \cup \Phi(D)) \to \mathbb{P}(X)$ ($\mathbb{P}$ is the powerset operator), which maps a term or a formula to the set of variables that it contains. Interpretations of terms and formulas are similar to those in first-order logic: for a given valuation of variables, a term denotes an element of a universe and a formula denotes either *true* or *false*. Additionally we introduce predicates over multiple domains, which use standard boolean connectives to compose formulas. For a set of domains $\{D_1, ..., D_N\}$, $\Phi(\{D1, ..., D_N\})$ represents the set of all predicates composed from formulas over $D_1, ..., D_N$.

Variables have a valuation that maps them to elements in their universe. Definition 2 presents how a valuation is extended to a set of variables from multiple domains.

Definition 2. Let $D_1, ..., D_N$ be domains with universes $U_1, ..., U_N$ and for a variable Y let $U(Y)$ denote its universe. A *valuation* of a set of variables $X = \{X_1, ..., X_M\}$ is a function $v : X \to \bigcup_{i=1,...,N} U_i$, such that $\forall X_i \in X : v(X_i) \in U(X_i)$. The set of all possible valuations of variables in X is denoted as $Val[X]$.

Definition 3 of functional finite state machines (FFSMs) uses variables and labels transitions with functions over their valuations.

Definition 3. Let $Doms = \{D_1, ..., D_N\}$ be a set of domains. A *functional finite state machine* is a tuple $F = (L, V, AV, A, GF, UF, OF, T, l_0, v_0)$, where:

- $L = \{l_1, ..., l_k\}$ is a finite set of *locations* with $l_0 \in L$ being the *initial location*,
- $V = \{v_1, ..., v_l\}$ is a finite set of *machine variables*. The function $D : V \to Doms$ assigns a type (domain) to a machine variable,

- AV is a finite set of *action variables* with type assignments $D : AV \to Doms$. Action variables are partitioned into sets of *input* and *output action variables* $AV_I = \{ai_1, ..., ai_n\}$ and $AV_O = \{ao_1, ..., ao_m\}$ such that $AV_I \cap AV_O = \emptyset$ and $AV_I \cup AV_O = AV$,

- A is a finite set of *actions* partitioned into *input* and *output actions* A_I and A_O. Each action is mapped to a set of action variables $vars : A \to \mathbb{P}(AV)$ such that $\forall a \in A : a \in A_I \Rightarrow vars(a) \subseteq AV_I \wedge a \in A_O \Rightarrow vars(a) \subseteq AV_O$,

- GF is a set of *guard functions*: $gf \in GF$ is a function $gf : Val[V] \times Val[AV'_I] \to \mathbb{B}$, where $AV'_I \subseteq AV_I$ and $\mathbb{B} = \{true, false\}$. So given valuations of machine variables (v) and of input action variables (v_i), $gf(v, v_i)$ is either *true* or *false*. For a function gf, $vars_all(gf)$ denotes the set of variables that its valuations range over, i.e., $vars_all(gf) = V \cup AV'_I$,

- UF is a set of *update functions*: $uf \in UF$ is $uf : Val[V] \times Val[AV'_I] \to Val[V]$, where $AV'_I \subseteq AV_I$. So given valuations of machine variables (v) and input variables (v_i), $uf(v, v_i)$ is a new valuation of machine variables. The operation $vars_all(uf)$ is defined as for guard functions,

- OF is a set *output functions*: $of \in OF$ is $of : Val[V] \times Val[AV'_I] \to Seq(A_O \times Val[AV'_O])$, where $AV'_I \subseteq AV_I$, $AV'_O \subseteq AV_O$ and $Seq(A_O \times Val[AV'_O])$ is the set of sequences of pairs that contain an output action and a valuation of the output variables used in that action, i.e., each element of a sequence is of the form (a, v) with $a \in A_O$ and $v \in Val[vars(a)]$. The operation $vars_all(of)$ is defined as for guard functions,

- T is a *transition relation*: $T \subseteq (L \times A_I \times GF \times UF \times OF \times L)$. We require that for a transition $t = (l, ai, gf, uf, of, l')$ (denoted also as $l \xrightarrow{ai, gf, uf, of} l'$) all functions $f \in \{gf, uf, of\}$ are such that $vars_all(f) = (V \cup vars(ai))$, i.e., they are defined only for machine variables and variables used in the input action ai,

- v_0 is an initial valuation of machine variables, i.e., $v_0 \in Val[V]$.

The set-theoretic definition of guard, update and output functions is not well suited for symbolic execution, because the tuples contained in the functions are explicitly enumerated, which makes such representation very verbose. Below, functions are defined in a more succinct way with logical terms and predicates over the set of domains $Doms$. In update and output functions a notion of cases is introduced to allow for attaching conditions to execution paths of action code (see Section 4).

Each guard function is defined as a predicate over domains of all variables:

$$gf \stackrel{def}{=} \psi \text{ such that: } \psi \in \Phi(Doms) \wedge vars(\psi) \subseteq (vars_all(gf))$$

Update functions are defined as a set of cases. Each case is like a conditional assignment and is represented as a pair consisting of a case condition (a predicate) and a function that assigns to each machine variable a term denoting its new value:

$$uf \stackrel{def}{=} \{uf_1, ..., uf_n\}$$
$$\wedge\ \forall uf_k \in uf : uf_k \in (\Phi(Doms) \times (V \to \bigcup_{v \in V} \Sigma(D(v))))$$

For each $uf_k = (\psi, assign)$, the variables used in terms and predicates are a subset of $vars_all(uf)$. We will use the projections $case(uf_k) = \psi$ and $assign(uf_k) = assign$. In Figure 2 `no_update`, `updateV1` and `updateV2` are examples of update functions.

The definition of output functions also uses cases, but the second element of each pair is a list of output actions with a valuation of output variables:

$$of \stackrel{def}{=} \{of_1, ..., of_m\}$$
$$\wedge \forall of_k \in of : of_k \in (\Phi(Doms) \times Seq(A_O \times (AssignOut))),$$

where $assignOut \in AssignOut$ if $assignOut : vars(ao) \to \bigcup_{v \in vars(ao)} \Sigma(D(v))$ is a function that for a given output action $ao \in A_O$ assigns terms to its output variables.
For each $of_k = (\psi, seq_o)$, the variables used in terms and predicates are a subset of $vars_all(of)$. We will use the projections $case(of_k) = \psi$ and $seq(of_k) = seq_o$. There are examples of output functions in Figure 2: `no_output`, `output_V` and `output_input`.

The execution of FFSMs relies on the evaluation of functions. The evaluation is performed using valuations $v \in Val[V]$ and $iv \in Val[AV'_I]$, where V are machine variables of an FFSM and AV'_I is a subset of its input variables. In order to use these valuations we define a substitution operation [ ]. For a term (or a predicate) p we use $p[v, iv]$ to denote an element of a universe (or a boolean value). The element (or the value) is obtained by replacing all variables in p with their values given by valuations v and iv. The *concrete evaluation* of functions is defined as follows:

1. Guard functions: their evaluation returns *true* or *false* and for *gf* defined with a predicate ψ we have:

$$eval[gf](v, iv) = \psi[v, iv]$$

2. Update functions: the evaluation returns a valuation of machine variables V. A new valuation is returned if at least one of the cases evaluates to *true*. Suppose $uf_k \in uf$ with $case(uf_k) = \phi$ and $update(uf_k) = u$, then we have:

$$eval[uf](v, iv) = v' \in Val[V] \text{ iff}$$
$$\exists uf_k \in uf : \phi[v, iv] \wedge \forall x \in V : v'(x) = u(x)[v, iv]$$

3. Output functions: the evaluation results in a sequence of output actions, which occurs only if one of cases evaluates to *true*. A sequence of output actions is generated with values of output variables given by the valuations v and iv. For a case of_k in an output function of, let $case(of_k) = \psi$ and $seq(of_k) = (o_1, v_1)(o_2, v_2)...$, then:

$$eval[of](v, iv) = s \text{ iff}$$
$$\exists of_k \in of : \phi[v, iv] \wedge s = (o_1, v_1[v, iv])(o_2, v_2[v, iv])...$$

Evaluation results of update and output functions may not be uniquely defined if more than one case evaluates to *true*.

The concrete execution semantics of an FFSM is defined using labeled transition system (LTS). In such an LTS each state represents a location and a valuation of machine variables. Transitions between states contain input actions and sequences of output actions along with values for their respective variables.

Definition 4. Let $F = (L, V, AV, A, GF, UF, OF, T, l_0, v_0)$ be an FFSM. Its *concrete execution* semantics is a tuple $\mathcal{E}(F) = (S, Tr, s_0)$, where:

- S is a set of execution states. Each state is a pair (l, v) where $l \in L$ is a location and $v \in Val[V]$ is a valuation of machine variables;
- Tr is a transition relation $Tr \subseteq (S \times A_I \times Val[AV'_I] \times Seq(A_O \times Val[AV'_o]) \times S)$, where $AV'_I \subseteq AV_I$ and $AV'_O \subseteq AV_O$. The transition relation is generated by the following rule:

$$\frac{l \xrightarrow{i,gf,uf,of} l', iv \in Val[vars(i)], \quad eval[gf](v, iv), v' = eval[uf](v, iv), seq = eval[of](v, iv)}{(l, v) \xrightarrow{i,iv,seq} (l', v')}$$

The set of all paths is defined as $paths(\mathcal{E}(F)) = \{s_0 t_0 s_1 ... | \forall k \geq 0 : t_k = (s_k, i, iv, seq, s_{k+1}) \wedge t_k \in Tr\}$.

Example 1. An example for an FFSM is presented in Figure 2 for a single domain of integers. The FFSM has locations $L = \{$`L0`,`L1`,`L2`,`L3`$\}$, machine variables $V = \{$`v1`,`v2`$\}$, action variables for input $AV_I = \{$`i1`,`i2`$\}$ and for output $AV_O = \{$`o1`$\}$. Actions are $A_I = \{$`Input1, Input2, Input3`$\}$ and $A_O = \{$`out`$\}$. The initial location is `L0` and initial valuation is $\{$`v1` $= 0,$ `v2` $= 10\}$. Figure 4 presents the initial states of the concrete execution.

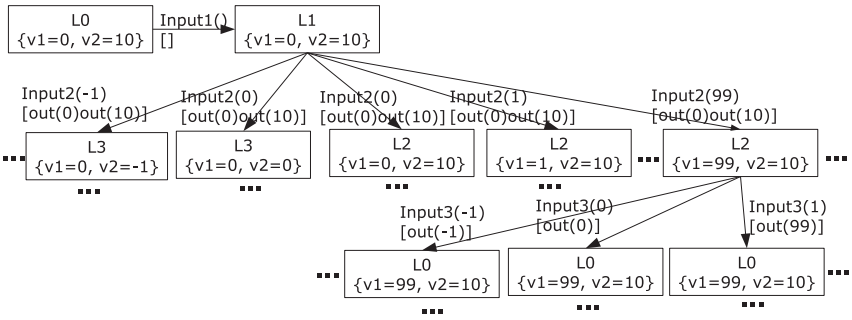

Figure 4: An initial part of concrete execution of the FFSM in Figure 2.

3.2 Symbolic execution of FFSMs

Symbolic execution of FFSMs, as outlined in Section 2, follows the general approach of symbolic execution of programs [21]. This means that values of variables are represented as terms over domains of the variables. Definition 5 introduces a notion of symbolic valuation for a set of variables.

Definition 5. For a set of variables X a *symbolic valuation* v^s is a function that maps a variable to a term from its domain, that is $v^s : X \to \bigcup_{v \in X} \Sigma(D(v))$ such that $\forall v \in X : v^s(v) \in \Sigma(D(v))$. The set of all possible symbolic valuations of variables in X is denoted as $Val^s[X]$.

The central element of the symbolic execution of FFSMs is a symbolic evaluation of guard, update and output functions. The evaluation is conducted for a given symbolic valuation of machine variables $v^s \in Val^s[V]$, path constraints pc and a symbolic valuation of input variables $iv^s \in Val^s[AV_I']$, where $AV_I' \subseteq AV_I$. During evaluation, variables are replaced by their values, which are symbolic and represented with terms. To achieve this, a replacement operator for terms and predicates is introduced. If p is a term (or a formula or a predicate) then $p[v^s, iv^s]$ denotes a term (or a formula or a predicate) with all variables in p substituted by their mapped symbolic values in v^s or iv^s. Additionally, evaluation of each function may update path constraints. This happens if a guard or a case condition cannot be inferred from the current path constraints and must be assumed. *Symbolic evaluation* of functions is given by ($pc \Rightarrow \phi$ abbreviates $((\bigwedge_{\psi \in pc} \psi) \Rightarrow \phi)$):

1. Guard functions: evaluation may result in an additional path constraint.

$$eval^s[gf](v^s, iv^s, pc) = \begin{cases} \emptyset, & \text{if } pc \Rightarrow gf[v^s, iv^s] \\ \{gf[v^s, iv^s]\}, & \text{otherwise.} \end{cases}$$

2. Update functions: evaluation results in a pair containing possibly a new path constraint and a new symbolic valuation. The evaluation is performed for each case and for the kth case $uf_k = (\phi, asgn)$ (ϕ is a case constraint and $asgn \in Val^s[V]$ is a symbolic valuation) we have:

$$eval^s[uf_k](v^s, iv^s, pc) = \begin{cases} (\emptyset, asgn[v^s, iv^s]), & \text{if } pc \Rightarrow \phi[v^s, iv^s] \\ (\phi[v^s, iv^s], asgn[v^s, iv^s]), & \text{otherwise.} \end{cases}$$

   where $asgn[v^s, iv^s]$ is a valuation such that $\forall v \in V : asgn[v^s, iv^s](v) = asgn(v)[v^s, iv^s]$.

3. Output functions: evaluation may result in a pair with a sequence of output actions as the second element. Evaluation is performed for each case. For the kth case $of_k = (\phi, (oa1, asgn1)(oa2, asgn2)...)$ we have:

Table 1: An example of symbolic evaluation of the functions from Figure 2.

Arguments:	$v^s = \{(v1 \leftarrow i1_1), (v2 \leftarrow 10)\}$, $iv_1^s = \{(i1 \leftarrow i1_2)\}$, $iv_2^s = \{(i2 \leftarrow i2_1)\}$, $pc = \{i1_1 \geq 0\}$
guard function: (i1 $\geq$ 0)	$eval^s[(\text{i1} \geq 0)](v^s, iv_1^s, pc) = \{(i1_2 \geq 0)\}$
update func.: `updateV1` $\stackrel{def}{=}$ {(true, (v1:=i1, v2:=v2))}	$eval^s[\text{updateV1}_1](v^s, iv_1^s, pc) =$ $(\emptyset, \{(v1 \leftarrow i1_2), (v2 \leftarrow 10)\})$
output func.: `output_inp` $\stackrel{def}{=}$ {((i2 > 0), [(out, o1:=v1)]), ((i2 $\leq$ 0), [(out, o1:=i2)])}	$eval^s[\text{output_inp}_1](v^s, iv_2^s, pc) =$ $(\{i2_1 > 0\}, [(out, (o1 \leftarrow i1_1))])$, $eval^s[\text{output_inp}_2](v^s, iv_2^s, pc) =$ $(\{i2_1 \leq 0\}, [(out, (o1 \leftarrow i2_1))])$

$$eval^s[of_k](v^s, iv^s, pc) = \begin{cases} (\emptyset, (oa1, asgn1[v^s, iv^s])...), & \text{if } pc \Rightarrow \phi[v^s, iv^s] \\ (\phi[v^s, iv^s], (oa1, asgn1[v^s, iv^s])...), & \text{otherwise.} \end{cases}$$

Example 2. Table 1 presents the symbolic evaluation of some of the functions in Figure 2. For instance, an evaluation of the guard function $(i1 \geq 0)$ results in a constraint $(i1_2 \geq 0)$ by simply replacing $i1$ with $i1_2$ based on a mapping iv_1^s. The output function `output_input` is evaluated using the mapping iv_2^s and the evaluation proceeds for both cases.

Definition 6. A *symbolic execution tree* for a given FFSM $F = (L, V, AV, A, GF, UF, OF, T, l_0, v_0)$ is a tuple $\mathcal{SE}(F) = (S^s, AV_I^s, InVars^s, Tr^s, s_0^s)$, where:

- AV_I^s is a set of input variables different from all other variables in F,
- $InVars^s$ is a set of mappings $iv^s : AV_I' \rightarrow AV_I^s$ with $AV_I' \subseteq AV_I$ that assign to an input variable a new variable that represents its symbolic value. Each mapping is therefore a special kind of a symbolic valuation for an arbitrary subset of input variables,
- S^s is a set of symbolic states $s = (l, v^s, pc)$, where $l \in L$ is a location, $v^s \in Val^s[V]$ is a symbolic valuation of machine variables and pc is a set of path constraints that are predicates, i.e., $pc \subseteq \Phi(Doms)$. For a state $s = (l, v, pc)$ we define projections $val(s) = v$ and $pc(s) = pc$. The initial symbolic state $s_0^s \in S^s$ is $(l_0, v_0, \emptyset)$,
- Tr^s is a relation $Tr^s \subseteq (S^s \times A_I \times InVars^s \times Seq(A_O \times Val^s[AV_O']) \times S^s)$, where $AV_O' \subseteq AV_O$. For a transition $t = (s, ai, iv, seq, s')$ we denote its source and target states with $sr(t) = s$ and $tg(t) = s'$ and its sequence of output actions as $seq(t) = seq$. The transition relation is generated by the following rule:

$$\frac{\begin{array}{l} l \stackrel{ai, gf, uf, of}{\longrightarrow} l', iv^s \in InVars, pc_1 = eval^s[gf](v^s, iv^s, pc), \\ \exists uf_k \in uf : (pc_2, v') = eval^s[uf_k](v^s, iv^s, pc), \\ \exists of_k \in of : (pc_3, seq) = eval^s[of_k](v^s, iv^s, pc), \\ pc' = pc[iv^s, v^s] \cup pc_1 \cup pc_2 \cup pc_3, pc' \text{ satisfiable} \end{array}}{(l, v^s, pc) \stackrel{ai, iv^s, seq}{\longrightarrow} (l', v^{s\prime}, pc')}$$

The set of all symbolic paths is defined as $paths^s(\mathcal{SE}(F)) = \{s_0^s t_0^s s_1^s ... | \forall k \geq 0 : t_k^s = (s_k^s, i, iv^s, seq, s_{k+1}^s) \wedge t_k^s \in Tr^s\}$

A symbolic execution tree for an FFSM represents its all possible concrete executions, which follows from the theorem below. In the theorem the relation $\models$ between paths is satisfied if values used in a concrete path satisfy all path constrains in a symbolic path and paths agree as for locations and actions.

Algorithm 1 `FFSM2SET` - symbolic execution of an FFSM

Require: $F = (L, V, AV, A, GF, UF, OF, T, l_0, v_0)$
Ensure: $\mathcal{SE}(F) = (S^s, AV_I^s, InVars^s, Tr^s, s_0^s)$
  $to_process \leftarrow < s_0^s >$
  while $to_process \neq \emptyset$ **do**
3: $s = (l, sv, pc) \leftarrow$ removed first element from $to_process$
    for all outgoing transitions $t = (l, ai, gf, uf, of, l')$ from l **do**
      $iv^s \leftarrow$ new mapping of variables in $vars(ai)$
6: $pc' \leftarrow pc \cup eval^s[gf](v^s, iv^s, pc)$
      for all $uf_k \in uf$ **do**
        $(pc'', v^{s\prime}) \leftarrow eval^s[uf_k](v^s, iv^s, pc')$
9: **for all** $of_k \in of$ **do**
          $(pc''', seq) \leftarrow eval^s[of_k](v^s, iv^s, pc')$
          $pc' \leftarrow pc' \cup pc'' \cup pc'''$
12: **if** (pc' is satisfiable) **then**
            $s' \leftarrow (l', v^{s\prime}, pc')$
            $S^s \leftarrow S^s \cup \{s'\}$
15: $Tr^s \leftarrow Tr^s \cup \{(s, ai, iv^s, seq, s')\}$
            $InVars^s \leftarrow InVars^s \cup iv^s$
            if $\neg(\exists s'' \in S^s : s'' \neq s' \wedge s' \prec s'')$ **then**
18: add $(l', v^{s\prime}, pc')$ at the end of $to_process$
  return $\mathcal{SE}(F)$

Theorem 1. Let F be an FFSM. For its execution semantics $\mathcal{E}(F)$ and symbolic execution tree $\mathcal{SE}(F)$ we have

$$\forall p \in paths(\mathcal{E}(F)) : \exists p^s \in paths^s(\mathcal{SE}(F)) : p \models p^s$$

(sketch). The proof is inductive over length of paths. For paths with the length 0 we note that initial states are the same in a symbolic and concrete execution. Now let assume $p \models p^s$ for paths of length n. For a concrete path p a transition is added according to the rule in Definition 4. The rule requires that there is a transition in which a guard function is satisfied and there is a case in an update and in an output function that are satisfied. The satisfaction of guard and cases implies that a transition is also generated for a symbolic path p^s. The transitions agree, thus the concrete path with length $n + 1$ satisfies a symbolic path of the same length. □

Algorithm 1 shows how to construct a symbolic execution tree from an FFSM. The algorithm explores symbolic states in a breadth-first search manner and maintains states waiting for exploration in a queue *to_process*. During the exploration of a symbolic state, for each outgoing transition from the state's location the guard function and each case of update and output functions are symbolically evaluated. If generated path constraints do not create a contradiction, then a new symbolic state is added to the tree along with a symbolic transition. If this new state is not subsumed (denoted as $\prec$ and explained below) by one of the previous states, then it is stored in *to_process*. The algorithm terminates if there are no more states to explore.

The presented method uses the subsumption relation. For two symbolic states $s' = (l', sv', pc')$ and $s'' = (l'', sv'', pc'')$, we say that s'' *subsumes* s' (denoted as $s' \prec s''$) iff the states have the same location ($l' = l''$), the same symbolic valuations $sv' =_{iv} sv''$ and the path constraints pc'' of s'' are weaker than the path constraints pc' of s'. The subsumption relation also relates states that are different only up to the replacement of the last input symbolic variable. The rationale behind this relation is that if there is a state s'' with weaker path constraints pc'' and with the same location and values of machine variables, it represents in fact more states with concrete values than s' does. This means that there is no point to further explore s'. Note that Algorithm 1 may not terminate, if there are loops with updates based on the previous values of machine variables to machine variables. To overcome this, each transition is symbolically executed only M times, where M is some user-defined bound.

Example 3. Figure 3 presents a part of a symbolic execution tree for the FFSM given in Example 1. Comparing the tree in Figure 3 to the concrete execution shown in Figure 4 we see that in the symbolic execution there are two symbolic states for the first occurrence of the location $L2$. In the

Table 2: The mapping of UML-RT elements to elements of an FFSM.

UML-RT State Machine element	**FFSM element**
non-composite states (with parent states)	locations L
attributes of a capsule	machine variables V
input events (signals) for base ports, output events (signals) for conjugated	input actions A_I
output events (signals) for base ports, input events (signals) for conjugated	output actions A_O
parameters of input and output events (signals)	action variables AV
transition chains between two states	transitions T
default values of attributes	initial valuation of machine var. v_0
initial pseudostate of a top level state	initial location l_0

concrete execution the number of states with the location $L2$ is given by the number of possible values of the input variable $i1$ and thus it is potentially infinite.

4 UML-RT State Machines as FFSMs and their analysis

The UML-RT language is a profile of UML 2 [5] and is used for modeling embedded and real time systems. The language is supported by the IBM RSA RTE tool (a successor of the IBM Rational@RoseRT tool) [3]. UML-RT models consist of a hierarchy of *capsules* connected with typed *ports* [31]. A capsule, as its name suggests, is a highly encapsulated entity, which communicates with other capsules only by sending and receiving signals through its ports. Each port has a type specified with a *protocol*, which gathers sent or received signals. Connected ports must implement the same protocol and one port in such a connection is declared to be a *base* one, whereas the second one is a *conjugated* one, in which send and receive signals of its protocol are inversed.

Capsules have their behavior specified with UML-RT State Machines [31], which are similar to UML State Machines [5]. A UML-RT State Machine contains hierarchical states and guarded transitions, which are triggered by signals received on ports. As the standard UML State Machines, UML-RT State Machines use a strict run-to-completion semantics of event-handling. The main difference between UML and UML-RT State Machines is that UML-RT State Machines do not have "and-states", i.e., no orthogonal regions. Hence all states are "or-states". Moreover, UML-RT State Machines deal with sending events and the initializations of timers in their action code. Figure 5 presents an example of UML-RT State Machine.

4.1 Representing structure

In order to represent structural elements of UML-RT State Machines, these elements are mapped to elements of FFSMs. This mapping is summarized in Table 2.

States in UML-RT State Machines might be hierarchical, whereas FFSMs do not support such hierarchy. Thus to be able to process hierarchical UML-RT State Machines, locations of FFSMs are assumed to contain the information about a state and all its parent states. In this way it is possible to discover transitions of a given state and its parent states (if any). The exploration of states and transitions is part of the on-the-fly algorithm introduced in Section 4.3. Hence explicit flattening of UML-RT State Machines is avoided and a flat UML-RT State Machine is not required as the input to symbolic execution.

In the presented mapping of UML-RT State Machines not all available features are considered. For instance, we omit history states or internal transitions. Although these are important modeling elements, their treatment is not essential in the presented approach. To include them only the state exploration part of the algorithm has to be adjusted, which does not influence the symbolic execution. The support for these elements is left for future work.

4.2 Representing action code

Action code from UML-RT State Machines is represented in FFSMs as functions. Such code is contained in entry or exit actions of states and as actions and guards on transitions. It is gathered during the discovery of transition chains (see Algorithm 2) in the order imposed by the run-to-completion semantics of UML-RT. To generate functions, the gathered code is executed symbolically [21] with the sending of events also included in the resulting symbolic execution tree. The results of a symbolic execution of action code are translated to functions in FFSMs.

Definition 7. Let assume that a piece of action code c has been executed symbolically and has produced a symbolic execution tree. Leaves of this tree are tuples $(pc_c, sv_c, ret_c, seq_c)$, where pc_c represents the path constraints, sv_c the symbolic values of machine variables (required by update functions), ret_c the returned boolean expression (required in guard functions) and seq_c the possibly empty sequence of output actions (required in output functions) with symbolic values of output variables. Let the set $SE(c)$ contain all leaves of the tree. Then c is represented by the functions:

- guard function:
  $gf \stackrel{def}{=} \bigwedge_{(pc_c, sv_c, ret_c, seq_c) \in SE(c)} ((\bigwedge_{p \in pc_c} p) \Rightarrow ret_c)$,
- update function:

$$uf \stackrel{def}{=} \{uf_k | \exists (pc_c, sv_c, ret_c, seq_c) \in SE(c) : (case(uf_k) = \bigwedge_{p \in pc_c} p) \ \wedge \ (update(uf_k) = sv_c)\},$$

- output function:

$$of \stackrel{def}{=} \{of_k | \exists (pc_c, sv_c, ret_c, seq_c) \in SE(c) : (case(of_k) = \bigwedge_{p \in pc_c} p) \wedge (seq(of_k) = seq_c)\}.$$

If c does not return any ret_c then the guard function is trivial. Similarly, if c has no updates or outputs then update or output functions are trival.

The exact specification how to symbolically execute action code depends on the used action language. In the next section we use C++, however the above method can be applied also to other languages.

4.3 Constructing FFSMs

The construction of an FFSM from a given UML-RT State Machine is performed in two stages. First, the static elements of the FFSM are created according to Table 2. These elements are all variables, input and output actions, an initial location and valuation. The second stage is performed on-the-fly during symbolic execution of the FFSM in Algorithm 1. The exploration that is added after line 18 of Algorithm 1 and is outlined in Algorithm 2. The overall process has been presented in Figure 1.

The intent of the above translation is that the resulting FFSM captures the semantics of the UML-RT State Machines. A formal proof of this property is outside the scope of the paper and is omitted due to space limitations. For the same reason a formal definition of the translation of UML-RT State Machines into FFSMs is not presented.

Algorithms 1 and 2 are designed to analyze UML-RT State Machines. However, it can be adapted to analyze the general UML State Machines [5] or any subset of those. For instance, to deal with "and-states" in UML 2 a mapping of a location to a configuration of states in a state machine is necessary. Additionally, symbolic execution of code needs to be adjusted. These enhancements are left as the future work.

Algorithm 2 `UMLRT2FFSM` - constructing an FFSM from a UML-RT State Machine

Require: UML-RT State Machine SM,
Require: FFSM $F = (L, V, AV, A, GF, UF, OF, T, l_0, v_0)$,
Require: location l'
  if l' not explored **then**
    for all transition chains tc in SM starting in l' **do**
      action_code ← gather exit, entry actions, guards, effects of transitions in tc
      SE(action_code) ← execute symbolically action_code
      generate functions gf, uf, of from SE(action_code)
      add target state l'' of tc to L,
      add (l',trigger of tc, gf, uf, of, l'') to T
      mark l' as explored

4.4 Analyses of UML-RT State Machines

A symbolic execution tree generated from an FFSM, which is translated from a UML-RT State Machine can be used to perform several analyses. This is possible, because the tree represents all possible executions of the machine up to the subsumption relation. Therefore, we can verify a variety of properties of UML-RT State Machines. The most important ones are introduced below.

A symbolic execution tree can be used to perform *reachability analysis* between states in the original UML-RT State Machines, which are represented with locations in the FFSM. The location s is reachable from the location s' if there is a path (which may contain subsumed states) in the symbolic execution tree, which starts in a symbolic state with the location s and ends in a symbolic state with the location s'.

Another useful analysis is *invariant checking*, which allows checking whether some conditions are satisfied in all states of execution of a UML-RT State Machine. Conditions are defined as predicates over attributes (i.e. machine variables in the FFSM) and input variables of the UML-RT State Machine. A predicate with those variables is an invariant if it can be inferred from path constraints in all symbolic states of the symbolic execution tree.

A symbolic execution tree can be also used for *output analysis*, that is, to check whether an output action is a part of any output sequence. In this way it is verified whether an event in a UML-RT State Machine is generated as output during its execution. If this event is represented with an output action a_o we check whether the FFSM has a transition in which a_o occurs in a sequence of output actions. If so, a path to such a transition is returned.

If a symbolic execution tree is deterministic it can be used to *generate test cases*. The tree is deterministic if two outgoing transitions with the same input action cannot be both taken for the same values of input variables, thus the path constraints in the resulting symbolic states do not have a solution in common. A set of test cases that provides path coverage of a UML-RT State Machine is defined as the traces to the leaves of a symbolic execution tree. A trace is a sequence of input and output actions gathered from the transitions that lead to some leaf. In order to be useful, such traces must be solved, which means that input variables in each trace receive values that do not contradict the path constraints.

The presented analyses are just examples of how symbolic execution trees can be used to verify some properties of UML-RT State Machines. They were chosen, because our industrial partners expressed interest in them. Nevertheless, other possible types of analysis as well as extensions of the above methods are possible.

5 Implementation and evaluation

5.1 Implementation

The implementation of the symbolic execution of UML-RT State Machines as well as the analysis consist of several Eclipse plugins in IBM RSA RTE [3]. The implementation follows the general approach as presented in Figure 1.

During the implementation several assumptions were made. Firstly, the symbolic execution of code is performed for a subset of C++ containing basic operations on variables, if statements and

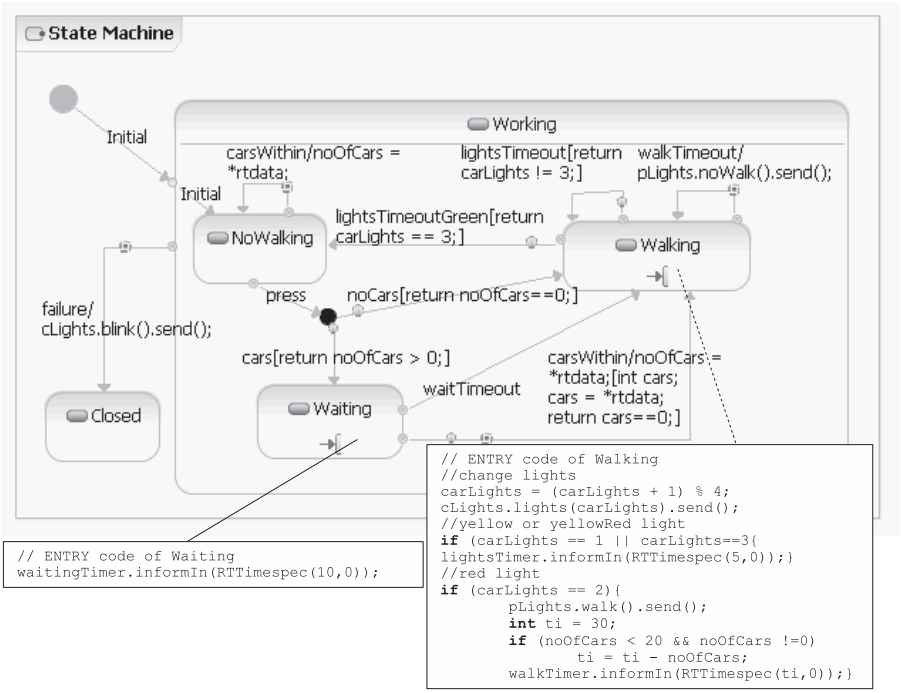

Figure 5: An example of UML-RT state machine (transitions are of the form `action/action code[guard code]`).

while loops (their symbolic execution is bounded). The symbolic execution of full C or C++ is not trivial and is not of interest here. We justify this simplification by the observation that action code used in UML-RT State Machines is typically not very complex. Secondly, variables are assumed to have numeric types (i.e., integer and reals). This is the consequence of using the constraint solver Choco [1] for constraint representation and for checking their satisfiability in Algorithm 1. Thirdly, constraints are compared syntactically (symbol by symbol). Therefore the subsumption relation $\prec$ is quite restrictive and does not take into account the meaning of constraints. Finally, we treat setting timers and timeout events as ordinary actions, but we require that a timeout event happens only if the corresponding timer has been set.

5.2 Example

To present some of the possible analysis results, the example of the UML-RT State Machine shown in Figure 5 is used. This machine is a part of a UML-RT model of traffic lights. The system controls pedestrian and car lights at a street crossing. It is equipped with buttons and with sensors that periodically report a number of cars approaching the crossing. The shown UML-RT State Machine specifies the behavior of the controller. There are two attributes `noOfCars` and `carLights`. The latter stores the encoding of the current color of lights for cars (0 is green, 1 is yellow, 2 is red, 3 is yellow-red). Both attributes are initialized to zero. In the presented code the pointer `*rtdata` represents input values, timers are set using the `informIn()` method and sending events uses the `send()` method.

The result of the symbolic execution of UML-RT State Machine in Figure 5 is a symbolic execution tree, which is shown partially in Figure 6. The whole tree has 98 states, which is the consequence of combining possible valuations of the attribute `carLights` and locations, as well as additional branching conditions based on the attribute `noOfCars`. The tree represents the possible executions of the given UML-RT State Machine with each symbolic state aggregating possibly

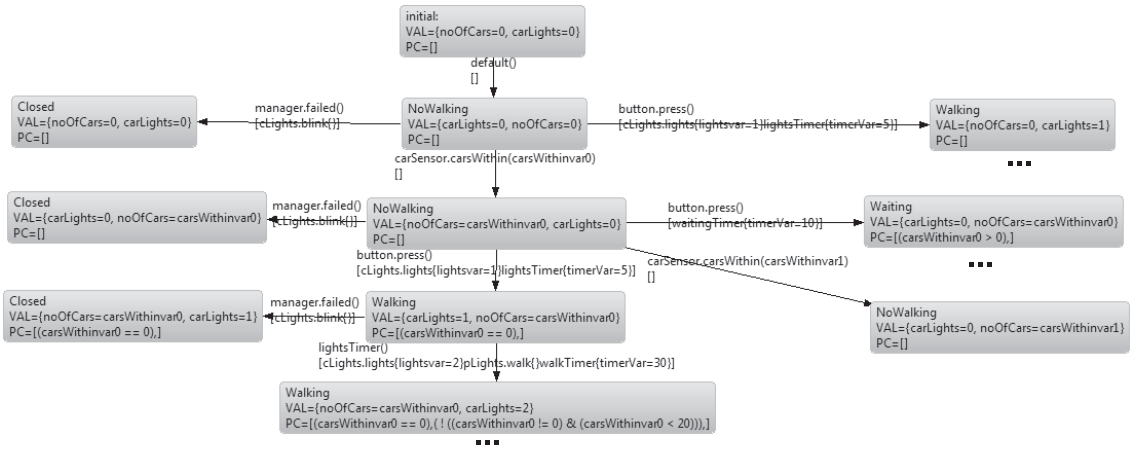

Figure 6: A part of a symbolic execution tree for the example UML-RT State Machine in Figure 5.

infinitely many concrete states.

In the implementation the computed tree is used to perform analyses as introduced in Section 4.4:

1. Reachability analysis. For instance, checking for paths from `Walking` to `NoWalking` returns a path (i.e., a sequence of symbolic states and transitions that connect them, the latter are enclosed in "<>") :

   `(Walking, (carLights=3, noOfCars=0), PC=[])`

   `<lightsTimeout(), [] >`

   `(NoWalking, (carLights=3,noOfCars=0), PC=[])`

   The last symbolic state reveals a fault, because the value of `carLights` is not properly reset to 0 (green lights for cars) and the `lights` signal with this value is not sent (the fault is in the transition between `Walking` and `NoWalking` in Figure 5).

2. Invariant checking. For example, when checking the invariant (`noOfCars` >= 0) there is a violation found for a simple path with only a single occurrence of the signal `carsWithin`. This is because the input value is not checked before the assignment to `noOfCars`, and the attribute takes on a negative value (this fault is in the action code of the self-loop transition in `NoWalking` state in the UML-RT State Machine in Figure 5).

3. Output analysis. For example, if we consider the signal `walkTimer.informIn()` the following traces are possible:

   - `<carsWithin(var0)[] > ...`
     `<lightsTimeout()[...,walkTimer(timerVar=(30-var0))] >`
     with `PC=[(var0 > 0), (var0 < 20)]`,
   - `<carsWithin(var0)[] > ...`
     `<lightsTimeout()[...,walkTimer(timerVar=30)] >`
     with `PC=[(var0 > 0), ¬(var0 < 20)]`,

   The traces show the values used to set the timer and their constraints.

4. Test case generation. The tree is deterministic and there are 50 test cases satisfying a path coverage criterion. One of them is:

   `<carsWithin(1)><press()[waitingTimer(10)]>`

   `<waitingTimeout()[lights(1),lightsTimer(5)]>`

   `<lightsTimeout()[lights(2),walk(), walkTimer(29)]>`

   `<failed()[blink()]>`

Table 3: Performance of generating symbolic execution trees (SET) and test cases (TC)

UML-RT State Machine	number of locations in FFSM	number of states in SET	time to generate SET (in seconds)	time to generate TC (in seconds)
Traffic lights (Figure 5)	5	98	2.9	0.3
Phone (simplified)	69	960	20.8	97.1
Phone	125	4069	51.0	362.4

5.3 Evaluation

In order to evaluate the presented approach we performed several experiments using different UML-RT State Machines. Table 3 presents the results of these experiments (performed on a standard PC with Intel Core i7 CPU 2.67 GHz, 3GB of RAM). In the table the first UML-RT State Machine is the one presented in Figure 5. The third UML-RT State Machine, which we obtained from one of our industrial partners, models interactions between a user and a phone. It has more than 100 states, which are nested up to the sixth level. In the simplified model (the second UML-RT State Machine in Table 3) some of the transitions are disabled using contradictory guards, so parts of the UML-RT State Machine are not reachable. The presented results include the test case generation time, which is the most complex analysis, because all paths in the tree must be considered.

As shown in Table 3 the sizes of the generated FFSMs differ for the last two cases. Although both UML-RT State Machines contain the same number of states and transitions, in the first model some parts are unreachable. Because Algorithm 2 generates the appropriate FFSM on-the-fly, the FFSM represents only a reachable part of the UML-RT State Machine. Therefore the number of locations in the FFSM is the same as the number of non-composite and reachable states in the respective UML-RT State Machine.

Times to generate SETs increase with the number of locations in FFSMs and with the number of states in SETs. Nonetheless, the increase is not proportional: on average it takes 0.03s to generate one symbolic state in the first model and only 0.01s in the third one. The difference between those two models is only in the used action code: the action code of the UML-RT State Machine shown in Figure 5 introduces more constraints than the action code in the phone model. With more constraints more queries are made to the constraint solver, which affects the performance. This effect manifest itself in the time required to generate test cases for the third state machie. There are more than 3000 paths in the SET and all these paths must have their constraints solved to become test cases and this process is very time consuming.

As presented, the prototype implementation performs the analysis of the subset of UML-RT State Machines. However the implementation has its limitations. One of them is inherited from the Choco solver [1], which cannot solve arbitrary complex constraints and is not very efficient. Nevertheless, for straightforward relations between variables, the performance is acceptable and the solver is sufficient. The other limitation is the result of the exhaustive exploration in Algorithm 1, which may mean that the implementation does not scale well to larger models. However, the presented technique, besides using symbolic values to aggregate states, is applied to a single state machine, so the analysis is performed on the unit level. As illustrated by the above examples, the technique can be applied to analyze non-trivial UML-RT State Machines. The general applicability of the method to industrial models is future work and will directly depend on the nature of the action code used in these models.

6 Related work

We partition our discussion of related work into three groups: 1) symbolic execution of programs, 2) symbolic execution of state-based models, and 3) formal analysis of state machines in UML and UML-RT.

1) Apart from the development of a dedicated tool for symbolic analysis, the use of a standard

model checker has also been suggested [20, 12]. A translation of UML-RT State Machines into a language supported by one these tools could be used to realize symbolic execution of UML-RT State Machines. While this approach would allow use of, e.g., the optimization and abstraction features of these model checkers, it would also complicate the support for different action languages. Approaches to make symbolic execution modular by creating summaries of computational units have also been developed (e.g., [6, 16]). Although the technical details differ, these summaries are similar in spirit to the functions used in FFSMs to capture the effect of action code.
2) Symbolic execution has been applied to different kinds of state-based models. For instance, Tretmans' testing theory [9], which formalizes circumstances under which an implementation (modeled as an input/output transition system, ioTS) can be considered conforming to a specification (modeled as a labeled transition system), has been extended to deal with infinite-state specifications using symbolic execution for test case generation [13] and selection [19]. The work in [15, 28] builds on this effort and presents approaches for test case generation for UML 1.x state machines by translating them into ioTS, and for implementing LTL model checking of ioTS by using symbolic execution. Test case generation for a timed extension of Harel's Statecharts is dealt with in [22] (the work in [8] does the same for Harel Statecharts, but without symbolic execution). In contrast to our work, none of these approaches allow for action code that is written in a standard programming language.
3) A deductive approach to verifying temporal properties state machines in UML 1.x is taken in [7, 33]. A simple, idealized action language containing assignment, sequencing, parallel composition is supported, however, the approach cannot be used for test case generation and is not fully automatic, since it is based on interactive theorem proving. The literature contains many proposals to model check state machines in UML-RT and UML. All approaches we are aware of translate the state machines into the input language of a model checker. Early work on translating UML-RT State Machines to Promela can be found in [29]. The approach in [30] deals with state machines in UML 1.4 in the same way, but also features an implementation that can check state machines with respect to collaboration diagrams (called communication diagrams in UML 2). The work is revised to deal with timing constraints in [25] using timed automata and Uppaal. In [10], Burmester et al also use a form of timed automata and Uppaal; however, the work also supports incremental, compositional development and refinement via the reasoning approach detailed in [14]. In [23], UML-RT State Machines are mapped to ASML; in contrast to other work on the topic, dynamic capsule creation, binding, and destruction via optional and imported capsules are supported as well; SpecExplorer is used for simulation of UML-RT models, but support for model checking and test case generation is left for future work. In [24], UML State Machines are translated to Java and analyzed using JPF. Unlike other works, the approach can deal directly with Java action code, but support for other action languages would require a translation to Java. A key difference between model checking and symbolic execution is that symbolic execution is capable of handling very large (possibly even infinite) state spaces.

A comprehensive, formal account of the semantics of UML-RT is still an open research problem. Since the primary goal of our work is to define and implement symbolic execution of UML-RT State Machines and not to present a full, formal definition of their semantics, we do not review this part of the literature. Finally, the model analysis capabilities of IBM RSA RTE and RoseRT are quite limited and do not cover the analyses we are interested in. Therefore, these tools also are not reviewed any further.

7 Conclusions and future work

This paper presents a method for the symbolic execution of UML-RT State Machines, able to handle different action languages. This is possible due to the modular treatment of action code via functions, which are constructed on-the-fly during the symbolic execution of state machines with the help of a symbolic execution engine for action code. The functions are incorporated into a formal representation of state machines, called functional finite state machines (FFSMs).

As shown, FFSMs can represent UML-RT State Machines, however translation from other state machines models is also possible. The symbolic execution of FFSMs is formally defined. To illustrate the method and the analysis, the paper presents an implementation which symbolically executes UML-RT State Machines with action code that is a subset of C++. The resulting symbolic execution tree is used to perform reachability analysis, invariant checking, output analysis and test case generation and two case studies are presented.

Apart from the future work already mentioned, more work is required to determine the applicability of the technique to industrial-size UML-RT models. If performance or functionality improvements become necessary some of the recently developed extensions of symbolic executions (e.g., with concrete execution) will be considered. Finally, work allowing for the symbolic analysis of collections of capsules by combining symbolic execution trees is currently ongoing [37].

8 Acknowledgements

Authors wish to acknowledge the support of NSERC, IBM Canada, and Malina Software.

References

[1] *Choco Solver http://www.emn.fr/x-info/choco-solver/doku.php?id=about.*

[2] *IBM Rational Rhapsody Architect, Version 7.5.3. http://www-01.ibm.com/software/ rational/products/rhapsody/swarchitect/.*

[3] *IBM Rational Software Architect, RealTime Edition, Version 7.5.5. http://publib.boulder.ibm.com/ infocenter/rsarthlp/v7r5m1/.*

[4] *Scade Suite (Esterel Technologies) http://www.esterel-technologies.com/products/scade-suite/.*

[5] Unified Modeling Language (UML 2.0) Superstructure. http://www.uml.org/.

[6] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS 2008 (LNCS 4963)*, pages 367–381, 2008.

[7] M. Balser, S. Baumler, A. Knapp, W. Reif, and A. Thums. Interactive verification of UML state machines. In *ICFEM 2004 (LNCS 3308)*, pages 434 – 48, 2004.

[8] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In *FM-Trends 1998 (LNCS 1641)*, pages 107 – 21, 1998.

[9] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *Summer School on Modeling and Verification of Parallel Processes (MOVEP'00)*, 2001.

[10] S. Burmester, H. Giese, M. Hirsch, and D. Schilling. Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In *SVERTS2004 (part of UML2004)*, 2004.

[11] P. Coward. Symbolic execution systems-a review. *Software Eng. J.*, 3(6):229 –239, 1988.

[12] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. ASE'06*, 2003.

[13] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In *FATES 2004 (LNCS 3395)*, pages 1 – 15, 2004.

[14] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. ESEC/FSE'03*, 2003.

[15] S. Gnesi, D. Latella, and M. Massink. Formal test-case generation for UML statecharts. In *ICECCS 2004*, pages 75–84, 2004.

[16] P. Godefroid. Compositional dynamic test generation. In *POPL '07*, 2007.

[17] P. Godefroid. From blackbox fuzzing to whitebox fuzzing towards verification. Invited talk at ISSTA'2010, 2010. http://research.microsoft.com/en-us/um/people/pg/.

[18] F. Ipate and M. Holcombe. Generating test sets from non-deterministic stream X-machines. *Formal Aspects of Computing*, 12(6):443 – 58, 2000.

[19] T. Jeron. Symbolic Model-based Test Selection. *ENTCS*, 240:167 – 184, 2009.

[20] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS'03*, 2003.

[21] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385 – 94, 1976.

[22] N. H. Lee and S. D. Cha. Generating test sequence using symbolic execution for event-driven real-time systems. *Microproc. and Microsys.*, 27(10):523 – 31, 2003.

[23] S. Leue, A. Stefanescu, and W. Wei. An AsmL Semantics for Dynamic Structures and Run Time Schedulability in UML-RT. Technical report, University of Konstanz, Germany, 2008.

[24] P. Mehlitz. Trust your model - verifying aerospace system models with Java pathfinder. In *IEEE Aerospace Conference*, 2008.

[25] S. Merz and C. Rauh. Model checking timed UML state machines and collaborations. In *Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, pages 395–414, 2002.

[26] OMG. Action Language for Foundational UML (Alf), 2010. ptc/2010-10-05.

[27] C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *J. on Software Tools for Technology Transfer (STTT)*, 11:339–353, 2009.

[28] N. Rapin. Symbolic execution based model checking of open systems with unbounded variables. In *Tests and Proofs*, volume 5668 of *LNCS*, pages 137–152, 2009.

[29] M. Saaltink and I. Meisels. Using SPIN to analyse RoseRT models. Technical report, ORA Canada, 1999.

[30] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *ENTCS*, 55(3):pp. 1–13, 2001.

[31] B. Selic. Using UML for modeling complex real time systems. In *ACM SIGPLAN Workshop LCTES 1998 (LNCS 1474)*, pages 250 – 60, 1998.

[32] V. Sperschneider and G. Antoniou. *Logic : a foundation for computer science.* Addison-Wesley, 1991.

[33] A. Thums, G. Schellhorn, F. Ortmeier, and W. Reif. Interactive Verification of Statecharts. In *INT 2004 (LNCS 3147)*, 2004.

[34] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA '07*, pages 97–107, 2007.

[35] T. Weigert, F. Weil, and K. Marth. Experiences in deploying model-driven engineering. In *SDL 2007: Design for Dependable Systems*, 2007.

[36] P. Whittaker, M. Goldsmith, K. Macolini, and T. Teitelbaum. Model checking uml-rt protocols. In *Workshop on Formal Design Techniques for Real-Time UML*, 2000.

[37] K. Zurowska and J. Dingel. Symbolic execution of collections of UML-RT State Machines. Draft, 2011.