Deriving Interaction Scenarios for Timed Distributed Systems by Symbolic Execution

Boutheina Bannour[1](✉), Arnault Lapitre[1], and Pascale Le Gall[2]

[1] Université Paris-Saclay, CEA, List, 91120 Palaiseau, France
{boutheina.bannour,arnault.lapitre}@cea.fr
[2] Université Paris-Saclay, CentraleSupélec, MICS, 91192 Gif-sur-Yvette Cedex, France
pascale.legall@centralesupelec.fr

Abstract. In this paper, we propose a symbolic framework to analyze and debug communicating distributed models. We implement dedicated symbolic execution techniques for such models and use them to compute interaction scenarios satisfying a particular user coverage objective. These scenarios reveal emergent temporal and data correlations that are part of the system specification. To support the understanding and the analysis of such learned knowledge, our tooling allows for an intuitive annotated scenario visualization using sequence diagrams. As an application, we develop behavioral models for the so-called distributed Trickle algorithm which manages information dissemination in Wireless Sensor Networks (WSN). We select relevant scenarios which cover critical communications achieving an up-to-date or outdated state of the network.

1 Introduction

Context and Related Work. Symbolic Execution (SE in short) [10] is a powerful execution technique for analyzing programs or models. Its main principle is to execute a program or a model using variables instead of concrete values. SE collects for each explored path a set of logical constraints on the introduced variables for its execution, called Path Conditions (PC). Therefore, such a symbolic path represents a possibly infinite set of concrete paths. As such, SE is a systematic execution technique, which is valuable for the analysis of complex programs or models. Moreover, SE is gaining interest due to the availability of computing resources and the progress made in constraint solving.

SE has been increasingly used in the analysis of concurrent multi-process systems and more recently of distributed systems. Although we are positioning our work at the level of models of distributed systems, we will discuss related approaches dealing with programs [1,11,20,22,23,25] for their relevance and for having addressed early issues related to non-deterministic concurrent executions and/or asynchronous communications. In [1], authors use an object oriented modeling language for programs which allows objects to execute concurrently (or

© Springer Nature Switzerland AG 2021

synchronize) and to interact by asynchronous method calls. In [25], the Symbolic PathFinder [21] tool is extended to deal with bytecode of multi-threaded Java programs without communication primitives. Another work [18] still based on PathFinder proposes analysis of inter-process communications. In [11], concrete and symbolic execution are combined to provide an efficient analysis based on a fixed order of scheduling multi-threaded programs. In [22], authors implement SE of distributed systems on top of the KLEE tool [5] by unfolding a number of execution paths for some subsystems while keeping track of their communication history in terms of emitted/received network packets and then identifying pairwise states on packets emitting/recipient subsystems. While combinatorial interleaving is often bypassed by using exploration heuristic or partial order reduction techniques, the originality of this work is to come up with an alternative to interleaved execution of distributed subsystems. SE has been extended to models using variants of abstract labeled transition systems, namely Symbolic Transition Systems (STS in short) [6,12]. Timed STSs (or TSTS for short) which extend STS with clocks have been defined later in [3,26,27]. [26,27] are rather a merge of Timed Automata [2] and STS so that clock values represent convex abstractions, known as zones, while [3] uses symbolic techniques to homogeneously handle both clocks and other data variables. The DIVERSITY tool [17] implements the SE of TSTS [3,6] and also provides them with textual syntax and editing features.

Contribution. We adopt a compositional modeling approach based on TSTS like those in [3] for the purpose of modeling behaviors of distributed subsystems. A technical improvement is that we give a new definition of TSTS in which transitions are equipped with an expressive sequential statement language. This statement language mixes computations on clocks and other variables, and enables many steps of communications, guards and variable updates to be combined into a single (symbolic) execution step. The objective is to provide modeling facilities for different levels of detail that may appear in the specification of real distributed systems, such as in the specification of the Trickle case study [13,14,16] which has motivated the present work. We then define models of distributed systems as a collection of TSTSs. These are endowed with SE using asynchronous communication mechanisms for which we define time and data interdependencies. More precisely, our SE mechanisms collect all possible information linking data and time on the execution of distributed subsystems. Thus, we propose identification constraints to take into account that values received at a given port necessarily correspond to a value emitted at a port connected to it. Similarly, since a duration between an emission of a message a followed by a reception of a message b at a given location should be greater than the duration between the reception of a followed by the emission of b at a remote location, we consider constraints on delays separating events (emissions/receptions). As part of our contribution, we implement the proposed TSTS-based distributed models together with their SE in the DIVERSITY tool. Interactions between communicating subsystems can therefore be explored in the tool. In order to primarily assess these interaction behaviors, we propose a dedicated selection method that highlights the emissions and their corresponding receptions.

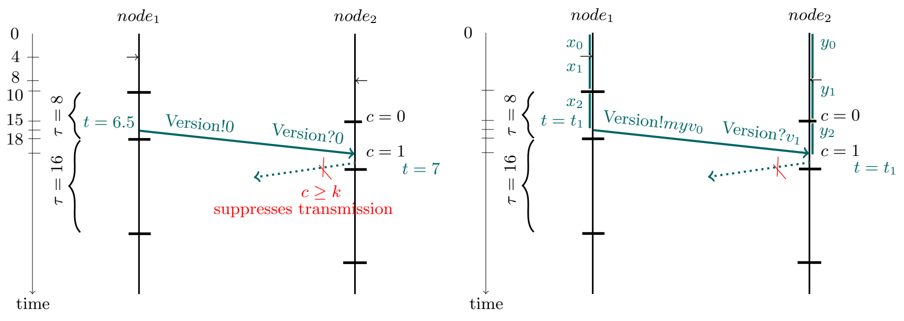
Structure of the Paper. Sect. 2 presents the motivating example of the Trickle-based information dissemination. Section 3 introduces TSTS and their associated SE mechanisms. In Sect. 4, we define TSTS-based models for distributed systems together with dedicated unfolding techniques applying correlated SE of the involved TSTS under asynchronous communication. Then, we show in the section how interaction scenarios can be selected from such models and discuss experimental results on the Trickle case study. Finally, we give some concluding remarks in Sect. 5.

2 Motivating Example: Trickle-Based Dissemination

Trickle [13,14,16] is the state-of-the-art distributed algorithm for the dissemination of information across a Wireless Sensors Network (WSN). This algorithm is provided as a standard library in TinyOS [15] and Contiki [8], two of the best known firmware Operating Systems (OS) for WSN. Trickle is also used in recently standardized WSN protocols namely the Multicast Protocol for Low Power and Lossy Networks (MPL) [9] and the Routing Protocol for Low Power and Lossy Networks (RPL) [28]. The dissemination of information across the network is through sensor-to-sensor short-range communications since such small devices can be equipped only with small radio antennas. In addition, such communications may be asymmetric: a node can send messages to another node without the opposite being possible. The network can be seen as a directed graph connecting nodes to their neighbors which can be reached by their transmissions. Trickle is a fully distributed algorithm. Each node applies a set of rules according to its state, i.e., the information it holds. The goal is to converge to an updated global state of the network where all nodes have the same information. The Trickle algorithm can be described as follows:

- each node maintains a current interval τ, a counter c and a broadcasting time t in current interval τ,
- global parameters to all nodes (same values) are k the redundancy constant, τ_l the smallest value for τ, and τ_h the largest value for τ,
- each node applies the following rules:
  1. at the start of a new interval a node resets its timer and counter c and sets at random t to a value in $[\tau/2, \tau[$,
  2. if the node receives a message consistent with the information it holds, it increments c,
  3. when its timer reaches t, the node broadcasts a message carrying the information it holds if $c < k$,
  4. when its timer expires at τ, it increases its interval length by setting τ to $\min(2 \cdot \tau, \tau_h)$ and starts a new interval,
  5. when a node receives a message that is inconsistent with its own information, then if $\tau > \tau_l$ it sets τ to τ_l and starts a new interval, otherwise it does nothing.

Each time an inconsistency is detected, the interval τ is set to τ_l, then τ is doubled up to τ_h. Note that the node transmits only if its neighbours are unlikely to be up-to-date, when $c < k$ given c counts receptions of consistent messages in the interval (k is fixed based on neighbours number). If $c \geq k$, the node suppresses the transmission. Moreover, since small intervals are considered immediately after the inconsistency, the frequency of transmissions is greater at the beginning (and decreases when approaching τ_h). This allows nodes to quickly share the same information. Now nodes are not necessarily synchronized, yet Trickle suggests choosing a random t (in $[\tau/2, \tau[$) together with imposing a listen-only period (first half of τ) in order to enhance the distribution of the transmission load between nodes in the interval (and hence energy cost).



(a) Concrete scenario (b) Symbolic scenario

$$PC_1 : \left(4 \leq node_1.t_1 < 8 \wedge x_2 = node_1.t_1\right)$$

$$PC_2 : \begin{pmatrix} 4 \leq node_2.t_1 < 8 \wedge y_2 \leq node_2.t_1 \\ \wedge\, node_2.v_1 = node_2.myv_0 \end{pmatrix}$$

data and time interdependencies :

$$node_1.myv_0 = node_2.v_1 \wedge x_0 + x_1 + x_2 \leq y_0 + y_1 + y_2$$

(c) Symbolic scenario constraints

Fig. 1. Illustration of Trickle scenarios

Figure 1 depicts a first scenario (Fig. 1a) between two nodes $node_1$ and $node_2$, each implementing the Trickle algorithm. The second scenario (Fig. 1b) is an abstract scenario that can be obtained by our symbolic execution for which the first scenario is a concretization. The scenarios illustrate the transmission suppress according to Trickle rules, the information being disseminated is a version number. Node $node_1$ emits a version number 0 on its port $node_1.Version$ (denoted by $Version!0$ on the message with source the axis of $node_1$). This version number is received later by node $node_2$ on its port $node_2.Version$ (denoted by $Version?0$ on the message with destination the axis of $node_2$). When $node_2$ receives the same version number as it holds, it increments its counter $node_2.c$ which reaches k ($k = 1$). This results in suppressing the transmission scheduled at the value stored in $node_2.t$ ($node_2.t = 7$) within the second half of the

τ-interval ($node_2.\tau = 8$). Executions at $node_1$ and $node_2$ are given by the following succession of delays and emissions/receptions:

$node_1$: “$4, 6, 6.5, \mathrm{Version!0}$” and $node_2$: “$8, 7, 5.5, \mathrm{Version?0}$”

Initial delays 4 and 8 are measured since a common fictitious time point denoted by 0 on the time axis. They correspond to elapsed time before respectively $node_1$ and $node_2$ are started. The small arrows indicate the start of the nodes on their respective axes. Introducing such delayed initialization is typical for distributed subsystems where there is no global clock. The symbolic scenario encodes such execution using (fresh) variables and logical constraints (Fig. 1c). Constraints $4 \leq node_1.t_1 < 8 \wedge x_2 = node_1.t_1$ and $4 \leq node_2.t_1 < 8 \wedge y_2 \leq node_2.t_1 \wedge node_2.v_1 = node_2.myv_0$ correspond to Path Conditions (PC). On the other hand, constraints $x_0 + x_1 + x_2 \leq y_0 + y_1 + y_2$ and $node_1.myv_0 = node_2.v_1$ reflect time and data inter-dependences. Both kind of logical constraints will be inferred by our symbolic execution mechanisms.

3 Models of Timed Symbolic Transition Systems

Preliminaries

A signature is a couple $\Omega = (S, Op)$ where S is a set of type names and Op is a set of operation names provided with a profile in S^+. We denote with $V = \coprod_{s\in S} V_s$ the set of typed variables with $type : V \to S$ the function that associates a variable to its corresponding type. The set of Ω-terms in V, denoted $\mathcal{T}_\Omega(V) = \coprod_{s\in S} \mathcal{T}_\Omega(V)_s$, is inductively defined over V and operations Op of Ω as usual and the function $type$ is extended to $\mathcal{T}_\Omega(V)$ as usual. The set of typed equational Ω-formulas over V, denoted as $\mathcal{F}_\Omega(V)$, is inductively defined over equality predicates $t = t'$ for any $t, t' \in \mathcal{T}_\Omega(V)_s$ and over usual boolean connectives. For $\varphi \in \mathcal{F}_\Omega(V)$, we denote $Var(\varphi) \subseteq V$ the set of variables occurring in φ. A substitution over V is a type-preserving application $\rho : V \to \mathcal{T}_\Omega(V)$. The identity substitution over V is denoted id_V and the notation $\rho[x \to t]$ means the substitution identical to ρ except that it associates t with x. Signature Ω includes a particular type, denoted $time \in S$, for representing durations, and which is provided with usual operations, i.e., $<: time \times time \to Bool$ and $+ : time \times time \to time, \dots$

An Ω-model is a set $M = \coprod_{s\in S} M_s$ with M_s a set of values for type s, hence inducing a mapping between each operation name $f : s_1, \dots, s_{n-1} \to s_n \in Op$ and the corresponding concrete operation $f_M : M_{s_1} \times \cdots \times M_{s_{n-1}} \to M_{s_n}$. The set M_{time} is denoted D (for the set of durations) and is isomorphic to the set of non-negative real numbers ($\mathbb{R}_{\geq 0}$). An interpretation is an application $\nu : V \to M$ that associates a value in M to each variable $v \in V$, canonically extended to $\mathcal{T}_\Omega(V)$ and $\mathcal{F}_\Omega(V)$ as usual. For $\nu \in M^V$ and $\varphi \in \mathcal{F}_\Omega(V)$, the satisfaction of φ by ν is denoted $M \models_\nu \varphi$ and is inductively defined w.r.t. the structure of φ as usual. We say a formula $\varphi \in \mathcal{F}_\Omega(V)$ is satisfiable, denoted $Sat(\varphi)$, if there exists $\nu \in M^V$ such that $M \models_\nu \varphi$.

A *TSTS-signature* is defined as a tuple $\Sigma = (\Omega, A, K, P)$, where $\Omega = (S, Op)$ is a signature, A, K and P are pairwise disjoint sets of variables representing respectively *generic data variables* (A), *clock variables* (K) and *ports* (P). Variables in A and P can take any type in S whereas variables in K (i.e. *clocks*) are limited to type *time*. So, each class of variables is partitioned w.r.t. types $s \in S$, hence $A = \coprod_{s \in S} A_s$, $P = \coprod_{s \in S} P_s$ whereas K can be reduced to K_{time}[1].

As glimpsed in the introduction, transitions of TSTS will be defined using sequential scheduling of statements like in programming languages.

Sequential Statements
A sequential statement (statement in short) *stm* is defined as follows:

$$\begin{array}{c} stm ::= \textbf{skip} \mid p?x \mid p!t_1 \mid x{:=}t_2 \mid \textbf{newfresh}(x) \mid [\phi] \mid \\ stm_1;stm_2 \mid \textbf{if}\,(\phi)\,\textbf{then}\,stm_1\,\textbf{else}\,stm_2 \end{array}$$

with $p \in P$, $x \in A \cup K$, $t_1 \in \mathcal{T}_\Omega(A \cup K)_{type(p)}$, $t_2 \in \mathcal{T}_\Omega(A \cup K)_{type(x)}$, and $\phi \in \mathcal{F}_\Omega(A \cup K)$. The set of statements over Σ is denoted by $Stm(\Sigma)$.

skip is the null statement; $p?x$ denotes the reception, on port p, of a value which is stored in x, whereas $p!t_1$ denotes the emission, on port p, of the value corresponding to the current interpretation of term t_1; $x := t_2$ assigns the variable x with a new value denoted by t_2; **newfresh**(x) randomly assigns x with a new value; $[\phi]$ denotes a condition on variables which enables the statement execution. Moreover, statements can be built using sequence (;) and condition (**if** . . . **then** . . . **else** . . .).

Timed Symbolic Transition Systems (TSTS)
A TSTS over a TSTS-signature $\Sigma = (\Omega, A, K, P)$ is a triple $\mathbb{G} = (Q, q_0, Tr)$, where Q is the set of states, $q_0 \in Q$ is the initial state and Tr is the set of transitions of the form (q, stm, q') with $q, q' \in Q$, and $stm \in Stm(\Sigma)$.

For a transition $tr = (q, stm, q')$, q (resp. q') is the source (resp. target) state of the transition. *stm* is called the statement of the transition. We use the notations $src(tr)$, $tgt(tr)$ and $stm(tr)$ to refer to q, q' and *stm* respectively. For $\mathbb{G} = (Q, q_0, Tr)$ a TSTS, we use $States(\mathbb{G})$, $initSt(\mathbb{G})$ and $Trans(\mathbb{G})$ to respectively refer to Q, q_0 and Tr.

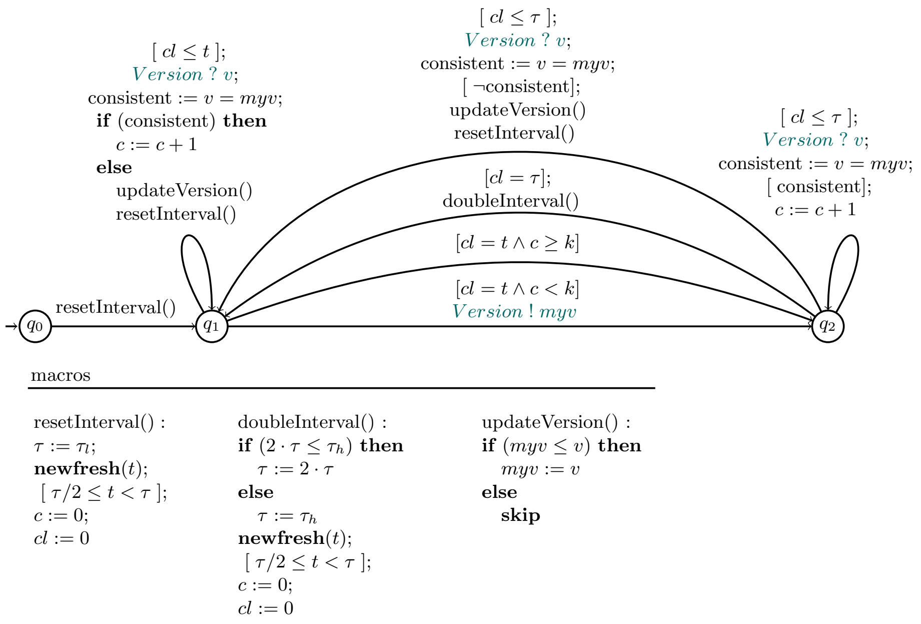
Figure 2 depicts a TSTS which models the behavior of a Trickle node, according to rules described in Sect. 2.

From the initial state q_0, the node assigns t with a new fresh value using the action **newfresh**(t), this new value for t is constrained by the guard $[\tau/2 \leq t < \tau]$. The clock[2] *cl* is initially reset, as glimpsed before it is used to activate the different Trickle events, that is when reaching t and starting new τ-intervals.

When node reaches the state q_1, it can receive a message (on port Version) from a neighboring node before reaching t (loop-transitions on q_1). Such message carries a version number. The message is processed as follows (see macro updateVersion): if the node is outdated, it updates its version number ($myv := v$). Trickle rule related to (in)consistency is then applied: if the neighbour is

[1] Variables in A_{time} are not of the same nature of those in K as they are only used to store terms of type *time*, while clocks are used to measure time passing.

[2] Any clock of a given node, here *cl*, evolves only if a transition of that node is executed: the clock is then implicitly incremented by a fresh duration.



```
resetInterval() :
$\tau := \tau_l$;
newfresh($t$);
 [ $\tau/2 \leq t < \tau$ ];
$c := 0$;
$cl := 0$

doubleInterval() :
if ($2 \cdot \tau \leq \tau_h$) then
   $\tau := 2 \cdot \tau$
else
   $\tau := \tau_h$
newfresh($t$);
 [ $\tau/2 \leq t < \tau$ ];
$c := 0$;
$cl := 0$

updateVersion() :
if ($myv \leq v$) then
   $myv := v$
else
   skip
```

Fig. 2. Timed Symbolic Transition Systems (TSTS) modeling a Trickle node

up-to-date, i.e., the node and its neighbour agree the version number, then the counter c of the node is incremented; otherwise a new τ-interval is considered for the node where τ and c are reset, since additional little-spaced messages are needed for convergence. When $cl = t$, the transmission is scheduled only if $c < k$ in order to inform neighbors of version number the node possesses (horizontal transition $q_1 \rightarrow q_2$), otherwise the process does nothing (curved transition $q_1 \rightarrow q_2$).

At the state q_2, while $cl \leq \tau$ and messages are consistent (agree on version number), the counter c is incremented (loop-transition on q_2), upon the reception of inconsistent message (different version number) the node starts a new τ-interval by resetting both cl and c (upper transition $q_2 \rightarrow q_1$). When $cl = \tau$, the node increases the listening interval by doubling τ (up to τ_h) (lower transition $q_2 \rightarrow q_1$).

Symbolic Execution of TSTS
Given a TSTS signature $\Sigma = (\Omega, A, K, P)$, we introduce a set of so-called *fresh variables* (also called symbolic parameters) $F = \bigcup_{s \in S} F_s$ which is disjoint from both the data and clock variables of Σ (i.e. $F \cap (A \cup K) = \emptyset$). We write $F_t \subseteq F$ for the set of fresh variables of type *time* and $F_d = F \backslash F_t$ for the set of non-time fresh variables. In the following, we will refer to the signature $\Sigma_F = (\Omega, F, \emptyset, P)$ to define the symbolic execution of a TSTS which is defined over Σ.

Execution contexts (EC in short) are data structures used to store information characterizing logical constraints on symbols related to an actual execution of a TSTS. An EC of a TSTS $\mathbb{G} = (Q, q_0, Tr)$ is a tuple $ec = (q, \pi, \lambda, \theta, Act, pec)$:

- $q \in Q$ is the (current) state of $\mathbb{G}$,
- $\pi \in \mathcal{F}_\Omega(F)$ is a constraint that should be satisfiable in order to reach ec.
- $\lambda : A \cup K \to \mathcal{T}_\Omega(F)$ is a type-preserving substitution through which data variables of $\mathbb{G}$ are replaced by terms over variables in F,
- $\theta \in \mathcal{T}_\Omega(F_t)$ is a term denoting the sum of durations elapsed so-far since the beginning of the execution of $\mathbb{G}$,
- Act is a sequence of emissions and/or receptions that have occurred since the beginning ($;;$ stands for the empty sequence),
- and pec is the parent EC which allowed ec to be reached.

We distinguish initial execution contexts of the form $ec_0 = (q_0, true, \lambda_0, z_0, ;;, self)$ such that λ_0 associates to each variable of $A \cup K$ a fresh variable in F verifying $\forall \theta \in K$, $\lambda_0(\theta) = z_0$ and $\forall x \neq y \in A$, $\lambda_0(x) \neq \lambda_0(y)$ and $\lambda_0(x) \neq z_0$."$self$" is a special identifier indicating that by convention, the parent EC of an initial execution ec_0 is itself. $Var(ec_0) \subseteq F$ is the set of all fresh variables occuring in ec_0. To refer to the constituents of an EC $ec = (q, \pi, \lambda, \theta, Act, pec)$, $q(ec)$ stands for q, $\pi(ec)$ for π, $\lambda(ec)$ for λ, $\theta(ec)$ for θ, $Act(ec)$ for Act and $pec(ec)$ for pec. We denote $\mathcal{EC}(\mathbb{G})$ ($\mathcal{EC}_0(\mathbb{G})$) the set of all (initial) execution contexts ec of $\mathbb{G}$ such that we have $Sat(\pi(ec))$. In order to take advantage that only some components of an EC are likely to be modified at each symbolic execution step, we will use a kind of additive notation which will point out the only modified components of ECs: for example, with ec an EC, $ec' = ec[Act \to Act; act]$ means that ec' coincides with ec except that the component $Act(ec')$ is $Act(ec); act$.

Symbolic Execution of Statements

Let stm and $\mathbb{G}$ be resp. a statement and a TSTS both defined over Σ and let $ec \in \mathcal{EC}(\mathbb{G})$. The set $SE(stm)_{ec} \subseteq \mathcal{EC}(\mathbb{G})$ of ECs reached from ec by symbolic execution of stm is defined as follows:

$$SE(\mathbf{skip})_{ec} = \{ec\} \tag{1}$$

$$SE(p?x)_{ec} = \{ec[\lambda \to \lambda[x \to y], Act \to Act; p?y]\} \quad y \in F_{type(x)} \tag{2}$$

$$SE(\mathbf{newfresh}(x))_{ec} = \{ec[\lambda \to \lambda[x \to y]]\} \quad y \in F_{type(x)} \tag{3}$$

$$SE(p!t_1)_{ec} = \{ec[Act \to Act; p!\lambda(t_1)]\} \tag{4}$$

$$SE(x := t_2)_{ec} = \{ec[\lambda \to \lambda[x \to \lambda(t_2)]]\} \tag{5}$$

$$SE([\phi])_{ec} = \{ec[\pi \to \pi \wedge \lambda(\phi)]\} \tag{6}$$

$$SE(stm_1; stm_2)_{ec} = \bigcup_{ec'_0 \in SE(stm_1)_{ec}} SE(stm_2)_{ec'_0} \tag{7}$$

$$SE(\mathbf{if}\,(\phi)\,\mathbf{then}\, stm_1\, \mathbf{else}\, stm_2)_{ec} = SE(stm_1)_{ec[\pi \to \pi \wedge \lambda(\phi)]} \cup SE(stm_2)_{ec[\pi \to \pi \wedge \lambda(\neg(\phi))]} \tag{8}$$

Symbolic Execution of Transitions
Let $tr = (q, stm, q')$ be a transition of $\mathbb{G}$ and $ec = (q, \pi, \lambda, \theta, Act, ec)$ be an EC of $\mathcal{EC}(\mathbb{G})$. The set of ECs $SE(tr)_{ec}$ reached from ec by symbolic execution of tr is defined as the set of all $ec' = (q', \pi', \lambda', \theta + z, Act', ec)$ such that there exists $(q, \pi', \lambda', \theta, Act', pec)$ in $SE(stm(tr))_{ec[\lambda \to \lambda'_0]}$ where λ'_0 is defined as follows:

$$\lambda'_0(w) = \lambda(w) + z \textbf{ for } w \in K \quad \textbf{and} \quad \lambda'_0(w) = \lambda(w) \textbf{ for } w \in A \tag{9}$$

with z a fresh variable in F_t, denoted as $delay(ec')$ in the sequel.

The set of ECs ec' reached from of a given EC ec by executing $tr \in Trans(\mathbb{G})$ is obtained in two steps. First, all clocks are increased by the same amount of time, denoted by the $time$-typed variable $z \in F_t$ which allows to obtain the intermediate substitution λ'_0 (9). Second, transition statement $stm(tr)$ is evaluated from the newly built context, i.e., ec in which λ'_0 replaces λ. ECs obtained then, characterize substitutions λ' reflecting data variables substitutions by new variables as defined by the statement symbolic execution.

Symbolic Execution of TSTS
The symbolic execution of a TSTS $\mathbb{G}$ from $ec_0 \in \mathcal{EC}_0(\mathbb{G})$ is the smallest set of ECs, denoted as $SE(\mathbb{G})_{ec_0}$, satisfying: $ec_0 \in SE(\mathbb{G})_{ec_0}$ and for any $tr \in Trans(\mathbb{G})$, $ec \in SE(\mathbb{G})_{ec_0}$ such that $q(ec) = src(tr)$, we have $SE(tr)_{ec} \subseteq SE(\mathbb{G})_{ec_0}$. This results in a tree structure en-rooted in ec_0.

4 Distributed Models and Interaction Scenarios Derivation

We use TSTS to specify Timed Distributed Systems (DS in short). A remote subsystem is characterized by a TSTS. Communications between subsystems TSTSs are specified by asynchronous data-passing over unbounded fifo queues.

Distributed System Model
A Distributed System model $Sys = (\mathbb{G}_1, \cdots, \mathbb{G}_m, \Gamma)$ is defined by:

- a family $(\mathbb{G}_i)_{i \in \{1,..m\}}$ of TSTS defined over $\Sigma_i = (\Omega, A_i, K_i, P_i)$ verifying that different sub-systems do not share constituents, i.e., $\forall i, j \leq m$ with $i \neq j$, $A_i \cap A_j = \emptyset$, $K_i \cap K_j = \emptyset$ and $P_i \cap P_j = \emptyset$,
- the function $\Gamma : P_{Sys} \to 2^{P_{Sys}}$ specifying connections between ports.

The set of all ports of Sys is denoted by $P_{Sys} = \bigcup_{i \leq m} P_i$. Besides, the set of all generic data variables (resp. clock variables) of Sys is denoted by $A_{Sys} = \bigcup_{i \leq m} A_i$ (resp. $K_{Sys} = \bigcup_{i \leq m} K_i$).

From now on, Sys will denote a generic DS model. We now define the symbolic execution of Sys over the signature $\Sigma_F^{Sys} = (\Omega, F, \emptyset, P_{Sys})$. Figure 3 depicts a DS for a grid topology of four Trickle nodes, each is associated with a TSTS.

System Execution Context. A *system execution context* (or a *SC*) of Sys is a tuple $ec_{sys} = (ec_1, \cdots, ec_m, \gamma, \chi, pec_{sys})$ where

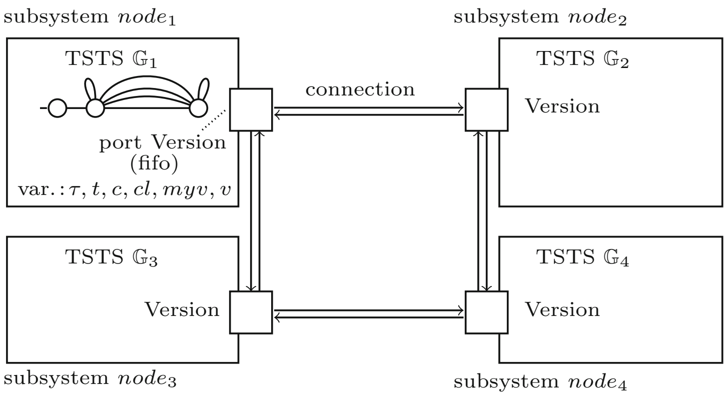


Fig. 3. Distributed System for a Trickle grid topology

- $ec_i \in \mathcal{EC}(\mathbb{G}_i)$ with $i \leq m$, is an EC over Σ_F^{Sys},
- $\gamma : P_{Sys} \to (\mathcal{T}_\Omega(F_d) \times \mathcal{T}_\Omega(F_t))^*$,
- $\chi \in \mathcal{F}_\Omega(F)$,
- and pec_{sys} is either the identifier $self$ or any other SC.

$\mathcal{EC}(Sys)$ is the set of all SC of Sys and we can access to components of ec_{sys} using notations $\gamma(ec_{sys})$,$\chi(ec_{sys})$ or $pec_{sys}(ec_{sys})$.

γ associates to each port the content of its fifo queue in terms of a received piece of data and its emission date. χ represents a constraint on time and data inferred from data exchanges in Sys. As in the unitary case, pec_{sys} gives access to the SC from which ec_{sys} has been built. Similarly, if $pec_{sys} = self$, ec_{sys} is then an initial SC. The symbolic execution of a DS-model consists in executing a transition of one of its component TSTS $\mathbb{G}_i$ and making execution contexts of Sys evolve accordingly. Intuitively, the evolution of the overall SC will essentially concern the EC relating to the TSTS $\mathbb{G}_i$ for which the transition is being executed.

Symbolic Execution of Transition in a DS-Model
For a $SC\, ec_{sys} = (ec_1, \cdots, ec_i, \cdots, ec_m, \gamma, \chi, pec_{sys})$, let tr be a transition in $\mathbb{G}_i$ such that $i \leq m$ and $src(tr) = q(ec_i)$. The symbolic execution $SE(tr)_{ec_{sys}}$ of tr from ec_{sys} is the set of $ec'_{sys} = (ec'_1, \cdots, ec'_i, \cdots, \gamma', \chi', ec_{sys})$ provided that all components of ec' are well defined according to the rules:

- $ec'_i \in SE(tr)_{ec_i}$
- for all $j \leq m$ such that $j \neq i$, $ec'_j = ec_j$
- $(\gamma', \chi') = SE(Act(ec'_i))_{(\gamma,\chi)}$ with $SE(Act)_{(\gamma,\chi)}$ inductively defined or undefined as follows:
  - for Act the empty sequence ($Act =;;$), (γ, χ)
  - for Act of the form $p!t; Act'$, $SE(Act')_{(\gamma[q \to \gamma(q).(t,\theta(ec'_i))]_{q \in \Gamma(p)}, \chi)}$

- for Act of the form $p?x; Act'$ s.t. there exists a port q verifying $p \in \Gamma(q)$,
  - $SE(Act')_{(\gamma[p \to w], \chi \wedge (\theta(ec_i') \geq \theta) \wedge (x=t))}$ if $\gamma(p)$ is of form $(t, \theta).w$
  - undefined if $\gamma(p) = \epsilon$ (i.e. $\gamma(p)$ is an empty fifo queue)
- for Act of the form $p?x; Act'$ s.t. for all ports q, $p \notin \Gamma(q)$, $SE(Act')_{(\gamma,\chi)}$.

When ec'_{sys} is defined, $ec(ec'_{sys})$ denotes ec'_i the execution context directly modified by the last executed transition ($tr \in Tr(\mathbb{G}_i)$). In a nutshell, an emission on a port p has the effect of filling all the fifo associated to the ports of $\Gamma(p)$ and the reception of a message on port p consumes the first message stored in its fifo. Simultaneously, all the knowledge about data and durations are translated into constraints. Let us note that for a transition tr carrying a reception on a port p connected to a port of other subsystems ($p \in \Gamma(q)$ for some q), if its fifo is empty ($\gamma(p) = \epsilon$), then $SE(tr)_{ec_{sys}}$ is undefined so that tr cannot be executed in the current SC.

Symbolic Execution of a DS-Model
Let us introduce ec^0_{sys} an initial SC defined as a tuple $(ec^1_0, \cdots, ec^m_0, \gamma_0, true, self)$ where for all $i \leq m$, ec^i_0 is in $\mathcal{EC}_0(\mathbb{G}_i)$ and for all p in P_{Sys}, $\gamma_0(p)$ is the empty queue, i.e. is ϵ. The symbolic execution $SE(Sys)_{ec^0_{sys}}$ of Sys is the smallest set of SC containing ec^0_{sys} and all SC reached by symbolic executions of any $tr \in \bigcup_{i \leq m} Trans(\mathbb{G}_i)$ from any $ec_{sys} \in SE(Sys)_{ec^0_{sys}}$. A system symbolic path pa_{sys} is a sequence $ec^0_{sys} ec^1_{sys} \ldots ec^k_{sys}$ such that for all $0 < i < k$, $pec_{sys}(ec^{i+1}_{sys}) = ec^i_{sys}$, we denote $tgt(pa_{sys}) = ec^k_{sys}$, the target of pa_{sys}.

For any $ec_{sys} = (ec_1, \cdots, ec_m, \gamma, \chi, pec_{sys})$, we note $\pi(ec_{sys}) = (\bigwedge_{i \leq k} \pi(ec_i)) \bigwedge \chi$.

A symbolic path pa_{sys} is feasible iff $Sat(\pi(tgt(pa_{sys})))$.

Selection Method
Interaction scenarios will be selected based on the computation of a feasible path pa_{sys} from the symbolic execution tree of the overall DS model Sys. The method favors the selection of system paths with a high coverage of subsystems pairwise Emissions (**E**) and Receptions (**R**) in the spirit of the work [24] and that of our previous experimental work on Trickle [4,19]. The idea is to guide the SE of Sys with the objective to compute a system path pa_{sys} which covers sequences where an emission ($p!a$) of a piece of data a by some subsystem is followed by the corresponding reception ($q?a$) of a by another subsystem ($q \in \Gamma(p)$). The selection is implemented according to the *Send Receive Pair Coverage criterion (SRPC)* [24] by defining coverage sequences based on internal pairwise Emissions and Receptions. To cope with the potential combinatorial explosion due to asynchrony, we have integrated the SRPC criterion with some exploration heuristics available in the tool DIVERSITY [17].

Experimentation. Table 1 gives some metrics on symbolic exploration for the computation of updated/outdated scenarios on grid topologies T1 (Fig. 3) and T2 (T2 has a connection less). The results are obtained on a PC equipped with an Intel Core i7 processor and 32GB RAM. The outdated scenario given in Fig. 4 is the one described on line 5 of Table 1. We report on the size of all

Table 1. Experimentation for Trickle

Scenario	Exploration	(|**E**|,|**R**|)	|SC|	Time	Coverage	Rate
Updated (T1)	Heuristic	(5, 11)	7239	1 m	100%	100%
Updated (T1)	BFS	(4, 9)	620120	1 h 0 m 34 s	72%	n/a
Outdated (T2)	Heuristic	(7, 15)	9156	23 s 650 ms	100%	100%
Updated (T2)	Heuristic	(8, 17)	13708	1 m 40 s 600 ms	100%	100%
Outdated (T1)	Heuristic	(3, 6)	1041	7 s 340 ms	100%	100%

|**E**|: Emission events count, |**R**|: Receptions events count in scenario, $|SC|$: count of System Contexts, Coverage: percentage of targeted coverage, Success rate for 20 trials of the selection heuristic

N.B., overall TSTSs size is 12 states and 28 transitions for both topologies T1 and T2

other generated scenarios in terms of number of Emissions and Receptions. We have used CVC4 solver to check the feasibility of logical constraints inferred for scenarios (system paths). We report on BFS (Breadth-First Search) in order to illustrate the exploration combinatorial. Even though left running for more than one hour, only partial coverage has been achieved with BFS. Then we have evaluated the coverage under the heuristic. Experiments show reduced running time. Given the overall short running time of our selection mechanism on the 4 asynchronous Trickle nodes, we expect it to be scalable to many other nodes. The reader can refer to [7,29,30] for some informative running time results on model-checking other Trickle case studies, yet in synchronous setting. Those first results need to be consolidated with further experiments.

Figure 4 depicts a sequence diagram of $node_4$ being outdated for some duration since the start of new version dissemination held by $node_1$ (Topology T1). The diagram has been adapted from the output generated by DIVERSITY. Neighbors of $node_4$, that are $node_2$ and $node_3$ are first updated by $node_1$ with a new version (green messages), they hence reset their interval to τ_l. After that, $node_4$ sends its version (old with respect of that of $node_1$) and gets them to reset their interval (blue messages). Thus, the transmissions of $node_2$ and $node_3$ are postponed; this gives time to $node_1$ re-transmit its version again and saturate their counters (orange messages); therefore they suppress their transmissions for $node_4$ respectively at $y_4 + y_5 = node_2.t_3$ and $z_4 + z_5 = node_3.t_3$ (last logical constraints in the scenario). Beyond highlighting such atypical scenarios, the symbolic execution of the model has allowed us to better understand the complex concurrency between nodes, which are ruled by non-trivial time constraints.

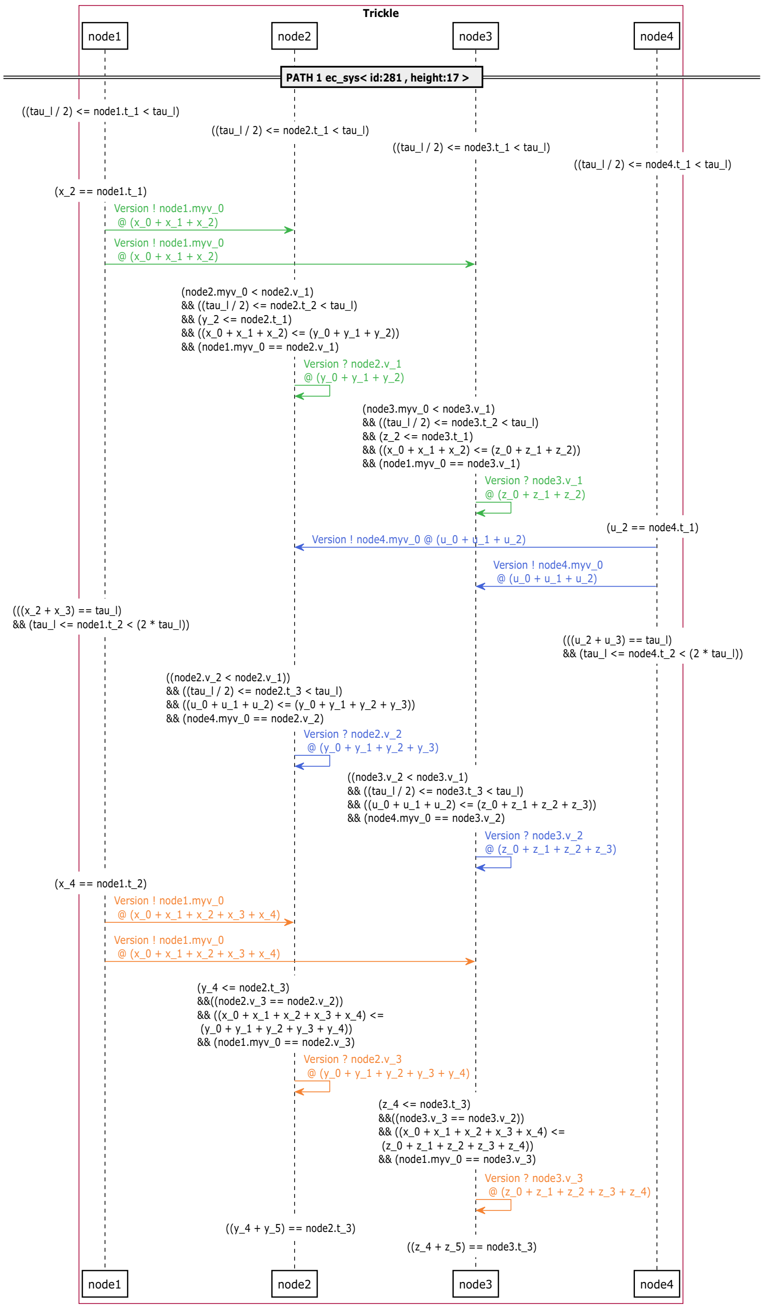

Trickle
node1
node2
node3
node4
PATH 1 ec_sys< id:281 , height:17 >
((tau_l / 2) <= node1.t_1 < tau_l)
((tau_l / 2) <= node2.t_1 < tau_l)
((tau_l / 2) <= node3.t_1 < tau_l)
((tau_l / 2) <= node4.t_1 < tau_l)
(x_2 == node1.t_1)
Version ! node1.myv_0 @ (x_0 + x_1 + x_2)
Version ! node1.myv_0 @ (x_0 + x_1 + x_2)
(node2.myv_0 < node2.v_1) && ((tau_l / 2) <= node2.t_2 < tau_l) && (y_2 <= node2.t_1) && ((x_0 + x_1 + x_2) <= (y_0 + y_1 + y_2)) && (node1.myv_0 == node2.v_1)
Version ? node2.v_1 @ (y_0 + y_1 + y_2)
(node3.myv_0 < node3.v_1) && ((tau_l / 2) <= node3.t_2 < tau_l) && (z_2 <= node3.t_1) && ((x_0 + x_1 + x_2) <= (z_0 + z_1 + z_2)) && (node1.myv_0 == node3.v_1)
Version ? node3.v_1 @ (z_0 + z_1 + z_2)
(u_2 == node4.t_1)
Version ! node4.myv_0 @ (u_0 + u_1 + u_2)
Version ! node4.myv_0 @ (u_0 + u_1 + u_2)
(((x_2 + x_3) == tau_l) && (tau_l <= node1.t_2 < (2 * tau_l))
(((u_2 + u_3) == tau_l) && (tau_l <= node4.t_2 < (2 * tau_l))
((node2.v_2 < node2.v_1)) && ((tau_l / 2) <= node2.t_3 < tau_l) && ((u_0 + u_1 + u_2) <= (y_0 + y_1 + y_2 + y_3)) && (node4.myv_0 == node2.v_2)
Version ? node2.v_2 @ (y_0 + y_1 + y_2 + y_3)
((node3.v_2 < node3.v_1)) && ((tau_l / 2) <= node3.t_3 < tau_l) && ((u_0 + u_1 + u_2) <= (z_0 + z_1 + z_2 + z_3)) && (node4.myv_0 == node3.v_2)
Version ? node3.v_2 @ (z_0 + z_1 + z_2 + z_3)
(x_4 == node1.t_2)
Version ! node1.myv_0 @ (x_0 + x_1 + x_2 + x_3 + x_4)
Version ! node1.myv_0 @ (x_0 + x_1 + x_2 + x_3 + x_4)
(y_4 <= node2.t_3) &&((node2.v_3 == node2.v_2)) && ((x_0 + x_1 + x_2 + x_3 + x_4) <= (y_0 + y_1 + y_2 + y_3 + y_4)) && (node1.myv_0 == node2.v_3)
Version ? node2.v_3 @ (y_0 + y_1 + y_2 + y_3 + y_4)
(z_4 <= node3.t_3) &&((node3.v_3 == node3.v_2)) && ((x_0 + x_1 + x_2 + x_3 + x_4) <= (z_0 + z_1 + z_2 + z_3 + z_4)) && (node1.myv_0 == node3.v_3)
Version ? node3.v_3 @ (z_0 + z_1 + z_2 + z_3 + z_4)
((y_4 + y_5) == node2.t_3)
((z_4 + z_5) == node3.t_3)
node1
node2
node3
node4

Fig. 4. A Trickle scenario for the topology of Fig. 3

5 Conclusion

We have provided a symbolic execution framework for timed distributed models fitted with feasible scenario selection mechanisms. Our framework has been completely implemented in the DIVERSITY tool and applied successfully on the distributed Trickle case study models using a heuristic approach. We believe that efficient novel partial order reduction techniques under the time setting -that we plan to develop in the near future- can leverage on this selection mechanism in order to accelerate the targeted selection for wide Trickle topologies.

References

1. Griesmayer, A., Aichernig, B., Johnsen, E.B., Schlatte, R.: Dynamic symbolic execution for testing distributed objects. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 105–120. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02949-3_9
2. Alur, R., Dill, D.: A theory of timed automata. J. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Bannour, B., Escobedo, J.P., Gaston, C., Le Gall, P.: Off-Line test case generation for timed symbolic model-based conformance testing. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 119–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34691-0_10
4. Bannour, B., Lapitre, A., Le Gall, P.: Exploring IoT trickle-based dissemination using timed model-checking and symbolic execution. In: Georgiou, C., Majumdar, R. (eds.) NETYS 2020. LNCS, vol. 12129, pp. 94–111. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67087-0_7
5. Cadar, C., Dunbar, D., Engler, R., Klee, D.: unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX (2008)
6. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006). https://doi.org/10.1007/11754008_1
7. Dong, J.S., Sun, J., Sun, J., Taguchi, K., Zhang, X.: Specifying and verifying sensor networks: an experiment of formal methods. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 318–337. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88194-0_20
8. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: LCN. IEEE (2004)
9. Hui, J., Kelsey, R.: Multicast protocol for low-power and lossy networks, request for comments: 7731. Technical report, Silicon Labs, February 2016
10. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**, 360248 (1976)
11. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_38
12. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES/RV -2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006). https://doi.org/10.1007/11940197_3

13. Levis, P., et al.: The emergence of a networking primitive in wireless sensor networks. Commun. ACM **51**(7), 99–106 (2008)
14. Levis, P., Clausen, T., Hui, J., Gnawali, O., Ko, J.: The trickle algorithm, request for comments: 6206. Technical report, Internet Engineering Task Force (IETF), March 2011
15. Levis, P., et al.: TinyOS: an operating system for sensor networks. In: Weber, W., Rabaey, J.M., Aarts, E. (eds.) Ambient Intelligence. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-27139-2_7
16. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: NSDI. USENIX Association (2004)
17. Arnaud, M., Bannour, B., Lapitre, A.: An illustrative use case of the DIVERSITY platform based on UML interaction scenarios. Electr. Notes Theor. Comput. Sci. **320**, 21 (2016)
18. Shafiei, N., Mehlitz, P.C.: Extending JPF to verify distributed systems. ACM SIGSOFT Softw. Eng. Notes **39**(1), 1–5 (2014)
19. Nguyen, N.M.T., Bannour, B., Lapitre, A., Le Gall, P.: Behavioral models and scenario selection for testing IoT trickle-based lossy multicast networks. In ICST Workshops. IEEE (2019)
20. Dustmann, S.O., Sasnauskas, R., Wehrle K.: Symbolic system time in distributed systems testing. In: ICST. IEEE (2012)
21. Pasareanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: ASE. ACM (2010)
22. Sasnauskas, R.S., Dustmann, O., Kaminski, B.L., Wehrle, K., Weise, C., Kowalewski, S.: Scalable symbolic execution of distributed systems. In: ICDCS. IEEE (2011)
23. Sasnauskas, R., Kaiser, P., Jukic, R.L., Wehrle, K.: Integration testing of protocol implementations using symbolic distributed execution. In: ICNP. IEEE (2012)
24. Robinson-Mallett, C., Hierons, R.M., Liggesmeyer, P.: Achieving communication coverage in testing. ACM SIGSOFT Softw. Eng. Notes **31**(6), 1–10 (2006)
25. Khurshid, S., PĂsĂreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-36577-X_40
26. Von Styp, S.C., Bohnenkamp, H., Schmaltz, J.: A conformance testing relation for symbolic timed automata. In: Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS Proceedings, pp. 243–255 (2010)
27. Andrade W., D. L. Machado P., Jéron T., Marchand H. Abstracting time and data for conformance testing of real-time systems. In: ICST Workshops. IEEE (2011)
28. Winter, T., et al.: Rpl: Ipv6 routing protocol for low-power and lossy networks, request for comments: 6550. Technical report, Cooper Power Systems and Cisco Systems and Stanford University (2012)
29. Woehrle, M., Bakhshi, R., Mousavi, M.R.: Mechanized extraction of topology antipatterns in wireless networks. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 158–173. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_12
30. Zheng, M., Sun, J., Liu, Y., Dong, J.S., Gu, Yu.: Towards a model checker for NesC and wireless sensor networks. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 372–387. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_26