An Illustrative Use Case of the DIVERSITY Platform based on UML Interaction Scenarios

Mathilde Arnaud[1] Boutheina Bannour[2] Arnault Lapitre[3]

CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 174, 91191 Gif-sur-Yvette, France

Abstract

DIVERSITY is a multi-purpose customizable platform based on symbolic execution. DIVERSITY has been designed for the purpose of managing the diversity of different semantics, but also the diversity of possible analyses based on symbolic execution. In this paper, we show how the input language of DIVERSITY can be used to encode the semantics of UML scenarios which include timing constraints expressed with the VSL language (standardized in the UML profile for embedded systems MARTE). We apply symbolic execution on practical scenarios of a system-on-chip example [a] in order to select test behaviors using an advanced exploration strategy implemented in DIVERSITY.

[a] Work partially supported by the European project OpenES www.openes-project.org

Keywords: Symbolic execution and tools, Modeling languages semantics, UML Scenario-based Interactions, VSL/MARTE timing constraints, Test selection strategy and coverage.

1 Introduction

Symbolic execution was first defined for programs [15]. The underlying concept consists in executing programs, not for concrete numerical values but for symbolic parameters, and computing logical constraints on those parameters at each step of the execution. Symbolic execution allows computing semantics of programs or models and representing them efficiently in an abstract manner. Model-based testing (MBT) is one of the most popular applications of symbolic execution[11,10,14,2]. Symbolic execution has been used to select some parts of the resulting symbolic representation of models, which may be infinite due to the presence of unbounded loops for example, according to some coverage objective. Test data are then generated from those chosen parts using constraint solving techniques. The increased efficiency of solvers in recent years [9,8,4] has helped symbolic execution to be adopted more widely for this purpose. Many symbolic execution based tools for formal treatments

[1] Email: mathilde.arnaud@cea.fr
[2] Email: boutheina.bannour@cea.fr
[3] Email: arnault.lapitre@cea.fr

This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

have been developed for diverse usages, for example the ones used in (Model-Based) Testing cited in the following survey [1]. Compared to these tools, the objective of the DIVERSITY platform is to offer an extensible platform to take into consideration various formal analysis possibilities. For this, DIVERSITY provides a common symbolic execution platform:

- generic enough to take into account semantics of a wide range of models;
- extensible to allow customizing of the basic symbolic treatments to implement specific formal functionalities (e.g. MBT algorithms, exploration strategies, etc.).

DIVERSITY is on its way to becoming an Eclipse open-source project [6]. In this paper, we give a brief introduction to DIVERSITY, and in particular we provide an example of its use. To illustrate extensibility, we show how an adaptation of the exploration strategy Hit-or-Jump [5],a heuristic whose aim is to achieve targeted test coverage, can be easily integrated into the customizable symbolic execution process. To illustrate the generality of DIVERSITY, we show how it provides interesting support of the semantics of the UML sequence diagrams [13]. Sequence diagrams display the UML graphical language used to describe the interaction behavior of system components. First, we have identified a subset of the input language of DIVERSITY to encode this interaction language. In fact, DIVERSITY provides a pivot language called *xLIA*(executable Language for Interaction and Architecture) which is a generic language with a variety of primitives which allow encoding a diversity of classical semantics. In particular, xLIA supports classical automata syntax involving symbolic data and communication actions. For MBT purposes, we have in previous work [3] provided a formal treatment of the semantics of UML sequence diagrams which involve timing constraints, specified using the Value Specification Language (VSL, standardized in the UML profile for MARTE [12]), by translating them into a kind of transition-labeled symbolic automata. In this paper, we extend this work by showing how these automata can be implemented in xLIA in an efficient way using asynchronous communication mechanisms and facilities to encode MARTE timing constraints. TIOSTS can be easily encoded as a subset of xLIA with a simple mechanism for communication. It appeared while implementing the translation mechanism described in [3] that it is not an efficient representation for symbolic execution in terms of performance, especially the message representation resulted in unnecessary computations. Thus it is useful to choose a different way of translating messages that alleviates this effect. We want to put particular emphasis on describing the translation mechanism and the use of DIVERSITY for coverage analysis, as an illustration of the more generic abilities of the tool.

Overview. Section 2 presents the transition-labeled automata in xLIA which are used to encode sequence diagrams and their associated symbolic semantics. Section 3 presents the symbolic execution process in DIVERSITY and how it is coupled with the Hit-or-Jump exploration strategy. Section 4 gives an example of the specification of a timed interaction behavior of a system-on-chip using sequence diagrams. Section 5 illustrates the translation of sequence diagrams into xLIA and exhibits some experimental results about their symbolic execution with DIVERSITY using the Hit-or-Jump exploration strategy. Finally, we conclude in Section 6.

2 Transition-labeled symbolic automata in xLIA

The xLIA language of DIVERSITY supports a form of symbolic automata involving variables to abstractly denote system states (we call them data variables) and variables to capture timing constraints (we call them time variables) on system executions. Such automata communicate by exchanging symbolic terms over channels acting as buffers where the sent data is stored to be consumed later. We call these automata *Symbolic Transition Systems* (*STS* for short). They are defined by triples (Q, q_0, Tr) where Q is a set of states which represent control points, q_0 is a distinguished control point of Q called the initial control point, and Tr is a set of labeled transitions. A transition is defined by a tuple $(q, \theta, \phi_t, \phi_d, act, \rho, q')$ where q (respectively q') is the source (respectively target) control point of the transition, ϕ_t is a first order formula on time variables called time guard, ϕ_d is a first order formula on data variables called data guard, θ is a set of time variables, act is a communication action and ρ is an assignment of data variables which represents state evolutions.

```
system< and > Sys {
  channel< buffer: fifo<*> > c ;
  statemachine< or > A {
   var time< real > t;
   var integer x, y, i;
   ...
   state q1 {
     transition tr --> q2 {
       update(t);
       tguard WF(t[i] - t[i-1] < 0.3) ;
       guard x < y ;
       output x+y via c ;
       {|and| y := y + 1; x := y; i := i+1 } ;
     }
   ...
```

Figure 1. DIVERSITY code for an output transition.

Data passing and update. The execution of a transition results in an action which may be the emission (resp. reception) of a value v on channel c, classically denoted $c!v$ (resp. $c?v$), or a particular action τ which stands for the absence of an observable action. Consider the transition tr of Figure 1 given in DIVERSITY encoding. The action of tr stands for the emissions of the value resulting from evaluation of $x + y$ through the channel c where x and y are data variables, and the channel c is associated with an unbounded fifo buffer. An example of input is `input x via c` which means that a value is received through channel c and assigned on x. Note the block `|and|{...}` introduced to encode the substitution of tr: Statements inside are evaluated in parallel [4]. For instance, assuming y is initially assigned with some value v before executing tr: if this block is not used, this gives rise to assigning x with $v+1$, yet in the semantics of interest, x has to be assigned with v. In the case where the action of tr is an input, the assignment induced by the input is taken into account before the other assignments. Then the transition is fired, if its data guard is satisfied before any data variable update in the case of an output action; and in the case of input action, only the assignment induced by the input is considered besides.

[4] All the top level statements are evaluated sequentially hence the importance of the ordering of the different components of a transition.

A system in our framework is defined by a set of communicating STS acting asynchronously, including for data passing. In fact, for any output of a value on a given channel (i.e. write to the associated buffer), that value may be consumed later by a different STS using an input action on the same channel (i.e. read from the associated buffer).

Time modeling. The symbol t corresponds to a time variable which is an array capturing consecutive execution instants of tr. These instants are picked in a time scale isomorphic to real numbers. The statement `update(t)` denotes a special action which appends the time instant of the last occurrence of tr to t. Time guards are evaluated after the time variables are updated. Consider now the time guard $t[i]-t[i-1] < 0.3$ which expresses that the delay between two successive executions of tr is lower than 0.3 time units. In the first execution, $t[i-1]$ is undefined and the time guard is conventionally evaluated to *true*. For this, we define the *weak form* of a time guard ϕ_t as $WF(\phi_t) \triangleq \phi_t \bigvee_{t[x]\in\phi_t}(x < 0 \vee x > len(t))$, where $t[x] \in \phi_t$ denotes a time instant term occurring within the time guard ϕ_t, and $len(t)$ denotes the length of t as an array of time instants. In fact, the weak form of a time guard characterizes situations where the index occurring in a time instant term is out of bounds for the corresponding time variable. For example, $WF(t[i] - t[i-1] < 0.3)$ results in $t[i] - t[i-1] < 0.3 \vee i < 1 \vee i > len(t)$ after simplification which is the actual guard to be satisfied for firing the transition.

Symbolic semantics. We provide STS with semantics using a symbolic execution technique which computes all the possible behaviors of the automata in the form of a *symbolic tree*. We start by discussing the symbolic execution of a transition, illustrated on tr in Figure 2. Such an execution is always described up to a reached *execution context* node in the tree, denoted EC which is composed of the following piece of information:

• a control point that determines which transitions can potentially be executed;

• a constraint on symbols denoting durations called *Path Time Condition*, PC_t;

• a constraint on symbolic data used for computation called *Path Condition* PC_d;

Path conditions fully characterize the constraints to be satisfied in order to follow the path in the symbolic tree associated with the EC.

• an instant, element of the time scale, represented by a sum of duration symbols, and representing the moment of occurrence of the last action encountered in the previous transition execution;

• and a substitution of the STS variables by terms over symbols, denoting their current associated values.

$$EC_k \xrightarrow{(\delta_{k+1})\ c!X+Y} EC_{k+1}$$

q_1	q_2
PC_t^k	$PC_t^k \wedge \delta_{j+1} + \ldots + \delta_{k+1} < 0.3$
PC_d^k	$PC_d^k \wedge X < Y$
$\delta_0 + \ldots + \delta_k$	$\delta_0 + \ldots + \delta_k + \delta_{k+1}$
$\sigma_k : t[9] \leftarrow \delta_0 + \ldots + \delta_j$	$\sigma_k : t[10] \leftarrow \delta_0 + \ldots + \delta_{k+1}$
$i \leftarrow 10,\ x \leftarrow X,\ y \leftarrow Y$	$i \leftarrow 11,\ x \leftarrow Y,\ y \leftarrow Y + 1$
$c \leftarrow w$	$c \leftarrow w.(X + Y)$

Figure 2. Symbolic execution of the transition of Figure 1.

Let us consider EC_k as a possible EC from which tr is a candidate transition to be fired. Executing tr from EC_k results in introducing a new symbol to denote the duration of tr which is δ_{k+1} and building a new EC, EC_{k+1}, where the two guards of tr are satisfied. In EC_{k+1}, the current time instant $\delta_1 + \ldots + \delta_{k+1}$ is appended to time variable t. PC_t gains a new constraint $\delta_{j+1} + \ldots + \delta_{k+1} < 0.3$ which denotes the satisfaction of the time guard $t[i] - t[i-1] < 0.3$: in this transition, $i = 10$ with $t[9] \leftarrow \delta_0 + \ldots + \delta_j$ and $t[10] \leftarrow \delta_0 + \ldots + \delta_{k+1}$. Similarly, the constraint $X < Y$ is added to the PC_d. When tr is fired, the communication action of tr, that is $c!x + y$, is denoted by the symbolic action $c!X + Y$ which results from substituting x and y by their respective associated symbols X and Y. x and y are then updated in EC_{k+1} by applying the transition substitution. Note that, in EC_k, the channel c is assigned w which is a finite word over symbolic fresh terms representing the content of the channel. In EC_{k+1}, the value bound to c is w to which is appended the emitted symbolic term $X + Y$.

The execution starts from the initial $EC_0 = (q_0, true, true, 0, \sigma_0)$ where the control state is q_0, the starting state of the STS, and σ_0 associates t and c resp. with the empty array and the empty word, and any other variable of the system with a distinct fresh symbol. It constitutes the root of the symbolic tree. The symbolic tree is computed by executing (symbolically) all STS transitions outgoing from EC_0 as described previously and then continuing inductively the execution from the STS control states reached by immediate previous executions.

3 DIVERSITY customizable symbolic processing and the Hit-or-Jump exploration strategy

DIVERSITY implements a generic symbolic execution processing (depicted in Figure 3) which can be customized on the fly thanks to the *filter mechanism* that we will discuss in this section.

The symbolic processing consists of five Steps $(i), \ldots, (v)$. The scheduling of these steps is cyclic. Each cycle consists in updating a queue of ECs. At the beginning of the first iteration of the cycle, the queue contains EC_0 which characterizes the initial symbolic values associated to the variables and the path condition is restricted to True because no constraint has yet been encountered. Each iteration step consists in: selecting one or more EC(s) (removed from the queue); computing their children ECs by symbolically executing all outgoing transitions from the control states reached by the parent ECs; deciding whether or not the parent ECs are added to the tree; in which case, their children ECs are added to the queue. The whole symbolic processing is based on the notion of *filter*. The purpose of a filter is to dynamically accept or reject ECs according to a specific user coverage purpose. It can be seen as a selection strategy to complement the traversal strategy in order to increase the chances of reaching the targeted coverage while avoiding combinatorial explosion.

Steps of the symbolic processing.

• Step (i) Selection of EC candidates for Step (ii): One or more EC are selected from the queue according to a customizable strategy. For the moment, the following exploration strategies for generating the symbolic tree are implemented: Ran-

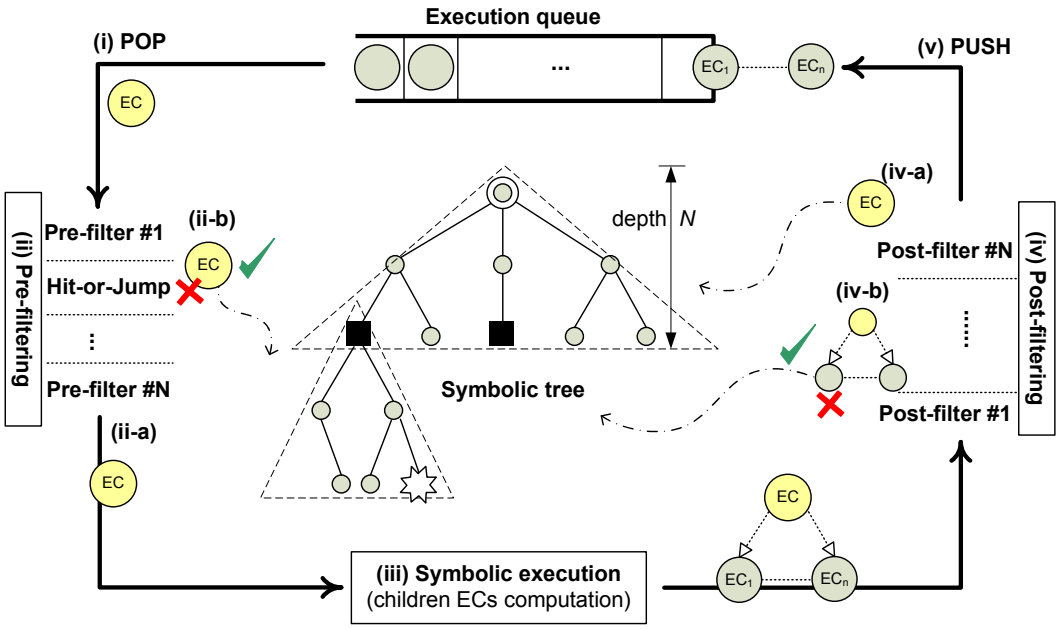


Figure 3. The running process of symbolic execution in DIVERSITY.

dom traversal, classical BFS (Breadth First Search) and DFS (Depth First Search) traversals. Some heuristics may however associate a weight with each of the EC, and thus induce an order of the ECs in the queue which means that the queue becomes a priority queue.

• Step (ii) Pre-filtering: Pre-filtering consists in applying one or more filters to reason about ECs before computing their children. If the EC successfully passes the scrutiny of each of the chained filters, it continues its way in the symbolic processing flow, through Step $(ii-a)$. Otherwise in Step $(ii-b)$, the EC will be ignored or possibly tagged (by an informative note on the reason for its failure) and inserted into the symbolic tree under construction. In the favorable case where all user coverage objectives are met, the symbolic processing stops.

• Step (iii) Symbolic execution: Each EC issued from Step $(ii-a)$ is evaluated symbolically. During the evaluation its children $EC_1, \ldots, EC_n$ are computed by symbolically executing outgoing transitions as detailed in Section 2.

• Step (iv) Post-filtering: Step (iv) is similar to Step (ii), except that the filter involved in post-filtering inspects the EC and its children to decide of the future of the symbolic processing. After passing the post-filters, there are two possibilities:

– Step $(iv-a)$ If successful, the symbolic processing continue with Step (v) in which case the the EC is added to the tree.

– Step $(iv-b)$ If failed, the EC and its children $EC_1, \ldots, EC_n$ are ignored or inserted in the symbolic tree (possibly tagged by an informative note on the reason for their failure);

As in Step (ii), in the favorable case where all user coverage objectives are met, the symbolic processing stops.

• Step (v): All the children $EC_1, \ldots, EC_n$ resulting from Step $(iv-a)$ are enqueued and the symbolic processing iterates with Step (i).

Hit-or-Jump exploration strategy. Classical exploration algorithms like

Breadth First Search (BFS) are implemented in DIVERSITY. However in some cases, using BFS exhaustively results in exploring a large number of paths in the symbolic tree which are irrelevant to the user targeted coverage criteria. DIVERSITY implements as a filter (which acts at the Pre-filtering Step) an adaptation of the heuristic traversal Hit-or-Jump (HoJ for short) [5]. This heuristics aims at computing a symbolic tree covering a declared sequence or set of automata constructs to cover such as transitions, states, input/output actions, or logical formulas to satisfy. The idea is to define a fixed maximal depth N for which a symbolic sub-tree is computed in BFS manner (sub-trees are delimited by dashed triangles in Figure 3). In order to build this sub-tree (of height N), each time an EC is selected at Step (i), the HoJ compares the relative height of EC [5] to N: if it is equal to N then the EC is added to the sub-tree; otherwise the symbolic execution continues with Step (iii). The sub-tree computation is finished when the execution queue is empty. At this level, the HoJ analyzes the resulting sub-tree to study whether or not some parts satisfy the coverage: (Hit) If some non-empty prefixes of the sequence has been covered, HoJ identifies the set of paths (of the sub-tree) that have covered the greatest prefix, and chooses one or several ones among them at random, else HoJ chooses at random one or several paths; (Jump) Once a path is chosen the whole process starts again from the last EC of the path (i.e. the target state of the last symbolic transition of the path) until the sequence is fully covered.

4 Sequence diagrams: System-on-chip example

As mentioned in the introduction, we use the following example: a System-on-Chip firmware in charge of dispatching graphical commands to several hardware units according to their priority and other factors. This example demonstrates several elements of interest: behavioral patterns such as parallel treatments, repeating behaviors and optional actions, and timing constraints that are modeled in VSL.

A set of UML sequence diagrams describes execution scenarios including timing information. Those scenarios represent the expected behavior of the system in terms of execution sequence order and timing. Sequence diagrams can be modeled easily using the graphical editor Papyrus [7] integrated with Eclipse.

System overview. The system is comprised of three components,
a Host, a Firmware, and a Hardware. The roles of those components are as follows:

- the Host sends commands to be executed by the Hardware;
- the Hardware executes blindly the commands it receives;
- the Firmware is in charge of scheduling which commands the Hardware executes.

Commands must be executed according to attached priorities. Furthermore, the commands can be executed in several phases. There are two tasks that the firmware must complete: ($FirmwareTaskA$) Pre-processing the commands it receives for maximum efficiency, adding relevant information and separating the command into several sub-commands, and ($FirmwareTaskB$) Computing an execution schedule on the fly and sending the sub-commands to the hardware according to this schedule. For the sake of simplicity and readability, we will consider only two levels of priority: HP (high priority) and LP (low priority). The firmware maintains two queues HPQ

[5] Distance of the EC to the root of the current sub-tree.

and LPQ containing the commands according to their priority level.

System behavior. The behavior described in diagrams *sdPreprocessing*, *sdEnqueuing*, *sdProcessing* and *sdFinishing* can be repeated an unknown number of times, which is captured by the loop operator. Furthermore, the four scenarios happen in parallel (as expressed by the par operator) though they share some data, such as the commands being treated. Each of the component roles is represented by a vertical lifeline where time evolves from top to bottom and where messages representing pieces of data transmitted between lifelines are represented by arrows between them. Messages have type Signal, which carries attributes, such as the specifics of a command to be executed, e.g. its priority and its weight. We use two lifelines to specify the role of the firmware: $FirmwareTaskA$ and $FirmwareTaskB$. $FirmwareTaskA$ is tasked with the communications with the host and the preprocessing of commands received from the host. $FirmwareTaskB$ is tasked with scheduling the execution of commands on the hardware, and communication with the hardware.

We use sequence diagrams with structuring operators which allow composing behaviors: the loop operator specifies a behavior which occurs cyclically, the alt specifies a choice between alternative behaviors, opt specifies a behavior which may occur optionally, and par specifies that behaviors occur in parallel. Those operators are graphically associated with rectangles (covering portions of lifelines and messages) to delimit the concerned behaviors. Four scenarios occur in parallel during the system execution: preprocessing of the commands, scheduling, computing, and reporting. We describe precisely only two scenarios that include elements of particular interest, especially in manner of time constraints: preprocessing and scheduling.

Preprocessing. The sequence diagram *sdPreprocessing* describes the preprocessing role of $FirmwareTaskA$: when the host requests that commands be treated, they are preprocessed and enqueued with respect to their priority. The potential arrival of a command is modeled by the opt operator. Upon reception of a new command, $FirmwareTaskA$ stores it in a queue and computes a new value for the boolean variable $Preemption$: its value is set to *true* if the new command newCmd is of (strictly) higher priority than the command currentCmd currently being treated. Furthermore, in practice, the firmware assigns $newCmd$ an internal id, denoted by $id(newCmd)$. The identifiers are incremented so that the identifiers reflect the order of reception of the commands. All along the scenarios, components perform actions we do not detail, such as retrieving attribute values when receiving a message, or the way the various queues are maintained. We chose to simplify the scenarios by hiding local executions on lifelines and operations on queues. If preemption is set to true, the firmware computes the next command nextCmd in the queue of highest priority and starts pre-processing nextCmd (possibly interrupting the pre-processing of another command of lower priority). When the pre-processing ends (i.e. when message endPreprocess(currentCmd) is received), the firmware enqueues the preprocessed command in the command queue HPQ or LPQ. Moreover, a timing constraint must be satisfied when pre-processing ends: namely, the preprocessing time of cmd, when taking into account the pre-processing time of com-

mands with higher priority received after the reception time of cmd, is within an acceptable amount depending on the weight of cmd and on a given factor F. More precisely, the following timing constraint must be satisfied:

$$t_2[i] - t_1[id(cmd)] - interruptTime(cmd) < weight(cmd) * F$$

where $interruptTime(cmd) = \sum\{finishTime(cmd') - startTime(cmd') \mid startTime(cmd') > startTime(cmd) \wedge priority(cmd') > priority(cmd)\}$ corresponds to the preprocessing time of the higher priority commands. In the case where there are only two kinds of priorities, the pre-processing of urgent commands cannot be interrupted. Consequently, if cmd is of priority HP, we have that $finishTime(cmd) = t_2[id(cmd)]$ and $startTime(cmd) = t_1[id(cmd)]$. Moreover, $startTime(cmd') > startTime(cmd)$ holds if and only if $id(cmd') > id(cmd)$, hence the formulation of the constraint given in the sequence diagram.

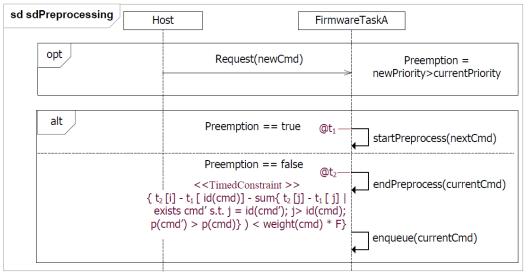


Figure 4: Preprocessing commands

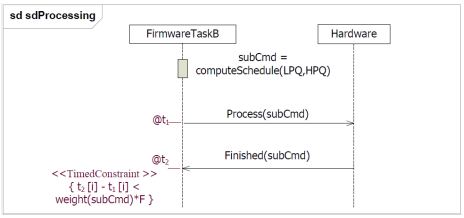
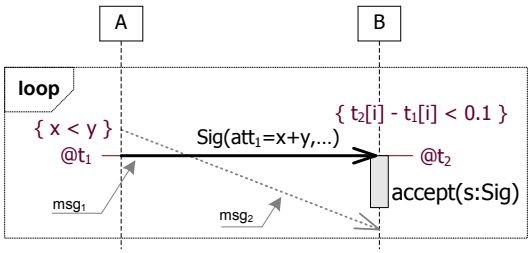


Figure 5: Scheduling preprocessed commands

Processing The aim of the preprocessing phase was to produce information in order to ease the processing of the commands: commands have been divided into subcommands built with a given quantity of information quanta to be processed, and stored in one of the command queues HPQ or LPQ. The subcommands are processed according to their priorities, in a roundrobin manner. $FirmwareTaskB$ computes from HPQ and LPQ . The subcommand is then sent to the hardware to be processed, and the hardware sends back a message finished(subCmd) when the computations end. The processing time for a single subcommand must be bounded by the weight of this subcommand, as captured by the time constraint $t_2[i] - t_1[i] < weight(subCmd)$. Note that the number of iterations of this behavior depends on the number of commands received and on the length of those various commands, and could thus be computed in order to prevent unnecessary interleaving.

5 Translation of sequence diagrams and experiments

Translation into xLIA.
We consider the subset of timed sequence diagrams with asynchronous message passing as they were presented in section 4. The semantics is obtained by translating timed sequence diagrams into a system of communicating STS, each corresponding to a lifeline. We show in figure 6 how to build transitions tr_1 and tr_2 that represent respectively the emission of signal Sig by lifeline A and its reception by lifeline B. The emitted signal conveyed by message $msg1$ transits through a channel $msg1Channel$. In fact we associate each message in the sequence diagram with




```
system< and > Sys {
  channel< buffer: fifo<*> > msg1Channel ;
  signal Sig (integer att1, ...);
  statemachine< or > A {
   var time< real > t1;
   var integer x, y;
   state q1 {
     transition tr1 --> q2 {
       update(t1);
       guard x < y ;
       output Sig(x+y,...) via msg1Channel ;
     }
   ...
  statemachine< or > B {
   timevar< real > t2;
   var Sig s;
   var integer i_t2;
   state q3 {
     transition tr2 --> q4 {
       update(t2);
       tguard WF(t2[i_t2] - t1[i_t2] < 0.1) ;
       input Sig(s) via msg1Channel ;
       i_t2 := i_t2 + 1
     }
   ...
```

Figure 6. Translation of asynchronous signal passing.

a channel equipped with fifo buffering whose role is to store sent signals until the target lifeline is ready to receive them. Having one channel per message allows our translation to support, for instance, message overtaking. For instance, messages msg_1 is sent after message msg_2 (depicted in dashed line), but the target lifeline receives it first. Such inversions may occur when communication is asynchronous as allowed in the UML standard [13]. We have introduced the notion of signal in DIVERSITY in the input/output actions: First the type Sig is declared globally with its attributes, then it can be seamlessly used in communication actions conforming to UML semantics:

• The output action `output Sig(x+y,...) via msg1Channel` builds implicitly an instance of the signal Sig with attributes $att_1, \ldots$ assigned with $x + y, \ldots$. This instance (with the attributes filled in) is buffered in the channel $msg1Channel$.

• The input action `input Sig(s) via msg1Channel` denotes that the signal received from channel $msg1Channel$ is stored on a local variable s of type Sig in the target lifeline STS B (recall that the signal is declared globally at the system level). This way, signal attributes can be used in computations by B.

Note that when there is a time guard associated to the emission or reception of a message using time instant terms of the form $t[i]$, we use an index i_t to capture the last instant of the occurrence of the execution. This is the case for the reception of the message msg_1: in tr_2, the index i_{t_2} refers to the last defined location in t_2 and is incremented accordingly after being used in the time guard $t_2[i_{t_2}] - t_1[i_{t_2}] < 0.1$.

The translation of combined operators consists mainly in creating decision and junctions states/transitions and then inductively translating the behaviors defined in their operands. We illustrate the translation of the *alt* operator in figure 7. From

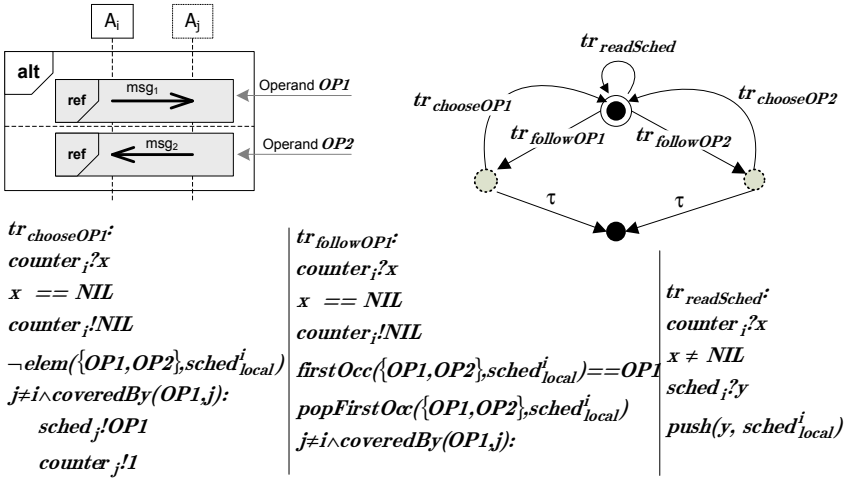


Figure 7. Translation of the choice operator.

the local point of view of lifeline A_i, the *alt* operator gives a non-deterministic choice between two scenarios inside operands OP_1 and OP_2: either it sends the message msg_1 or it consumes the message msg_2 from the associated channel. For example in the latter case, the lifeline A_j is meant to send the message msg_1 so that both lifelines execute the same scenario (inside OP_2). The translation introduces decision transitions in the STS to reflect those choices. The lifelines A_i, A_j will inform each other of their choice by sending coordination messages through dedicated *scheduling channels* $sched_i, sched_j$. So A_i being busy with previous executions, may receive from A_j in $sched_i$ successively OP_1, OP_2, OP_1 informing it to follow their nested behaviors to be consistent with the choices of A_j (see the transition $tr_{followOP1}, tr_{followOP2}$). Or A_i may operate at a faster rate and initiate the choice (see the transitions $tr_{chooseOP1}, tr_{chooseOP2}$). Note that reads from communication channels are by hypothesis blocking, i.e. A_i cannot test for emptiness of $sched_i$. We use an *counter channel* $counter_i$ which is filled: by 1 each time a value is written on $sched_i$ and; by a particular symbol NIL that indicates the end of the buffer. This counter channel forces A_i to read all the values in $sched_i$ while NIL is not read (see transitions $tr_{readSched}$). All the operand names read from $sched_i$ are stored in a local buffer $sched^i_{local}$ before being analyzed to look for a given operand. Knowing that in full generality, some operand $OP \neq OP_1, OP_2$ associated with another combining operators covering A_i and some other lifeline $A_k, k \neq j$ may also be stored in $sched^i_{local}$, we have defined in DIVERSITY advanced routines $firstOcc$ and $popFirstOcc$ on local schedulers to access, and respectively to consume, the value (among a set of given values) that occurs the first in the scheduler.

Experiments with DIVERSITY. We have developed `UML2DIVERSITY` a plug-in for Papyrus [7] that allows translating automatically sequence diagrams into xLIA. The plug-in generates a textual file describing the System-on-Chip specification in xLIA from the sequence diagrams given in Section 4: It includes respectively 107 and 199 STS states and transitions. In order to demonstrate the interest of using the HoJ heuristic described in Section 3, we have defined two families of objectives to cover:

• $Obj_1(k)$: Covering sequentially $Request(newCmd)$ k times,

Objective	Success Rate	Best execution metrics			
		#Steps	#Jumps	Exec. Time	#ECs
$Obj_1(1)$	100%	222	11	$1s607$	301
$Obj_1(2)$	90%	548	23	$3s867$	767
$Obj_1(4)$	100%	1,902	40	$12s561$	2,663
$Obj_1(8)$	100%	4,443	93	$28s408$	6,202
$Obj_1(16)$	90%	9,133	190	$1m4s3ms$	12,764
$Obj_2(16)$	100%	8,885	181	$1m2s479ms$	12,435

Table 1
Hit-or-Jump metrics.

then $enqueue(currentCmd)$ k times, and $finished(cmd)$ k times.

- $Obj_2(k)$: Covering k times the sequence

$$Request(newCmd) \dots enqueue(currentCmd) \dots finished(cmd).$$

We provide for several values of k the success rate for 10 trials. For the most effective execution among successful trials, we also give the following metrics: the number of execution steps computed, the number of Jumps, the execution time and the number of ECs computed. We parametrized the symbolic exploration strategy as follows: the fixed maximal depth N for the exploration of a sub-tree was 5, the number of ECs to be kept in the event of a Hit (at least one transition of interest was covered during the exploration of the current sub-tree) was 7 and the number of ECs to select in the event of a Jump (no transition of interest covered in the sub-tree at the end of its construction) was 5.

We observe that the strategy can sometimes fail at covering all the desired transitions. Indeed, the Hit-or-Jump strategy is a heuristics where some randomness is involved, namely the number of branches to be kept at the end of each step. We observe that there are very few occurrences of timeouts, and the successful explorations are very fast. In particular, remark that the measures do not grow exponentially, which would be the case if we had opted to use a more classical exploration strategy. As indicative of the efficiency of the HoJ strategy, note that in order to cover a sequence of transitions such as $Request(newCmd) \dots enqueue(currentCmd) \dots finished(cmd)$, any of the classical exploration strategies would have to reach a depth of at least 40, and that is costly. For information, using the BFS strategy to explore a depth of 40, we computed more than 500,000 execution steps. It is obvious to see that such computations are too costly to produce test inputs in large quantities, and why the use of heuristics in those cases is desirable.

6 Conclusion

The expressive syntax of DIVERSITY's xLIA input language has allowed us to encode most of the concepts of UML sequence diagrams, which involve additionally MARTE timing constraints. In the future, we plan to support a larger set of UML, especially the state machines seem a particularly interesting subset of UML to reflect the xLIA specification in the form of transition-labeled symbolic automata. In this paper, we have also described the symbolic execution implemented in DIVERSITY and how it has been coupled with the fast exploration strategy Hit-or-Jump thanks to the customization facilities provided by DIVERSITY. The performance experi-

ments with system-on-chip example have been concluding on the suitability of the Hit-or-Jump strategy to achieve coverage objectives on highly concurrent system models. We also plan to integrate in DIVERSITY more advanced techniques such as the *Partial-Order Reduction* [16] for efficient exploration of concurrent models.

References

[1] Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold and P. Mcminn, *An orchestrated survey of methodologies for automated software test case generation*, J. Syst. Softw. **86** (2013), pp. 1978–2001.

[2] Bannour, B., J. P. Escobedo, C. Gaston and P. L. Gall, *Off-line test case generation for timed symbolic model-based conformance testing*, in: *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*, 2012, pp. 119–135.

[3] Bannour, B., C. Gaston and D. Servat, *Eliciting unitary constraints from timed sequence diagram with symbolic techniques: Application to testing*, in: *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*, 2011, pp. 219–226.

[4] Barrett, C., C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds and C. Tinelli, *CVC4*, in: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 171–177.

[5] Cavalli, A. R., D. Lee, C. Rinderknecht and F. Zaïdi, *Hit-or-jump: An algorithm for embedded testing with applications to IN services*, in: *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China*, 1999, pp. 41–56.

[6] CEA LIST, *DIVERSITY*, http://projects.eclipse.org/proposals/diversity/.

[7] CEA LIST, *Papyrus*, http://www.eclipse.org/papyrus/.

[8] de Moura, L. M. and N. Bjørner, *Z3: an efficient SMT solver*, in: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340.

[9] Dutertre, B. and L. M. de Moura, *A fast linear-arithmetic solver for DPLL(T)*, in: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 2006, pp. 81–94.

[10] Frantzen, L., J. Tretmans and T. A. C. Willemse, *A symbolic framework for model-based testing*, in: *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, 2006, pp. 40–54.

[11] Gaston, C., P. L. Gall, N. Rapin and A. Touil, *Symbolic Execution Techniques for Test Purpose Definition*, in: *Proc. of Int. Conf. Testing of Software and Communicating Systems (TestCom)* (2006), pp. 1–18.

[12] Group, O. M., *A UML profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, VSL* (2009), http://www.omg.org/spec/MARTE/.

[13] Group, O. M., *Unified modeling language (uml)* (2012), http://www.uml.org/.

[14] Jéron, T., *Symbolic model-based test selection*, Electr. Notes Theor. Comput. Sci. **240** (2009), pp. 167–184.

[15] King, J. C., *A new approach to program testing*, Proceedings of the international conference on Reliable software **10** (1975), pp. 228–233.

[16] Valmari, A., *Stubborn sets for reduced state space generation*, in: *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990* (1991), pp. 491–515.