Timed symbolic testing framework for executable models using high-level scenarios

Mathilde Arnaud, Boutheina Bannour, Arnaud Cuccuru, Christophe Gaston, Sébastien Gerard, Arnault Lapitre

Timed symbolic testing framework for executable models using high-level scenarios*

Mathilde Arnaud, Boutheina Bannour, Arnaud Cuccuru
Christophe Gaston, Sebastien Gerard and Arnault Lapitre

CEA, LIST, LISE Laboratory
Point Courrier 174, 91191, Gif-sur-Yvette, France
{firstname.lastname}@cea.fr

Abstract. Refining high-level system scenarios into executable models is often not automatic and subject to implementation choices. We develop techniques and tools combining different modes of simulation in order to assess automatically the correctness of executable fUML activities with respect to system scenarios specified as UML MARTE sequence diagrams. In this paper, we show how test data are extracted from sequence diagrams using symbolic execution and how they are used as inputs to test system activities in the standardized fUML virtual machine.

1 Introduction

Over time, software systems tend to be used in highly critical scenarios in a variety of areas. Examples include advanced driver assistance systems, autopilots in aircraft or railway systems, etc. Such systems are often made of multiple components which are highly concurrent, and are all tied together in complex ways. Hence, standards under which large-scale software systems have to be developed are increasingly stringent and demanding regarding confidence in their quality. Formal methods are essential to achieve higher confidence levels since they allow replacing validation operations done manually by automatic or semi-automatic techniques with mathematical foundations justifying their use. Unfortunately, the reference models used in validation are often large and therefore it is unclear how much confidence to award them. A top-down process which relies on refinement techniques should be used in order to shift the burden of formal analysis from detailed reference models to the correctness of refinement steps. In this paper, we do not define refinement mechanisms for transforming high level models into low level executable models. We rather consider the refinement as maintaining a correctness relation which states sufficient conditions on conforming executable models with respect to high level scenarios. The objective of the paper is to investigate a refinement approach integrated with the Unified Modeling Language (UML) which is a graphical language that can be used to design complex software systems at different levels of abstraction. A complex system can be first specified as a UML sequence diagram with timing properties

* Work partially supported by the European openES project.

using the constraint language MARTE::VSL [9]. In sequence diagrams, one may describe execution scenarios in terms of partially-ordered sequences of messages exchanged between basic interaction points (called ports) owned by components to communicate with their environment. Message descriptions may include constraints on the type of value transmitted and the time at which the message is processed. Conceptually, sequence diagrams characterize requirements on system behaviors while abstracting as much as possible internal computation flows inside components. Later in the design cycle, a more detailed model of each component behavior may be designed with UML activity diagrams. Activities in UML are flow charts built with communication and computation actions. System activities are deterministic and directly executable according to the fUML execution semantics [11]. In our approach, we develop a tooled testing process to assess automatically the conformance of the system activities with respect to sequence diagrams. We also pay particular attention to evaluating compliance with timing constraints since the system depends on those constraints being satisfied to operate correctly. Test data are generated from sequence diagrams by symbolic execution [13] which improves behavioral coverage and hence fault-detection capability of the testing process.

This paper is organized as follows. The section 2 gives an overview of our approach and tools. It also introduces an automotive system used as a running example in this paper. Section 3 describes how sequence diagrams are used in our approach to specify high-level scenarios and their symbolic treatment. Section 4 shows how activities of components in the system are designed and presents their execution semantics in the fUML virtual machine. Section 5 defines the testing process. Section 6 presents the experimental results and discusses practical issues in test coverage within the context of our models. Section 7 reviews some related works. Finally, Section 8 draws the conclusions.

2 Approach overview

The goal of this section is twofold. First, we briefly discuss the tools involved in the approach, and then we present the approach itself which is illustrated step-by-step in Figure 1.

• *Papyrus* is the tool used for modeling UML diagrams[1]. Papyrus is a graphical editing tool for UML2 integrated with Eclipse.

• *Moka* is an eclipse plug-in which aims at providing support for model execution in the context of Papyrus [20]. *Moka* provides basic execution, debugging and logging facilities for foundational UML (fUML) [11], an executable subset of UML with precise operational semantics. We use activity diagrams that are part of fUML to model the internal behavior of components.

• `sdToTIOSTS` is an eclipse plug-in for Papyrus. It is used to translate sequence diagram models into Timed Input Output Symbolic Transition Systems (TIOSTS) [3]. TIOSTS are symbolic automata with time guards. Resulting TIOSTS can be analyzed by means of the symbolic execution tool *Diversity*.

[1] www.eclipse.org/papyrus

• *Diversity* is a symbolic automatic analysis and testing tool [4]. *Diversity* uses symbolic execution techniques to compute a symbolic tree representing all the possible executions of a TIOSTS. A symbolic path represents all the timed traces that can be computed by satisfying path conditions on data and time. *Diversity* offers coverage criteria such as transition coverage. Finally, *Diversity* is coupled with sat-solvers in order to generate timed traces associated to symbolic paths and to test whether a timed trace reveals non-conformance relatively to a trace-based conformance relation called tioco [19].

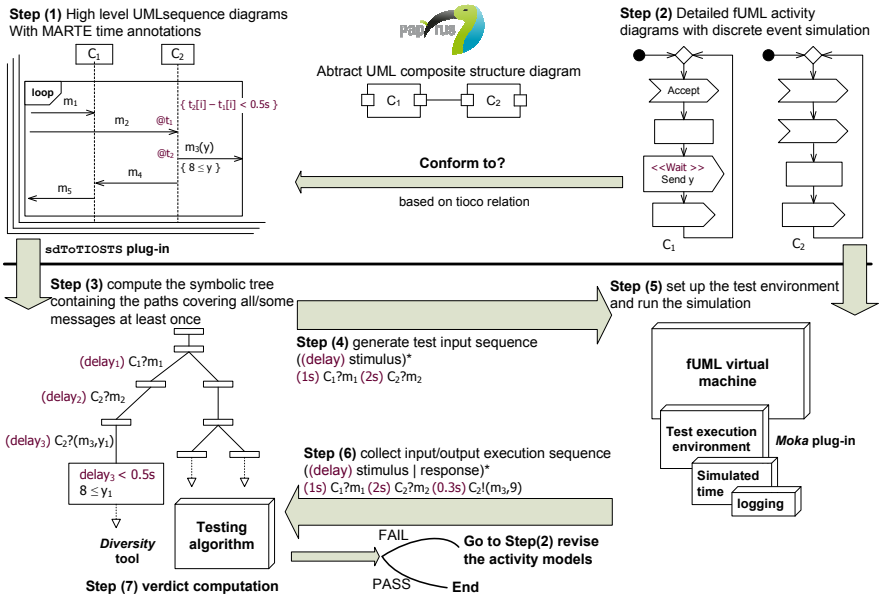


Fig. 1: Validation Process.

Let us overview the different steps of the approach in Figure 1. The first step (1) consists in specifying system scenarios which describe the intended interactions between all components of the system. The second step (2) consists in refining the scenarios into an executable model which specifies with an activity diagram the internal behavior of each component. System scenarios as sequence diagrams are analyzed with *Diversity* in step (3). For each sequence diagram, *Diversity* computes a symbolic tree, where each path denotes a possible (symbolic) execution in the sequence diagram. Then a path is selected in step (4) relating to a specific behavior and a sequence of stimuli (as a timed trace) is extracted from it. Next in step (5), the fUML virtual machine of the tool *Moka*, being supplied with the test stimuli, is used to set up a test environment and execute the system activities. In Step (6) the system responses are collected by *Moka*. The latters are taken as inputs by the testing algorithm in *Diversity* which computes in step (7) a verdict concerning the *tioco*-conformance of execution and the coverage of the requirement. Naturally, in case of fault-detection the system activities need to be revised by the designer.

Automotive example. For the rest of the paper, we will use a running example whose structure is depicted in Figure 2. It specifies a rain-sensing wiper system

in a car, denoted RSW. Three components are involved in the RSW system: a controller, a calculator and a wiper motor. These are some of the requirements that must hold for the actual implementation of the system: **(R1)** RSW adjusts the wiping speed according to the amount of rain detected; **(R2)** RSW controls automatically the adjustment of the wiper activity; and **(R3)** RSW response time is less than 0.5 seconds after detection.

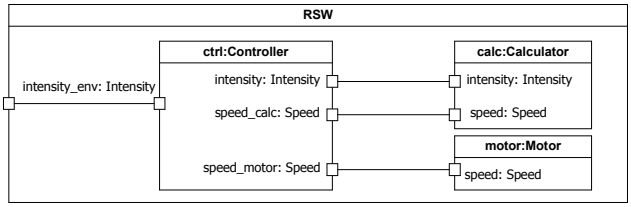


Fig. 2: RSW composite structure diagram.

3 Interaction scenarios and symbolic simulation

We develop next high level system scenarios and explain how we analyze them.

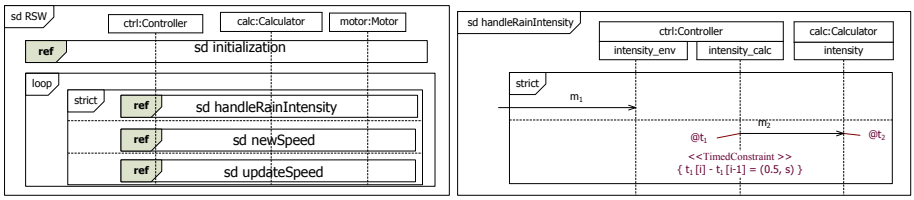


Fig. 3: RSW sequence diagram–Subdiagram representing the handling of the received rain intensity.

3.1 Sequence diagrams

The sequence diagrams given in Figures 3–5 describe the global behavior expected from the RSW components. Each of the ports is represented by a lifeline. The behavior described in the diagram is repetitive and the number of iterations is not known beforehand, which is captured by the *loop* operator. The controller receives inputs from the environment in message m_1 and sends periodical updates to the calculator about the rain intensity on channel m_2. Note that the messages are asynchronous and may thus be received at a later date than their emission. The periodicity of the updates is given by a time constraint of the form $t_1[i] - t_1[i-1] = (0.5, s)$, where i represents how many times the behavior has looped and t_1 is an array containing at index i the time value associated. The calculator then computes what speed the wiper should adopt given such intensity: this is represented in the sequence diagram by the computation of a new value for the speed variable : $new(speed)$. The calculator sends the result of this computation back to the controller over message m_3. Then there are two

alternatives as captured by the *alt* operator: either the new computed value for the speed of the wiper is the same value as the previously computed speed, and in that case nothing need be done, or the new value is different. In that case the value of the stored previous speed is updated and a signal is sent to the motor with the new speed value (message m_4). In a sequence diagram, the lifelines are running asynchronously. In order to synchronize some executions, we use the *strict* operator which ensures that the behavior in the first part is finished before the behavior in the second part begins.

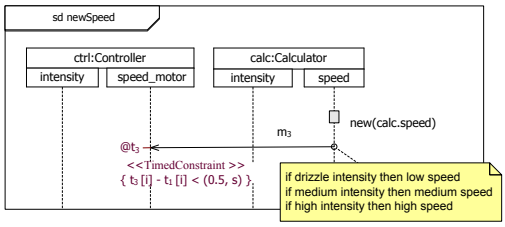


Fig. 4: Controller computing new speed.

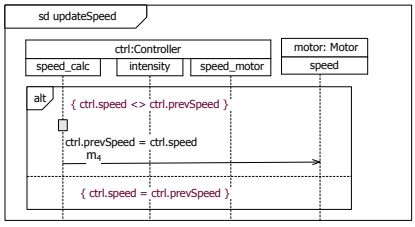


Fig. 5: Updating speed.

3.2 Symbolic execution

In order to formally reason about sequence diagrams, we provide them with formal semantics given by TIOSTS automata [3]. Symbolic execution may be carried out on such automata and thus we obtain timed traces representing possible behaviors characterized by the sequence diagram.

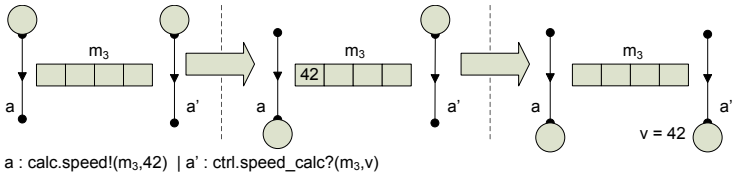


Fig. 6: Example of transitions representing an asynchronous communication.

Translation into communicating TIOSTS. Let V be a set of variables and C be a set of channels. A TIOSTS over (V, C) is a triple (Q, q_0, T) where Q is a set of states, q_0 is the initial state and T is the set of transitions. Transitions are of the form $q \xrightarrow{t,\phi_t,\phi_d,act,\rho} q'$ where q, q' are states, t is an array of time instants, ϕ_t and ϕ_d are guards resp. over the time and the data constraints, ρ is a substitution over variables of V and *act* is one of the possible actions that can be triggered. Possible actions are: receiving a value x on channel $c \in C$ denoted by $c?x$; sending value u on channel $c \in C$ denoted by $c!u$; assigning a new value to variable x denoted by $new(x)$; and the empty action τ. We translate a sequence diagram into a set of TIOSTS: each lifeline is translated into a TIOSTS, by translating successive events into transitions with the appropriate constraints. We only detail the translation of the asynchronous communications which was not considered in [3]. An unbounded FIFO variable is associated to each channel in order to emulate the communication actions. Each time a message m is sent

on a channel c by a lifeline l, it writes m on the FIFO associated to channel c. Each time a lifeline receives on channel c, it reads on the FIFO associated with c the message. This process is illustrated in Figure 6.

Symbolic tree. Reasoning with concrete input values can result in a very large, possibly infinite, number of executions of the system. We use symbolic execution techniques instead. The underlying idea is to abstract some of the values, be they data values or time values, as variables, and thus characterize classes of executions. Besides data, we define symbolic handling of TIOSTS time variables. Symbolic states allow storing information about the execution of the system that may constrain the values of the variables in *path conditions*. A symbolic

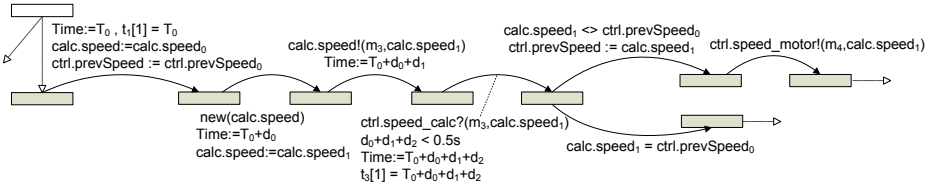


Fig. 7: Part of the symbolic tree of the RSW sequence diagram.

execution corresponds to a concrete one if and only if the collection of path conditions is satisfiable. For example, the path condition collected in one path of Figure 7 is made of two parts, the time path condition $PC_t = d_0 + d_1 + d_2 < 0.5s$ and the data path condition $PC_d = calc.speed_1 <> ctrl.prevSpeed_0$. Using solving techniques on this path condition, we can deduce concrete traces.

Timed traces of sequence diagrams. We use a set D of durations and a data model M which includes most common types. A sequence diagram SD is defined over a signature $(\mathcal{P}, \mathcal{M}sg)$ where $\mathcal{P}$ is a set of ports and $\mathcal{M}sg$ is a set of messages. The set of communication actions on port $p \in \mathcal{P}$ is $Act(p)$ of the form $I(p) \cup O(p)$, where $I(p) = \{p?(m,v) | m \in \mathcal{M}sg, v \in M\}$ and $O(p) = \{p!(m,v) | m \in \mathcal{M}sg, v \in M\} \cup \{\overline{p}!(m,v) | m \in \mathcal{M}sg, v \in M\}$: an action of the form $p!(m,v)$ corresponds to an emission of a message by p; $\overline{p}!(m,v)$ corresponds to a reception of a message sent by an internal component of the system, and $p?(m,v)$ corresponds to a reception of a message coming from the environment of SD. We define the set $I(SD)$ of all inputs (resp. outputs) in SD as $\bigcup_{p \in \mathcal{P}} I(p)$ ($\bigcup_{p \in \mathcal{P}} O(p)$). The set of all communication actions in SD is $I(SD) \cup O(SD)$, denoted $Act(SD)$. A *timed trace* of a sequence diagram SD is a word from $(Act(SD) \cup D)^*$ which respects the causal order inferred from the sequence diagram together with the timing constraints and the performed computations on inputs. The define $TTraces(SD)$ as the set of all timed traces of SD.

4 Activity diagrams and numeric simulation

The objective of this section is to introduce a subset of activities that we use and discuss their underline execution semantics in the fUML virtual machine.

4.1 Activity diagrams

Components involved in the system are refined by designing an activity diagram for each individual component. Each activity diagram specifies the communica-

tion and the computation logic. Figure 8 illustrates activities associated with the controller and the calculator of the RSW system.

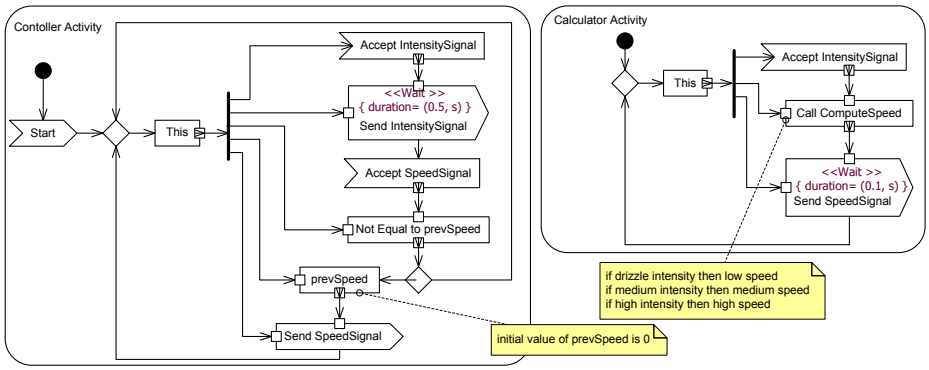


Fig. 8: RSW activity diagrams.

Both activities specify a cyclic behavior. Let us discuss actions of the controller:
• *AcceptEventActions* (nodes *Start*, *Accept SpeedSignal*, and *Accept IntensitySignal*) specify synchronization points, where the controller waits for inputs from its environment;
• *SendSignalActions* (nodes *Send IntensitySignal*, and *Send SpeedSignal*) specify asynchronous communications between the Controller and its environment;
• Other actions specify computations or access to context information of the component object: *Not Equal to prevSpeed* determines if the received speed value is equal to the previous one for example.
Nodes may be annotated (using the stereotype <<Wait>>) with durations which must elapse before they are executed. E.g. *Send IntensitySignal* corresponds to sending a rain intensity value to the calculator immediately after a $0.5s$ delay.

4.2 fUML virtual machine and Discrete-Event simulation

The fUML virtual machine (VM) is implemented in the Moka tool. It allows the execution of fUML activities of components structured with UML composite diagrams. A specific simulation library defining a Model of Execution (MoC) is responsible for controlling the execution and simulating extra-functional aspects such as timing features. In this paper, we use a particular MoC, *discrete-event MoC* which introduces a discrete model of time. During the traversal of the control flow of the activity, an event is triggered by the fUML VM each time a communication action is interpreted and is stored in the event queue of a scheduler. Events correspond to fUML signals, carrying a delay. This value indicates the time when this communication node will be woken up. During the execution, the event with the smallest delay is then selected, firing the communication action referenced by this event and removing it from the event queue.

Once communication actions are fired, the fUML VM is in charge of propagating the values through the connector architecture conforming to [10]. Finally as glimpsed in Section 2, we need to collect executions traces in order to analyze their conformance: to that end we integrate in the fUML VM run-time logging capabilities.Thanks to discrete-event MoC, the execution trace is enriched with durations that have elapsed between exchanged data.

5 Conformance testing

We present in this section the *tioco* conformance relation and then, we describe in practice the conformance testing process.

5.1 Conformance relation

In our settings, an *Activity model* $\mathbb{A}$ defined over a set of ports $\mathcal{P} = \amalg_{i \leq n} \mathcal{P}_i$, is a finite set $A_1, \ldots, A_l$ of activities defined respectively over $\mathcal{P}_1 \ldots \mathcal{P}_n$ where n is the number of components in the system. The *Activity model* is directly executable in the fUML virtual machine as explained in Section 4. The simulation history of an activity model can be mathematically characterized as a set of traces defined over the set of ports denoted $TTraces(\mathbb{A})$. We also define an *Interaction model* $\mathbb{S}$ defined over $(\mathcal{P}, \mathcal{M}sg)$ as the set of k sequence diagrams $SD_1, \ldots, SD_k$ defined respectively over $(\mathcal{P}_1, \mathcal{M}sg_1) \ldots (\mathcal{P}_k, \mathcal{M}sg_k)$ such that: for all $i \leq k$,$\mathcal{P}_i \subseteq 2^{\mathcal{P}}$; and $\mathcal{M}sg = \amalg_{i \leq k} \mathcal{M}sg_i$. That is, a sequence diagram in $\mathbb{S}$ may include only a subset of all system ports and the sets of messages are disjoint. Timed traces of an interaction model $\mathbb{S}$, denoted $TTraces(\mathbb{S})$, is the union of timed traces of all sequence diagrams in $\mathbb{S}$, $\bigcup_{i \leq k} TTraces(SD_k)$ (we define similarly the set of all input and output sets of $\mathbb{S}$ resp. $O(\mathbb{S})$ and $I(\mathbb{S})$). Let $\sigma \in TTraces(\mathbb{S})$, we define the auxiliary function $h(\mathcal{M}sg, \sigma)$ as follows: if σ is of the form $p_1 \diamondsuit (m_1, v_1) \ldots p_n \diamondsuit (m_n, v_n)$, where $\diamondsuit \in \{?, !\}$ then $h(\mathcal{M}sg, \sigma) = p_1 \diamondsuit v_1 \ldots p_n \diamondsuit v_n$; otherwise $h(\mathcal{M}sg, \epsilon) = \epsilon$. Let us consider further the following definition: Let σ_1, σ_2 be two traces respectively in $TTraces(\mathbb{S}), TTraces(\mathbb{A})$. σ_2 is not distinguishable from σ_1 w.r.t $\mathcal{M}sg$, denoted $\sigma_2 \sim_{\mathcal{M}sg} \sigma_1$, if and only if $\sigma_2 = h(\mathcal{M}sg, \sigma_1)$. We adapt in the following definition the conformance relation *tioco* [19] in order to define the correctness of an activity model $\mathbb{A}$ w.r.t an interaction model $\mathbb{S}$.

Definition 1 (tioco). *Let $\mathbb{S}$ be an interaction model over $(\mathcal{P}, \mathcal{M}sg)$ and $\mathbb{A}$ be an activity model over $\mathcal{P}$.* $\mathbb{A}$ conforms to $\mathbb{S}$*, denoted $\mathbb{A}$ tioco $\mathbb{S}$, if and only if for every $\sigma \in TTraces(\mathbb{S})$ and $r \in O(\mathbb{S}) \cup D$,*

$$\forall \sigma' \in TTraces(\mathbb{A}) : \sigma' \sim_{\mathcal{M}sg} \sigma.r \Longrightarrow \sigma.r \in TTraces(\mathbb{S})$$

Example. Let us consider the requirement **(R1)**. Applying the previous definition, the following trace of the activity model violates this requirement:
$(6ms).ctrl.intensity\_env?HIGH.(2ms).ctrl.intensity!HIGH.(1ms)$
$.\overline{calc.intensity}!HIGH.(3ms).calc.speed!FAST.(1ms).\overline{ctrl.speed\_calc}!FAST$
$.(6ms).ctrl.speed\_motor!FAST.(2ms).\overline{motor.speed}!FAST.(3ms)$

$.ctrl.intensity\_env?DRIZZLE.(484ms).ctrl.intensity!DRIZZLE.(1ms)$
$.\overline{calc.intensity}!DRIZZLE.(4ms).calc.speed!FAST$

The violation is due to the inappropriate calculated wiper speed. In fact, for the first detected amount of rain, high intensity, the fast speed is correct. At drizzle, the calculator computed again a fast speed, however the speed must be low.

5.2 Testing process

We use the so-called *off-line testing* presented in [3] to test the conformance of the activity model, in the sense of *tioco*. The process starts with choosing a path in the symbolic tree as a *test purpose* which covers a specific requirement. Then using constraint solving techniques, the idea is to derive a sequence of concrete inputs and durations which would allow the execution of the activity model to potentially cover the test purpose. In order to submit the input sequence, a tester is connected to the system. The behavior of the tester is specified with an fUML activity in a textual form, ALF (Action Language for Foundational UML)[2]. This allows for automatic generation of tester behavior from input sequences. Repeatedly, the tester sends an input value on the targeted port of the system and then waits for the subsequent duration. See Figure 9 for illustration on the RSW system. Note that classically *tioco* assumes that *the system under*

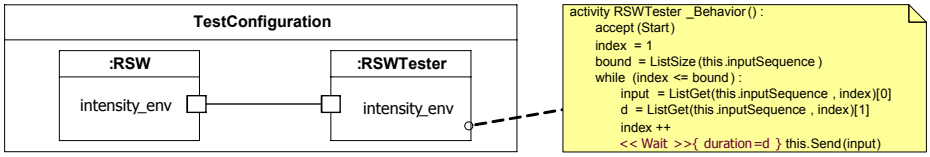


Fig. 9: RSW test configuration – tester activity in ALF syntax.

test is input enabled, i.e. that an SUT cannot refuse an input from the tester. This hypothesis is satisfied by running the tester behavior in the fUML virtual machine which is also used as the test harness in our framework. The other hypothesis of *tioco* is called *time elapsing* and expresses that the absence of an output amounts to observing no reaction from the system during a delay. As a discrete time simulator, the fUML virtual machine grants this hypothesis for consistent executions. Indeed after the initialization at the beginning of the execution, the simulated time elapses to reach the earliest waiting time specified either in the tester or in any component of the system.

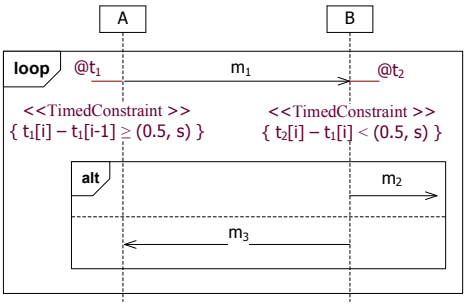
As required by off-line testing, outputs and durations between them are logged during the test execution. This output sequence is merged with the input sequence to form a timed trace. In the final phase, the trace is analyzed w.r.t. the interaction model to emit a verdict: PASS, if we observe exactly the desired behavior; INCONC, if we observe a behavior that is not compatible with the test purpose; and FAIL, if we observe an output or a delay that is not specified.

[2] http://www.omg.org/spec/ALF/

6 Experiments

Coverage and path explosion problem. We consider *message coverage* which is one of the criteria defined in the literature for scenario models [2]. It states that any message must be covered at least once. In order to achieve coverage, Diversity allows to define exploration strategies. Classical search algorithms like Breadth First Search (BFS) are implemented. However, using BFS results in exploring a large number of paths in the symbolic tree which are irrelevant to the coverage criteria. We suggest using the heuristic search called *Hit-or-Jump* [5] which computes a symbolic tree covering a declared set of transitions. In our case, its is a set of transitions matching emissions/receptions of the messages in the sequence diagram. first we define a maximal depth N for which a symbolic tree is computed in BFS manner. Once computed, an analysis is realized to study whether or not a part of the tree satisfies the coverage: (*Hit*) If some non empty prefixes of the sequence has been covered, Diversity identifies the set of paths that covered the greatest prefix, and chooses one among them at random else Diversity chooses at random one path; (*Jump*) Once a path is chosen the whole process starts again from the last symbolic state of the path (i.e. the target state of the last symbolic transition of the path) until the sequence is fully covered. Another version of the Hit-or-Jump (and more accurate as in [5]) tries to cover a set of transitions rather than an enforced sequence which is useful in some cases when it is not easy to predict an appropriate sequence of messages as illustrated in figure 10.

Note that introducing timing constraints may constrain the FIFO size. Recall



(SET1)
$\{A!m_1, B?m_1, B!m_2, B!m_3, A?m_3\}$
(SEQ1) $A!m_1.B?m_1.B!m_3$
$.A?m_3.A!m_1.B?m_1.B!m_2$
(SEQ2) $A!m_1.A!m_1.B?m_1.B!m_3$
$.A?m_3.B?m_1.B!m_2$

Input ($N = 2$)	Hit-or-Jump	#Jumps
(SET1)	covered	$5N$
(SEQ1)	covered	$> 5N$
(SEQ2)	timeout	$>> 5N$

Fig. 10: Hit-or-Jump/sample sequence diagram.

that we associate to each message in the sequence diagram an unbounded FIFO buffer. In the sequence diagram of figure 10, the FIFO is similar to one-place buffer due to the timing constraints. Hence the sequence **(SEQ2)** can hardly be covered within a reasonable number of jumps regarding the size of the diagram (if the actions in the sequence must be consecutive, the Hit-or-Jump deadlocks). Let us consider again the sequence diagram of the RSW system. In Table 1, are given some metrics about the symbolic execution of its set of communicating TIOSTS using the heuristic Hit-or-Jump. *Failing behaviour.* While first testing the fUML activities of the RSW system w.r.t Requirement **(R2)**, Diversity delivered FAIL verdicts for some input sequences. The failure was due to intensity measures being ignored by the controller. Recall the fUML execution semantics

in Section 4. The activity of the controller while waiting for the speed from the calculator was: checking the event pool, reading intensity measures and ignoring them. To solve this problem which is caused by the event handling in the fUML virtual machine, a separate buffering mechanism for measures was successfully introduced in the activity model.

#TIOSTS	#States	#Transitions
7	81	192

Input	N	time	#Jumps
(SET2)	7	$5s$	$2N$
(SEQ3)	7	$22s$	$7N$
(SEQ4)	3	$9s$	$14N$

Requirement **(R3)/(SET2)**
$\{ctrl.intensity!m_2, \ldots, motor.speed?m_4\}$
Requirement **(R2)/(SEQ3)**
$ctrl.intensity?m_1 \ldots ctrl.intensity?m_1$
$\ldots ctrl.intensity?m_1$
Requirement **(R1)/(SEQ4)**
$calc.intensity?m_2.new(calc.speed).calc.speed!m_3$

Table 1: Hit-or-Jump/symbolic execution of the RSW system.

7 Related works

We can find in recent literature approaches [14,16] which have addressed conformance testing based on sequence diagram in the frame of the *ioco* relation (the untimed version of *tioco*). Authors in [16] use sequence diagrams in testing activities. They derive test cases expressed as sequence diagrams from state-based UML models guided by test objectives, also expressed as sequence diagrams. Authors in [14] have defined operational semantics for sequence diagrams where they handle in addition assertion and negative operators (*neg* and *assert*) for forbidden and required behaviors. However, they do not consider timing features in the test derivation algorithm. Let us discuss the approaches which deal with symbolic test generation from scenarios. Testing based on symbolic denotation of scenarios has been considered in [18] where scenarios are graphical MSCs (Message sequence chart) [12] like sequence diagrams. The test cases are experimented against the implementation within the frame of *ioco*. Outside *ioco* frame, symbolic techniques are used in [7] to generate test inputs from information contained in class diagrams and sequence diagrams. Transformation rules are defined to obtain a directed graph VGA (Variable Assignment Graph). The authors define coverage criteria for sequence diagrams in order to select relevant paths and use solvers to compute test inputs. This work is closely related to ours since they use likewise generated inputs to test an executable form of the design models, however they do not consider timing constraints. Our approach is compliant with the lately standardized fUML virtual machine to execute activity models and an ongoing standardization of the semantics of UML composite structures [10]. Formal verification of fUML executable models has been studied in [1, 17]. We rather focus on testing fUML models as in [6, 15]. In particular, authors in [15] set up a test environment with an interpreter to run test cases in the fUML virtual machine. Our work is more complete because we integrate test generation capabilities from sequence diagrams.

8 Conclusion

In this paper, we have presented an approach which aims at enhancing confidence in the correctness of wide system models through refinement. The refinement is based on maintaining a correctness relation which states sufficient conditions on conforming executable models with respect to high-level timed scenarios. Our approach is tooled and compliant with UML standards. In the future, we plan to integrate more refinement techniques as in [8] and extend them to timing issues.

References

1. I. Abdelhalim, S. Schneider, and H. Treharne. Towards a practical approach to check UML/fUML models consistency using csp. In *ICFEM*, 2011.
2. A. A. Andrews, R. B. France, S. Ghosh, and G. Craig. Test adequacy criteria for uml design models. *Softw. Test., Verif. Reliab.*, 2003.
3. B. Bannour, J. P. Escobedo, C. Gaston, and P. L. Gall. Off-line test case generation for timed symbolic model-based conformance testing. In *ICTSS*. Springer, 2012.
4. B. Bannour, C. Gaston, A. Lapitre, and J. P. Escobedo. Incremental symbolic conformance testing from UML MARTE sequence diagrams: railway use case. In *HASE*. IEEE, 2012.
5. A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-jump: An algorithm for embedded testing with applications to IN services. In *FORTE*. Springer, 1999.
6. F. Craciun, S. Motogna, and I. Lazar. Towards better testing of fUML models. In *ICST*, 2013.
7. T. T. Dinh-Trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test UML design models. In *ISSRE*. IEEE, 2006.
8. A. Faivre, C. Gaston, P. L. Gall, and A. Touil. Test purpose concretization through symbolic action refinement. In *TestCom/FATES*. Springer, 2008.
9. O. M. Group. A UML profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, VSL, 2009. http://www.omg.org/spec/MARTE/.
10. O. M. Group. Pscs: Precise semantics of uml composite structures, 2013. Second revised submission. To appear.
11. O. M. Group. Semantics of a foundational subset for executable uml models (fUML), 2013. http://www.omg.org/spec/FUML/.
12. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1997.
13. J.-C. King. A new approach to program testing. *Proc. of Int. Conf. on Reliable software*, 1975.
14. M. S. Lund and K. Stølen. Deriving tests from uml 2.0 sequence diagrams with neg and assert. In *AST*, 2006.
15. S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel. A framework for testing UML activities based on fUML. In *MoDeVVaMoDELS*, 2013.
16. S. Pickin, C. Jard, T. Jéron, J.-M. Jézéquel, and Y. L. Traon. Test synthesis from UML models of distributed software. *IEEE Trans. Software Eng.*, 2007.
17. E. Planas, J. Cabot, and C. Gómez. Lightweight verification of executable models. In *ER*, 2011.
18. A. Roychoudhury, A. Goel, and B. Sengupta. Symbolic message sequence charts. *ACM Trans. Softw. Eng. Methodol.*, 2012.
19. J. Schmaltz and J. Tretmans. On Conformance Testing for Timed Systems. In *FORMATS*. Springer, 2008.
20. J. Tatibouet, A. Cuccuru, S. Gerard, and F. Terrier. Principles for the realization of an open simulation framework based on fuml (WIP). In *DEVS*. ACM, 2013.