MCMT in the Land of Parameterized Timed Automata

Alessandro Carioni
DTI, Università degli Studi di Milano (Italia)

Silvio Ghilardi
DSI, Università degli Studi di Milano (Italia)

Silvio Ranise
FBK-Irst, Trento (Italia)

Abstract

Timed networks are parametrised systems of timed automata. Solving reachability problems for this class of systems allows one to prove safety properties regardless of the number of processes in the network. Usually, these problems are attacked in the following way: the number *n* of processes in the network is fixed and a tool for timed automata (like Uppaal) is used to check the desired property for increasing values of *n*. In this paper, we explain how to deal with fully parametric reachability problems for timed networks by translation into the declarative input language of MCMT, a model checker for infinite state systems based on Satisfiability Modulo Theories techniques. We show the success of our approach on a number of standard algorithms, such as the Fischer protocol. Preliminary experiments show that fully parametric problems can be more easily solved by MCMT than their instances for a fixed (and large) number of processes by other systems.

1 Introduction

The land of timed networks is populated by families of networks with many identical timed processes parametrised by their set of identifiers. In addition, each network may have a central controller (i.e. a finite state automaton), capable of monitoring and affecting the timed processes.

Solving reachability problems (e.g., whether a set of unsafe states can ever be reached from the set of initial states) for this class of systems allows one to prove safety properties regardless of the number of processes in the network. The difficulty in solving this kind of verification problems is two-fold. First, each process has (at least one) clock variable ranging over an infinite set, such as the reals or the integers. Second, every system is parametrised with respect to the number of processes and the topology of the network. Hence, there are two dimensions along which these systems are infinite state.

Surprisingly, it is possible to show that the reachability problem for restricted classes of parametrised timed networks is decidable under suitable assumptions (e.g., on the number of clock variables or on the granularity of time) [4, 5] by a backward reachability procedure. Despite these theoretical results, there are few systems capable of automatically solving parametrised and timed reachability problems. Instead, the number *n* of processes in the network is fixed (for increasing values), a single timed automaton is built by parallel composition, and then a tool like Uppaal for timed automata is used to check the desired property for the given *n*.

In this paper, we explain how to solve fully parametric and timed reachability problems by translation to the declarative input language of MCMT, the Model Checker Modulo Theories which we currently develop and which is based on Satisfiability Modulo Theories (SMT) techniques. SMT solvers have already proved to be quite effective for solving problems resulting from the encoding of large verification problems as witnessed by the SMT-LIB initiative (`http://www.SMT-LIB.org`); MCMT uses Yices (`http://yices.csl.sri.com`) as the back-end SMT solver.

We show the practical viability of our techniques on a number of standard algorithms, such as the Fischer protocol which we use as the running example to illustrate our ideas. Our preliminary experiments show that fully parametric problems can be more easily solved by MCMT than instances for a fixed (and large) number of processes by Uppaal or similar systems. We believe that this is due to the capability of MCMT of discovering and exploiting the symmetries in the network by quantifiers instantiation techniques.

2 Timed Networks

Informally, a 'timed network with $k > 0$ clocks' is a system formed by a controller and finitely many identical timed processes, each one equipped with k clocks. The controller is a finite state automaton and timed processes are also finite state automata extended with k clock variables. There are two kinds of transitions: one modelling the passing of time (specified by incrementing all the clocks of the same amount of time) and another one in which the controller and a given number of timed processes (usually 1 or 2) synchronize and change their states simultaneously. Transitions of the second kind are guarded by conditions on the states of the controller, on the timed processes that are involved, and on the values of their clocks. If the guard is satisfied, the states of the controller and the involved processes are updated and the value of some clocks is reset, if the case. Initially, the controller and all the timed processes are in a distinguished initial state and their clocks have value 0. The value of the clocks is always positive and ranges over either $\mathbb{R}$ (in the continuous case) or $\mathbb{N}$ (in the discrete case).

This notion of parametrised timed network is first formalized in [4]. In the rest of this section, we recast such a notion into the formal framework of [13] underlying the infinite state model checker MCMT, which will be used to solve parametrised reachability problems of timed networks. Such networks can be modelled as a subclass of guarded assignment transition systems whereby state variables are functions mapping a subset of the natural numbers (used as identifiers of the processes in the network) to either a finite set of locations—the finitely many states of an automaton—or an infinite set of time points—the values of clocks. (Readers familiar with [13] will recognize an instance of the notion of array-based systems.)

Background notions. Formally, we work in many sorted first-order logic. We consider a signature Σ containing a sort `INDEX` for process identifiers, a sort `LOC` for control locations, and a sort `T` for time points. Furthermore, Σ contains two sort symbols `ARRAY_LOC` and `ARRAY_T` for arrays mapping indexes to locations and to time points, respectively. The function symbols $\_[\_]_{\mathtt{LOC}} : \mathtt{ARRAY\_LOC}, \mathtt{INDEX} \rightarrow \mathtt{LOC}$ and $\_[\_]_{\mathtt{T}} : \mathtt{ARRAY\_T}, \mathtt{INDEX} \rightarrow \mathtt{T}$ are also in Σ and denote the usual array dereferencing operations. (By abuse of notation, we will simply write $\_[\_]$ instead of $\_[\_]_{\mathtt{LOC}}$ or $\_[\_]_{\mathtt{T}}$.) Finally, Σ also contains a finite set $L = \{l_1, \ldots, l_m\}$ of constants of sort `LOC` ($m \geq 1$),[1] the set of numerals $0, 1, ...$ of sort `T`, and the function symbols $+, -$ of sort $\mathtt{T}, \mathtt{T} \rightarrow \mathtt{T}$, and the predicate symbols $<, \leq, >, \geq$ of sort `T,T`. Variables of sort `INDEX` will be written as $z, z_1, z_2, z_3, \ldots$.

Semantically, we shall restrict our considerations to Σ-structures such that (i) `T` is interpreted as either $\mathbb{R}$ or $\mathbb{N}$ and $+, -, <, \leq, >, \geq$ (as well as the numerals) have their natural meaning; (ii) `LOC` is interpreted as the finite set $L = \{l_1, \ldots, l_m\}$ and each l_i is interpreted as itself (for $i = 1, ..., m$); (iii) `INDEX` is interpreted as a *finite* subset of the natural numbers; (iv) `ARRAY_T` and `ARRAY_LOC` are interpreted as the set of functions from the interpretation of `INDEX` to the interpretation of `T` and `LOC`, respectively, and $\_[\_]$ is interpreted as function application. According to the SMT-LIB standard (`http://www.smt-lib.org`), a pair comprising a signature Σ and a class of Σ-structures (also called *models*) identifies a *theory*: the theory described above will be called TPN in the rest of the paper. When referring to *assignments, satisfiability* and *truth* we always mean assignments, satisfiability and truth in one of the models of TPN.

[1]To simplify notation, we use a unique sort comprising both controller's and processes' locations; this is w.l.o.g. since transitions can be written so that different kinds of locations are not mixed up.

Formal specification of timed networks. A *timed network with k clocks* is a tuple

$$\langle I(\sigma,\underline{C},s),\{T_i(\sigma,\sigma',\underline{C},\underline{C}',s,s')\}_i,D(\sigma,\sigma',\underline{C},\underline{C}',s,s')\rangle,$$

where σ is a state variable of sort `LOC` for the controller, s is a state variable of sort `ARRAY_LOC` for the states of the timed processes, and $\underline{C}=(C_1,\ldots,C_k)$ is a tuple of state variables, each one of sort `ARRAY_T`, for the k clocks of the timed processes. As usual, the primed versions of the variables denote the values of the state variables after the execution of a transition. The *initial formula* $I(\sigma,\underline{C},s)$ is of the form

$$\sigma=m_0\wedge\forall z(s[z]=l_0\wedge\bigwedge_{j=1}^{k}C_j[z]=0)$$

where l_0,m_0 are location constants, i.e. they are of sort `LOC`.

Before describing the other two components of the tuple, we need to introduce the following notion. A *constraint* is a conjunction of literals of the following forms: $t >> c$, $d >> u$, and $v=l$, where $>>\in\{>,\geq\}$, c,d are numerals, l is a constants of type `LOC`, t,u are terms of the form $C_i[z_j]$, and v is σ or a term of the form $s[z_j]$. A *difference logic constraint* is a constraint comprising also literals of the form $t-u>>c$ and $d>>t-u$, for t,u,c,d as above.

The *step transition formula* $T_i(\sigma,\sigma',\underline{C},\underline{C}',s,s')$ is of the form

$$\exists z_1\cdots z_n.\bigwedge_{i<j}z_i\neq z_j\wedge\left(\begin{array}{l}\phi_G(\sigma,s[z_1],\ldots,s[z_n],\underline{C}[z_1],\ldots,\underline{C}[z_n])\wedge\\ \wedge s'=\lambda z.F_S(z)\wedge\sigma'=m\wedge\bigwedge_j C'_j=\lambda z.F_j(z)\end{array}\right),$$

for $i\geq 1$, where ϕ_G is a constraint, the *update functions* F_S,F_j are specified by nested 'if-then-else' expressions as follows:[2]

$$\begin{array}{rcl}F_S(z)&:=&\texttt{if}\ z=z_1\ \texttt{then}\ l_1\ \texttt{else}\ \texttt{if}\ z=z_2\cdots\ \texttt{then}\ l_n\ \texttt{else}\ s[z]\\ F_j(z)&:=&\texttt{if}\ z=z_1\ \texttt{then}\ t_{1j}\ \texttt{else}\ \texttt{if}\ z=z_2\cdots\ \texttt{then}\ t_{nj}\ \texttt{else}\ C_j[z],\end{array}$$

and t_{ij} is either 0 or $C_j[z_i]$. Formula T_i above means that (i) $z_1,\ldots,z_n$ are the processes that are going to synchronize (in the actual implementation of MCMT, it must be $1\leq n\leq 2$; this does not seem too restrictive as all timed networks considered in the examples satisfy this limitation), (ii) the constraint ϕ_G must hold for the transition to fire (ϕ_G involves the locations of the controller and of the timed processes together with the values of their clocks), (iii) during the transition the controller moves its location to m and the synchronizing processes to $l_1,\ldots,l_n$ (respectively), (iv) some of the clocks of the synchronization processes are reset, and finally (v) non synchronizing processes do not change location and clocks values.

Finally, the *delay transition formula* $D(\sigma,\sigma',\underline{C},\underline{C}',s,s')$ is of the form

$$\exists\varepsilon>0\ (\sigma'=\sigma\wedge\bigwedge_j C'_j=\lambda z.(C_j[z]+\varepsilon)\wedge s'=s)$$

where ε is a variable of sort `T`, expressing that clocks are incremented by $\varepsilon>0$ time units and that locations do not change in the meantime. A remark is in order. Both step and delay transition formulae can be written entirely in first-order logic by using universal quantifiers. We use here λ-abstractions because formulae are more compact and it is easier to grasp their intuitive reading.

Let $TN:=\langle I(\sigma,\underline{C},s),\{T_i(\sigma,\sigma',\underline{C},\underline{C}',s,s')\}_i,D(\sigma,\sigma',\underline{C},\underline{C}',s,s')\rangle$; the *states* of TN are the *assignments* to the variables σ, s, and $\underline{C}$.

[2] Along the lines of the SMT-LIB format, we extend many-sorted first-order logic with conditional expressions.

Extensions. In the literature, there exist several variations and generalizations of the notion of parametrised timed networks introduced in [4] and formalized above in our framework. For example, delay transitions can be extended with *location invariants*, that can be used to force a process 'to react,' i.e. to leave a certain location before a fixed amount of time has passed. Fortunately, this extension—as well as many others—can be accommodated in the formal framework of MCMT. In fact, the formula $D(\sigma,\sigma',\underline{C},\underline{C}',s,s')$ for delay transitions with location invariants have the following form:

$$\forall z.Inv(\sigma,s[z],\underline{C}[z],\underline{C}'[z]) \wedge \exists \varepsilon > 0(\sigma' = \sigma \wedge \bigwedge_j C' = \lambda z.(C_j[z]+\varepsilon) \wedge s' = s),$$

where *Inv* is a boolean combination of constraints involving both locations and clocks. While the declarative framework underling MCMT is capable of easily supporting delay transition with location invariants, some care must be taken to handle the universally quantified condition $\forall z.Inv$, also called *global* condition. In fact, it is well-known (see, e.g., [3]) that systems containing transitions with global conditions are difficult to analyze automatically and some form of approximation is needed. In MCMT, global conditions are handled by adopting the *stopping failures* model [15] (similar to the 'approximate model' in [3, 2]). The stopping failures model of the system satisfies a subset of the class of safety properties satisfied by the original system (since the latter has fewer runs), hence establishing a safety property for the stopping failures model implies that the same property is enjoyed by the original system. MCMT displays an explicit warning when it adopts the stopping failures model. (For a detailed treatment of the stopping failures model in MCMT, the reader is pointed to [11].)

Another extension which is crucial for modelling benchmarks concerns step transitions. In fact, it is often the case that *broadcasts* must be modelled so that also an unknown number of the timed processes not involved in the synchronization of a step transition can change their locations or even reset some of their clocks. This kind of transitions (as well as more general ones with non-deterministic updates) can be specified in the framework of MCMT by a careful definition of the update functions F_S and F_j using nested 'if-then-else' expressions. Finally, we mention that the framework underlying MCMT can also support the specification of timed networks where the controller has its own clocks or data variables (even ranging over unbounded sets, such as process identifiers).

We hope that this discussion suggests the flexibility of the framework underlying MCMT, which makes it capable of handling verification problems *independently* of any fixed formal model (such as the extended automata used in [5]). A complete account of the formal framework can be found in [13].

3 Reachability Problems for Timed Networks

The controller reachability problem is one of the most relevant verification problems for timed networks and can be stated as follows: is there a run (i.e. a sequence of transitions) of the network leading the controller to a certain control location q? Let $TN := \langle I(\sigma,\underline{C},s), \{T_i(\sigma,\sigma',\underline{C},\underline{C}',s,s')\}_i, D(\sigma,\sigma',\underline{C},\underline{C}',s,s')\rangle$ be a timed network with k clocks. A *run* of length n of *TN* is an assignment satisfying the following formula (we use T for the disjunction $D \vee \bigvee_i T_i$):

$$I(\sigma_0,\underline{C}_0,s_0) \wedge T(\sigma_0,\sigma_1,\underline{C}_0,\underline{C}_1,s_0,s_1) \wedge \cdots \wedge T(\sigma_{n-1},\sigma_n,\underline{C}_{n-1},\underline{C}_n,s_{n-1},s_n),$$

where $\sigma_0,\underline{C}_0,s_0,\ldots,\sigma_n,\underline{C}_n,s_n$ are $n+1$ renamed copies of $\sigma,\underline{C},s$. The controller reachability problem for *TN* amounts to checking the satisfiability (modulo the theory TPN, see the '*Background notions*' of Section 2 for a definition) of the following formula:

$$I(\sigma_0,\underline{C}_0,s_0) \wedge \bigwedge_{r=0}^{n-1} T(\sigma_r,\sigma_{r+1},\underline{C}_r,\underline{C}_{r+1},s_r,s_{r+1}) \wedge \sigma_n = q,$$

for some $n \geq 0$. In general, the problem is undecidable (already for $k = 2$, i.e. for two clocks per process [5]), but it turns out to have many decidable instances; e.g., it is decidable when $k = 1$ (i.e. when timed processes have just one clock) [5] or the sort `T` is interpreted as $\mathbb{N}$, regardless of the number k of clocks per process [16].

The decidability results mentioned above are obtained by using a *backward reachability procedure* to explore the (infinite) state space of a timed network and proving that the procedure terminates. Potentially infinite sets of states are finitely represented by so-called 'configurations': if these and the transitions of a class of systems enjoy certain properties (namely, being well-quasi-ordered), then backward reachability always terminate [1, 9, 7]. Backward reachability is also the starting point of the implementation of automated verification systems for classes of infinite state systems (see, e.g., [2, 3]) where configurations are encoded by *ad hoc* data structures, one for each class of systems (like broadcast protocols, lossy channels, or timed networks). Instead, MCMT uses particular classes of first-order formulae to represent configurations parametrised with respect to two theories, one for data and one for the topology so that backward reachability can be implemented by a fixed set of logical manipulations and the capability of solving SMT problems in the combination of the theories of indexes and of the topology. Below, we give a high-level description of the backward reachability procedure implemented in MCMT when the background theory is TPN.

Deductive backward reachability. Backward reachability repeatedly computes the pre-image of the set of states from which it is possible to reach the set of unsafe states, i.e. the states violating the desired safety property. The algorithm halts in two cases, either when the current set of (backward) reachable states has a non-empty intersection with the set of initial states and the system is unsafe, or when such a set has reached a fix-point (i.e. further application of the transition does not enlarge the set of reachable states) and the system is safe. To mechanize this procedure in the framework of MCMT, we need to ensure that (a) the class of formulae representing sets of states is closed under pre-image computation and (b) the checks for safety and fixed-point can be reduced to decidable SMT problems.

Sets of states are represented by $\exists^I$-formulae. An *$\exists^I$-formula* is a formula of the form $\exists z_1 \cdots z_n \phi$, where ϕ is obtained from a quantifier free formula $\psi(z_1, \ldots, z_n, \underline{t}, \underline{v})$ by replacing the free variables $\underline{t}$ of type `T` with the terms $C_i[z_1], \ldots, C_i[z_n]$ (for $i = 1, \ldots, k$) and the free variables $\underline{v}$ of type `LOC` with the terms $\sigma, s[z_1], \ldots, s[z_n]$. Notice that n can be 0, like in the formula $\sigma = q$, specifying that the controller state is equal to the control location q.

The *pre-image* of a formula $\phi(\sigma, \underline{C}, s)$ where at most the variables $\sigma, \underline{C}, s$ occur, is defined to be the formula

$$\exists \sigma' \, \exists \, \underline{C}' \, \exists s' \, (T(\sigma, \sigma', \underline{C}, \underline{C}', s, s') \wedge \phi(\sigma', \underline{C}', s')).$$

Concerning requirement (a) above, the following result is crucial.

Proposition 3.1. *The pre-image of an $\exists^I$-formula is equivalent to an $\exists^I$-formula modulo* TPN *(i.e. in the class of models being considered).*

Proposition 3.1 is a special case of a general result proved in [10]. Computing a pre-image w.r.t. a delay transition requires a quantifier elimination step in $\mathbb{N}$ or $\mathbb{R}$, depending on the interpretation of `T`, of the time variable ε. It is possible to refine Proposition 3.1 and show closure under pre-image computation when the $\exists^I$-formula is of the form $\exists z_1 \cdots z_n . (\bigwedge_{i<j} z_i \neq z_j \wedge \xi)$, where ξ is a difference logic constraint. Among the many advantages of this restricted format for $\exists^I$-formulae, actually supported by MCMT, one of the most important is the dramatic simplification of the pre-image computation w.r.t. a delay transition. In fact, there is no need to handle the Boolean structure while eliminating the existentially

quantified variable ε and the standard Fourier-Motzkin algorithm can be used both in the real and the integer case, as observed for instance in [18, 6].

Notice that one can employ (disjunctions of) $\exists^I$-formulae (in the restricted format) as an unsafety formula, instead of the controller reachability formula $\sigma = q$, without affecting the above considerations. As a consequence, we will consider below unsafety problems in this more liberal format.

For $n \geq 0$, the formulae $BR_n(\sigma, \underline{C}, s)$ are defined as follows: BR_0 is $\sigma = q$ and BR_{n+1} is the disjunction between BR_n and its pre-image. The sequence $BR_0, BR_1, ..., BR_n$ of formulae is computed until either (i) $I \wedge BR_n$ is satisfiable modulo TPN (*safety check*) or (ii) $I \wedge BR_n$ and $\neg(BR_{n+1} \rightarrow BR_n)$ are both unsatisfiable modulo TPN (*fix-point check*). In the former case, the controller location q is reachable whereas in the second case a fixed point has been obtained and q is unreachable. In case (i), the backward reachability procedure returns 'unsafe,' whereas in case (ii) it returns 'safe' (of course, there is a third possibility, namely that the algorithm diverges, which might happen if the problem is not in a class for which termination is guaranteed).

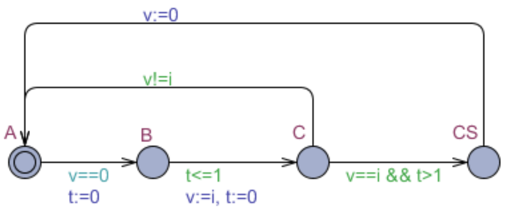
Concerning issue (b) above, we observe that fix-point and safety checks are discharged in MCMT by using the SMT solver Yices on formulae which contain universal quantifiers. Unfortunately, the performances of Yices and most of the other state-of-the-art SMT solvers on formulae involving quantifiers are somewhat unsatisfactory. Hence, we have implemented a dedicated quantifier instantiation procedure in MCMT for the class of quantified formulae arising in safety and fix-point checks. It is not difficult to see that such formulae can be transformed into the following form: $\exists z_1 \cdots z_n \forall y_1 \cdots y_m \phi$, where ϕ is quantifier-free. We call this class of formulae *$\exists\forall$-formulae.*

Proposition 3.2. *Checking the satisfiability of $\exists\forall$-formulae modulo* TPN *is decidable.*

Proposition 3.2 is an instance of a more general result in [10] and looks very similar to the decidability of the well-known $\exists^*\forall^*$-fragment of first-order logic. The main difference is that satisfiability here is restricted to the class of models of the theory TPN (i.e. we consider only those models that make sense for our model-checking problems). Proposition 3.2 is the basis of a three-step decision procedure for the satisfiability of $\exists\forall$-formulae (modulo TPN). First, the z_i's are Skolemized away. Second, the y_j's are instantiated in all possible ways with the z_i's (considered as free constants). Third, the resulting formula is quantifier-free and can be sent to the SMT solver (in the case of MCMT, Yices). Notice that the second step is computationally very expensive; hence, powerful heuristics to avoid useless instantiations are needed [12].

4 A worked-out Example

To illustrate our approach, we present the verification of the mutual exclusion property of the parametric and real-time protocol proposed by Fischer in the variant described in [5]. This protocol is considered a standard benchmark to measure the performance of verification tools for timed automata by considering an increasing number of processes. Here we prove that the mutual exclusion property is satisfied regardless of the number of processes in the network. Other versions of this protocol have been specified and verified by our tool: one extracted from the SAL distribution (`http://sal.csl.sri.com`), one from the Uppaal distribution (`http://www.uppaal.com`), and a third one taken from Chapter 13 of [17]. The alternative formulations exploits the flexibility of MCMT input language as they require transitions with location invariants and unbounded global (i.e. controller's) variables (see paragraph '*Extensions*' in Section 2). The results of our tool on the problem described in this section and its variants (as well as more problems) are listed in Table 3 of Section 5.1 with a comparison with Uppaal for some (increasing) number of processes.



Legend: t is the local clock value and v is the shared variable

Figure 1: Extended automaton for one process of the Fischer's protocol

Informal description. The goal of Fischer protocol is to ensure mutual exclusion in a network of processes whose number is arbitrary, using a clock and a shared variable. Each process has a local clock and a control state variable ranging over values in the set $\{A, B, C, CS\}$. Each process is identified by a natural number strictly bigger than 0 and can read/update a shared variable whose values is either 0 or the index of one of the processes. A process wishing to enter the critical section CS starts in the initial state A. If the value of the shared variable is 0, the process goes to state B and resets its local clock. From B, the process can go to state C if the clock value is still less than 1 time unit and it sets the value of the shared variable to its own index and again resets its clock. From C, the process can go to CS if the clock is strictly more than 1 time unit and the value of the shared variable is still equal to the index of the process performing the transition. When exiting CS, the process sets the value of the shared value to 0. The set of unsafe states, i.e. those states violating the mutual exclusion property, can be characterized by the presence of at least two processes which enter CS at the same time. This specification is depicted in Figure 1.

Formalization in MCMT. We follow the formal specification of the Fischer's protocol done in [5] to illustrate that our declarative framework can precisely describe the notion of timed network as given in [5]. However, as already mentioned above, we emphasize that our specification language is more flexible and it is also able to formalize the original specification of the protocol as well as other extensions.

We assume $\mathtt{T}$ to be $\mathbb{R}$ and $\mathtt{LOC} := \mathtt{LOC}_C \cup \mathtt{LOC}_P$ where $\mathtt{LOC}_C := \{udf, df\}$ and $\mathtt{LOC}_P := \{A, B, C, CS, A^+, B^+, C^+, CS^+\}$. The controller state is either udf, indicating that the value of the shared variable is 0 (undefined), or df, indicating that the value of the shared variable is not 0 (defined). The locations marked with $+$ correspond to configurations where the value of the shared variable is equal to the index of that particular process. We need three state variables: one shared variable c for the controller state ranging over $\mathtt{LOC}_C$, one local variable $q : \mathtt{ARRAY\_LOC}_P$ for the state variables of the processes, and one local variable $t : \mathtt{ARRAY\_T}$ for the processes clocks (notice that $k = 1$, i.e. we have just one clock per process).

The set of initial states can be characterized as follows:

$$c = udf \wedge \forall z.(q[z] = A \wedge t[z] = 0), \tag{1}$$

i.e., at the beginning, the control state variable c is undefined, the control state variable of each process is A, and the clock value of each process is 0.

The set of unsafe states is the disjunction of the following three $\exists^I$-formulae:

$$\exists z_1, z_2. \quad (z_1 \neq z_2 \wedge \quad q[z_1] = CS \wedge q[z_2] = CS) \tag{2}$$
$$\exists z_1, z_2. \quad (z_1 \neq z_2 \wedge \quad q[z_1] = CS^+ \wedge q[z_2] = CS^+) \tag{3}$$
$$\exists z_1, z_2. \quad (z_1 \neq z_2 \wedge \quad q[z_1] = CS^+ \wedge q[z_2] = CS), \tag{4}$$

i.e., the set of unsafe states consists of all those states where at least two processes are either both in CS, formula (2), or both in CS^+, formula (3), or one in in CS and one in CS^+, formula (4).

	τ
initiate	$\exists z. \ (c = udf \wedge q[z] = A \wedge t[z] \geq 0 \wedge q' = \lambda j.(\text{if } j = z \text{ then } B \text{ else } q[j]) \ )$
choose$_1$	$\exists z. \left(\begin{array}{l} c = udf \wedge q[z] = A \wedge t[z] < 1 \wedge \\ c' = df \wedge q' = \lambda j.(\text{if } j = z \text{ then } C^+ \text{ else } q[j]) \wedge \\ t' = \lambda j.(\text{if } j = z \text{ then } 0 \text{ else } t[j]) \end{array} \right)$
choose$_2$	$\exists z_1, z_2. \left(\begin{array}{l} z_1 \neq z_2 \wedge c = df \wedge q[z_1] = A \wedge t[z_1] < 1 \wedge q[z_2] = \ell^+ \wedge \\ c' = df \wedge t' = \lambda j.(\text{if } j = z_1 \text{ then } 0 \text{ else } t[j]) \wedge \\ q' = \lambda j.(\text{if } j = z_1 \text{ then } C^+ \text{ else } (\text{if } j = z_2 \text{ then } \ell \text{ else } q[j])) \end{array} \right)$
choose$_3$	$\exists z. \left(\begin{array}{l} c = df \wedge q[z] = B^+ \wedge t[z] < 1 \wedge \\ q' = \lambda j.(\text{if } j = z \text{ then } C^+ \text{ else } q[j]) \wedge t' = \lambda j.(\text{if } j = z \text{ then } 0 \text{ else } t[j]) \end{array} \right)$
enter	$\exists z. \ (c = df \wedge q[z] = C^+ \wedge t[z] > 1 \wedge q' = \lambda j.(\text{if } j = z \text{ then } CS^+ \text{ else } q[j]) \ )$
fail	$\exists z. \ (q[z] = C \wedge 0 \leq t[z] \wedge q' = \lambda j.(\text{if } j = z \text{ then } A \text{ else } q[j]) \ )$
exit$_1$	$\exists z. \left(\begin{array}{l} c = df \wedge q[z] = CS^+ \wedge 0 \leq t[z] \wedge \\ c' = udf \wedge q' = \lambda j.(\text{if } j = z \text{ then } A \text{ else } q[j]) \end{array} \right)$
exit$_2$	$\exists z_1, z_2. \left(\begin{array}{l} z_1 \neq z_2 \wedge c = df \wedge q[z_1] = CS \wedge 0 \leq t[z_1] \wedge q[z_2] = \ell^+ \wedge \\ c' = udf \wedge q' = \lambda j.(\text{if } j = z_1 \text{ then } A \text{ else } (\text{if } j = z_2 \text{ then } \ell \text{ else } q[j])) \end{array} \right)$
exit$_3$	$\exists z. \ (c = udf \wedge q[z] = CS \wedge 0 \leq t[z] \wedge q' = \lambda j.(\text{if } j = z \text{ then } A \text{ else } q[j]) \ )$

Legend: $\ell \in \{A, B, C, CS\}$ and if a state variable α is not mentioned in τ, then it is assumed to be unchanged and the conjunct $\alpha' = \lambda j.\alpha[j]$ (if α is either q or t) or the conjunct $\alpha' = \alpha$ (if α is c) must be added to τ.

Figure 2: The step transition formulae of Fischer's protocol

The step transition formulae are listed in Figure 2 and the delay transition has the following form:

$$\exists \varepsilon > 0.(t' = \lambda j.t[j] + \varepsilon). \tag{5}$$

Symbolic computation of pre-images. To understand how it is possible to obtain an $\exists^I$-formula as the result of a pre-image computation, let us consider $pre(\text{enter}, (3))$,[3] i.e.

$$\exists z. \left(\begin{array}{l} c = df \wedge q[z] = C^+ \wedge t[z] > 1 \wedge \\ q' = \lambda j.(\text{if } j = z \text{ then } CS^+ \text{ else } q[j]) \end{array} \right) \wedge \exists z_1, z_2. \left(\begin{array}{l} z_1 \neq z_2 \wedge q'[z_1] = CS^+ \wedge \\ q'[z_2] = CS^+ \end{array} \right),$$

where q' is also existentially quantified. In the following, it will be our goal to eliminate this variable. Without loss of generality, it is possible to instantiate z with z_1 so that after some simple manipulations (i.e., substitute the λ-expression for the update of q, eliminate the existentially quantified variable q' by substitution, and finally perform β-conversion) we derive:

$$\exists z_1, z_2. \left(\ z_1 \neq z_2 \wedge c = df \wedge q[z_1] = C^+ \wedge t[z_1] > 1 \wedge q[z_2] = CS^+ \ \right).$$

Remark In MCMT, state formulae are maintained as existentially quantified conjunctions of literals closed under the following rule: if z_i and z_j are distinct index variables occurring in the existentially quantified prefix, then the formula contains the literal $z_i \neq z_j$. To maintain this invariant, when computing $pre(\text{enter}, (3))$, we must produce three disjuncts corresponding to the cases when the existentially quantified variable z in enter is equal (or distinct) to one (all) of the existentially quantified variables z_1, z_2 in (3): one disjunct is obtained by instantiating z to z_1, one by instantiating z to z_2, and the the last by requiring that z is different from both z_1 and z_2. After the fixed-point check, just one of the three disjunct is kept while the the other two are subsumed. In this way, MCMT *lazily* introduce existentially

[3]By $pre(\text{enter}, (3))$, we mean $pre(\text{enter}, \phi)$, where ϕ is the formula labelled with (3).

quantified variables of sort `INDEX` so as to keep as compact as possible the representation of the set of backward reachable sets. This feature, together with the heuristics for handling quantifier instantiation while performing the safety and fix-point checks, are the key to the competitive performances of MCMT (see Section 5.1 for more details). □

By computing the pre-image of the last formula again w.r.t. enter by instantiating—w.l.o.g.—z with z_2, we are able to derive:

$$\exists z_1, z_2. \left(\; z_1 \neq z_2 \wedge c = df \wedge q[z_1] = C^+ \wedge q[z_2] = C^+ \wedge t[z_1] > 1 \wedge t[z_2] > 1 \; \right).$$

Now, we show how to compute the pre-image w.r.t. the delay transition (5) involving the existentially quantified variable ε; such a variable can always be eliminated as the constraints involving the clocks belong to Linear Arithmetic over the reals that is well-known to admit elimination of quantifiers. To illustrate, consider the computation of the pre-image of the last formula above w.r.t. (5), i.e.

$$\begin{array}{r} \exists \varepsilon > 0.(t' = \lambda j.t[j] + \varepsilon) \wedge \\ \exists z_1, z_2. \left(\; z_1 \neq z_2 \wedge c' = df \wedge q'[z_1] = C^+ \wedge q'[z_2] = C^+ \wedge t'[z_1] > 1 \wedge t'[z_2] > 1 \; \right). \end{array}$$

By simple substitutions, we are able to derive

$$\exists z_1, z_2. \left(\begin{array}{l} z_1 \neq z_2 \wedge c = df \wedge q[z_1] = C^+ \wedge q[z_2] = C^+ \wedge \\ \exists \varepsilon > 0.(t[z_1] + \varepsilon > 1 \wedge t[z_2] + \varepsilon > 1) \end{array} \right),$$

that, by using standard quantifier elimination on the real variable ε, is easily seen to be equivalent to the following $\exists^I$-formula:

$$\exists z_1, z_2. \left(\; z_1 \neq z_2 \wedge c = df \wedge q[z_1] = C^+ \wedge q[z_2] = C^+ \; \right).$$

Safety and fix-point checks. In our framework, the safety check—i.e. the test that the intersection between the set of initial states and the current set of backward reachable states is empty—reduces to the check the unsatisfiability of the conjunction of the formula representing the set of initial states and the formula of the current pre-image. To illustrate, let us consider the first pre-image computed above (i.e. $pre(\mathsf{enter}, (3))$) and formula (1), i.e.

$$\begin{array}{r} \exists z_1, z_2. \left(\; z_1 \neq z_2 \wedge c = df \wedge q[z_1] = C^+ \wedge t[z_1] > 1 \wedge q[z_2] = CS^+ \; \right) \wedge \\ c = udf \wedge \forall z.(q[z] = A \wedge t[z] = 0). \end{array}$$

It is immediate to see that the formula is unsatisfiable by considering the two equalities involving the state variable c of the controller. In fact, by simple equational reasoning, one immediately derives the unsatisfiable equality $df = udf$.

The fix-point checks reduce to testing the unsatisfiability of the negation of the implication between the formula representing the new set of backward reachable states (i.e., at the $i+1$-iteration of the backward reachability procedure) and the old set of backward reachable states (i.e., at the i-iteration of the backward reachability procedure). To give a concrete example, consider again the first pre-image computed above; negating its implication with the disjunction of the formulae (2), (3) and (4), we obtain the following formula:

$$\neg \left(\exists z_1, z_2. \left(\; z_1 \neq z_2 \wedge c = df \wedge t[z_1] > 1 \wedge q[z_1] = C^+ \wedge q[z_2] = CS^+ \; \right) \rightarrow \left(\begin{array}{l} \exists z_1, z_2.(z_1 \neq z_2 \wedge q[z_1] = CS \wedge q[z_2] = CS) \vee \\ \exists z_1, z_2.(z_1 \neq z_2 \wedge q[z_1] = CS^+ \wedge q[z_2] = CS^+) \vee \\ \exists z_1, z_2.(z_1 \neq z_2 \wedge q[z_1] = CS \wedge q[z_2] = CS^+) \end{array} \right) \right),$$

which, by standard manipulations, is equivalent to

$$\begin{aligned}
(\ z_1 \neq z_2 \wedge c = df \wedge t[z_1] > 1 \wedge q[z_1] = C^+ \wedge q[z_2] = CS^+\ ) \wedge \\
\forall w_1, w_2.\neg(w_1 \neq w_2 \wedge q[w_1] = CS \wedge q[w_2] = CS) \wedge \\
\forall w_1, w_2.\neg(w_1 \neq w_2 \wedge q[w_1] = CS^+ \wedge q[w_2] = CS^+) \wedge \\
\forall w_1, w_2.\neg(w_1 \neq w_2 \wedge q[w_1] = CS \wedge q[w_2] = CS^+),
\end{aligned}$$

where z_1 and z_2 are considered as Skolem constants. It is tedious but straightforward to check that by instantiating the universally quantified variables w_1 and w_2 with z_1 and z_2 in all possible ways, the resulting quantifier-free formula is satisfiable;[4] thereby implying that more pre-images must be considered.

Running MCMT. It is possible to feed MCMT with a slightly massaged version of the specification of safety problem for the Fischer's protocol considered above. The specification in the concrete syntax of the tool is omitted for lack of space (it is available from the website mentioned below). The protocol is certified to be safe in around 10 seconds (regardless of the number of processes in the network) on an Intel Centrino 1.729 GHz with 1 Gbyte of RAM. The number of pre-images computed by MCMT during the backward search is 105 and the number of calls to the SMT solver (to discharge instances of the formulae encoding safety and fix-point checks) is 5922. An excerpt of the output of MCMT is the following:

```
MCMT - version 1.0.1
--------------------------
 node 3= [t5_1][1]
 node 4= [t5_2][2]
 node 5= [t2_3][t5_1][1]
 node 6= [t5_2][t5_1][1]
 node 7= [t10_1][t5_1][1]

 [...OMITTED TEXT...]

 node 88 is deleted!
 node 104= [t10_1][t3_2_4][t10_1][t5_2][t7_2][t6_1][t1_2][t1_1][t2_2]
           [t3_1_3][t10_1][t5_2][t5_1][1]
 node 105= [t10_1][t3_2_4][t10_1][t5_2][t7_2][t2_1][t10_1][t5_1][t7_1]
           [t1_2][t1_1][t2_2][t3_1_3][t10_1][t5_2][t5_1][1]
=========================================================================
Global fixpoint reached!

System is SAFE!

Max depth:17   #nodes:105   #deleted nodes:9   #SMT-solver calls:5922
#invariants found:0
=========================================================================
```

The numbering of nodes starts from 3 because the first three nodes (numbered 0, 1, and 2) corresponds to the unsafe formulae (2), (3) and (4). The transitions in Figure 2 are re-named as `t1`, ..., `t9` from the top to the bottom of the figure, i.e. `t1` is initiate, `t2` is choose_1, and so on. For example, Node 3 corresponds to the formula $pre(\texttt{enter}, (3))$ computed above. Since MCMT implements backward reachability by visiting on-the-fly a forest of trees (whose roots corresponds to the unsafe formulae) [13], the tool reports its

[4]That such instantiations suffice follows from Proposition 3.2.

depth (17), the number of nodes it contains (105), and those that have been deleted by a subsumption check [12]. It also reports the number of calls to Yices (5922) and the number of invariants that it was able to synthesize (when the option is turned off, it always indicates 0).

5 Experiments

We describe here some more safety problems (beside the one considered in Section 4) taken from various sources. For the sake of completeness, we include the descriptions of the systems as extended automata (in the style of Uppaal) in Appendix A.

Lynch-Shavit Mutual Exclusion. In the original Fischer's algorithm (see Fig. 4), two numeral parameters A, B guarantee its correctness provided that $B > A$.[5] On the contrary, Lynch-Shavit's algorithm [14] does not depend on constraints on the parameters A and B in order to guarantee the mutual exclusion property. Yet it does rely on them in order to ensure some sort of deadlock freedom. The key point is to add a Boolean variable *v2* in order to check whether the value of another process wishing to enter the critical region was overwritten.

The formulation considered in Figure 5 and formalized in the benchmark `lynch_mahata` from the table of Figure 3 below is taken from [16]. This is a simplified version of the original algorithm from [14]: in the original formulation, minimum and maximum values for a process to perform an action are introduced and timing constraint (*à la* Fischer) are needed, as explained above, to guarantee deadlock freedom. We submitted to MCMT the original formulation too (the input language of our tool is expressive enough to accept it); MCMT was able to check not only mutual exclusion but also the safety lemmas relevant to infer deadlock freedom (all these problems are formalized in the benchmark `lynch_full` from the table of Figure 3 below).

The CSMA Protocol. The CSMA/CD protocol is designed to enable network communication among clients sharing a single bus. The goal of the protocol is to avoid as much as possible that two clients transmit simultaneously (an event called 'collision'); when a collision occurs, the protocol must be able to undertake proper actions. In order to accomplish these tasks, the clients rely on their ability to "listen to" the bus: when the electrical signals carried by the bus are jammed, it means that a collision has occurred.

The protocol roughly behaves as follows. When a client wants to send data, it listens to the connecting medium: if there is no noise deriving from ongoing transmission, the bus is free and the data can be sent; otherwise the client waits for a random amount of time before trying again. Because of the delay in the propagation of electrical signals, the bus may look free for a client, while another one is nevertheless transmitting. In that case, when the collision is detected, the clients interrupt the transmissions and retry after a random time.

Our formalization considers a simplified version of the protocol taken from [20], which models a 10 Mbps Ethernet having a worst case propagation delay σ of 26μs and a transmission time λ of 808μs. The system comprises two kinds of processes: the clients and the bus. All the processes are modeled by timed automata synchronizing by the following events: *begin*, when a client starts a transmission; *end*, when the current transmission is over; *busy*, when the bus notifies to clients that someone is transmitting; *cd*, when a collision is detected by a client. The clients act according the description in Figure 6. In the initial location *wait*, a client can either send a message or do nothing. In the first case, if the bus is free, the client can send event *begin* to the bus, signaling the transmission begins, and go to location

[5] These timing constraints are explicit in standard formulations like [17] and are reflected in the formulation of Figure 2 by the different guards $t[z] < 1$ and $t[z] > 1$ used in the choose and enter transitions.

transm. But if the bus is busy, because some other client is transmitting, the client goes to the location *retry* after it receives event *busy*. Similarly when a collision is detected, it goes to location *retry* after it gets collision event *cd*. In the second case (the client has no data to transmit), it simply stays in location *wait*. Notice that we have two transitions with label *cd* in order to differentiate the scenario when the client detects a collision and wants to transmit, and the scenario when the client detects a collision but has nothing to do. In the location *transm*, the client can either return to location *wait* when the data has been transmitted (after λ time units have elapsed) sending event *end* to the bus; or it can go to location *retry* due to a collision. In location *retry*, the client waits a random time (smaller than $2 * \sigma$), and then begins a retransmission (going to location *transm*) or still waits if another collision occurs. The bus in Figure 7 starts in location *idle*. If it receives event *begin*, it goes to location *active*. There, if the transmission has just begun (a time smaller than σ has elapsed), a collision can occur. In that case, the bus goes to location *collision*. Otherwise, the bus stays in location active and signals that a transmission is ongoing by event *busy* (this means that all clients are able to detect a transmission because the electrical noise have achieved both the ends of the connecting medium). In location *collision*, the bus broadcasts event *cd* to all clients in order for them to recover.

The TTA Protocol. The Time Triggered Architecture (TTA) enables distributed components to communicate in a fault-tolerant safety-critical way; it is employed in control units of cars and aircrafts. A TTA system is composed of host computers (the nodes) connected over a shared bus. Because many nodes share the bus, a time slot of equal duration is assigned to them, so that each node has to await for its turn in order to transmit - this is called a 'time-division multiple-access strategy' (TDMA). Moreover, since no global clock signal exists in the system, each node relies only on its local clock in order to know when its turn starts. *The goal of the start-up protocol is to synchronize local clocks so that each clock agrees with the others on the owner of the current slot.*

We consider here one of its component, called the start-up protocol. Our model is a simplified version of [8] (in [8] bus collisions are added as well, whereas the paper [19] is a further refinement of the protocol, introducing faulty conditions too) and it is depicted in Figure 8. The basic idea of the protocol is that, when a node receives a packet (carrying the sender's identity in the global variable *origin*), it knows the position of the TDMA schedule for the current time and consequently can appropriately set its clock (the local variable *c*). From this event, a new transmission is going to start every interval with duration equal to the slot time. A node (with identifier *i*) starts in location *init*, where it performs some initialization. After at most *max_init_time* time units, it enters in location *listen*. At this point two scenarios may occur: either some nodes are already synchronized or none is synchronized. In the former case, when the node will eventually receive an *i_frame* (that is, a message carrying data) from a synchronized node, it knows the current slot scheduled and so it resets its clock, writes the identifier of the current sender in the local variable *slot*, and goes to location *active*. In the latter case, after *tau_listen(i)* time units, it broadcasts a "wake up" signal (that is, a *cs_frame*) to the other nodes, which are not synchronized, and goes to location *coldstart*. A node goes in this location as well, whenever it gets a *cs_frame* (but not an *i_frame*) from another node. In location *coldstart*, a node goes in location *active* as soon as it receives either an *i_frame* or a *cs_frame*. If nothing happens after *tau_coldstart(i)* time units, the node repeats the broadcast of another *cs_frame*. When eventually a node is in *location active*, it is synchronized and it can send data whenever its slot turn comes, that is whenever the variable *slot* is equal to its identifier *i*.

5.1 Results

We consider a benchmark set made of safety problems derived from the protocols considered above. For each protocol, we tried some variants found in the literature and also some buggy versions (identified

MCMT	Time
fischer_abdulla	10.571
fischer_sal	0.626
fischer_sal_buggy	0.130
fischer_rust	0.187
fischer_uppaal	0.114
lynch_mahata	0.190
lynch_full	95.830
csma	0.908
csma_buggy	1.353
tta	2.098
tta2	14.077

UPPAAL	N=2	N=5	N=10
fischer_sal	0.013	0.082	37.392
lynch_mahata	0.008	0.053	44.345
csma	0.009	0.189	>10min
tta2	0.011	0.062	>10min

Legend: all experiments were conducted on an Intel Centrino 1.729 GHz with 1 Gbyte of RAM, running Ubuntu v9.0 (Linux kernel v.2.6); the version of Yices used was 1.0.27. Timings are in seconds. Both MCMT and Uppaal were run with their default settings.

Figure 3: MCMT and UPPAAL running times

with the suffix '_buggy' in the name of the benchmark). The files containing all the safety problems in the input format of MCMT[6] considered in this paper are available at `http://www.dti.unimi.it/~carioni/mcmt_ta.html`. The table on the left of Figure 3 shows the running times of MCMT v. 1.0.1. For each problem, we check that a safety property is satisfied (or not, for problems with suffix '_buggy') for an arbitrary number of processes. These results clearly show that the approach for the automated verification of parametrized timed networks described in this paper is viable in practice and promises to scale up to large systems. Since, to the best of our knowledge, there is no available model checker capable of solving fully parametric safety problems for timed networks, we compare our results with those of Uppaal v. 4 (`http://www.uppaal.com`) for three (increasing) values of the number N of processes for a sub-set of the problems considered for MCMT, see right of Figure 3. The better performances of MCMT are clear for $N = 10$, not to mention the fact that MCMT certifies safety regardless of the number of processes. The key of the success of MCMT on these benchmarks is its compact representation of the number of process in the configurations, obtained by the lazy introduction of existentially quantified variables of sort `INDEX` while computing pre-images (c.f. Remark 4). Another reason of success lies in the (optimized) full quantifier instantiation, which is able to detect subsumptions due to internal symmetries in the formulae representing nodes.

6 Conclusions and future work

We have shown the viability of the SMT-based model checking approach introduced in [10] for the verification of safety properties of timed networks by using our tool MCMT. There are two advantages in our techniques. First, our tool performs parametrised verification (i.e. it is to verify properties regardless of the number of processes in the network) while other tools (like Uppaal or Kronos) can only deal with bounded instances or (like Sal) require substantial user interaction. Second, our approach seems to be superior also compared to dedicated algorithms like those in [5, 16] as it fully declarative and support variants to the models in a flexible, expressive, and modular way. Experimental evidence on a set of well-known benchmarks show promising performances of MCMT.

[6]The web page of MCMT is `http://homes.dsi.unimi.it/~ghilardi/mcmt`.

Concerning future work, we plan to perform a more exhaustive experimental analysis with MCMT and to enlarge the scope of applicability of the tool (e.g., to consider the verification of hybrid systems). We will also study how to exploit the constraint solving capabilities offered by SMT solvers so as to synthesize constraints on (numeric) parameters that allows one to guarantee the safety of a system. Constraining such parameters is crucial for many of the timed systems considered in this paper such some of variants of the Fischer's protocol (e.g., the one in Figure 4) and the CSMA protocol.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.

[2] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers. In *TACAS*, volume 4424 of *LNCS*, pages 721–736, 2007.

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, LNCS, pages 145–157, 2007.

[4] Parosh Aziz Abdulla, Johann Deneux, and Pritha Mahata. Multi-clock timed networks. In *Proc. LICS'04, IEEE Computer Society*, 2004.

[5] Parosh Aziz Abdulla and Bengt Jonsson. Model checking of systems with many identical timed processes. *Theoretical Computer Science*, pages 241–264, 2003.

[6] B. Bérard and L. Fribourg. Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In *CONCUR'99*, LNCS, pages 178–193, 1999.

[7] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Proc. of CSL*, volume 1683 of *LNCS*, pages 50–66, 1999.

[8] B. Dutertre and M. Sorea. Timed systems in sal. Technical Report SRI-SDL-04-03, SRI International, Menlo Park, CA, 2004.

[9] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS*, pages 352–359. IEEE Computer Society, 1999.

[10] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS, 2008.

[11] S. Ghilardi and S. Ranise. A Note on the Stopping Failures Models. 2009. Draft, available from MCMT web site.

[12] S. Ghilardi and S. Ranise. Model Checking Modulo Theory at work: the intergration of Yices in MCMT. In *AFM 09 (co-located with CAV09)*, 2009.

[13] S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *Proc. of IJCAR 2010*, LNCS, 2010.

[14] N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proc. of IEEE Real-Time Systems Symposium*, pages 2–11, 1992.

[15] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[16] Pritha Mahata. *Model Checking Parametrized Timed Systems*. PhD thesis, Department of Information Technology, Uppsala University, Uppsala, 2005.

[17] H. Rust. *Operational Semantics for Timed Systems: A Non-standard Approach to Uniform Modeling of Timed and Hybrid Systems*, volume 3456 of *LNCS*, chapter 13. A Case Study: Fischer's Protocol. Springer-Verlag, 2005.

[18] R. Shostack. Deciding linear inequalities by computing loop residues. *J. ACM*, pages 769–779, 1981.

[19] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks*, pages 2–11, 2004.

[20] S. Yovine. Kronos:a verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1997.

A Extended automata for the Benchmarks

We collect here the extended automata (in Uppaal style) of the systems considered in our benchmarks. In the graphical format of Uppaal, locations are depicted as circles; each location has a name and optionally an invariant condition placed just near the circle. The initial location is marked with a double circle. Edges between locations are the transitions of the system. Each transition has a guard (green conditions), a list of updates (violet statements) and a synchronization information (cyan expressions), which comprises a channel name followed by either the symbol '?' or the symbol '!'. Guards, updates and synchronizations are optional.

Synchronization is needed to build product automata: the arguments of the product can be copies of the same automaton (like in the TTA case below) or two different automata (like in the CSMA case below).

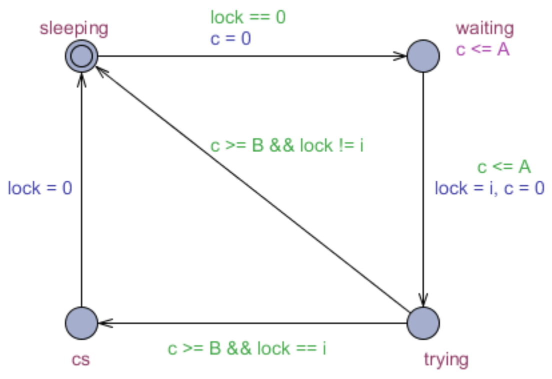


Figure 4: Fischer's algorithm, taken from [8]

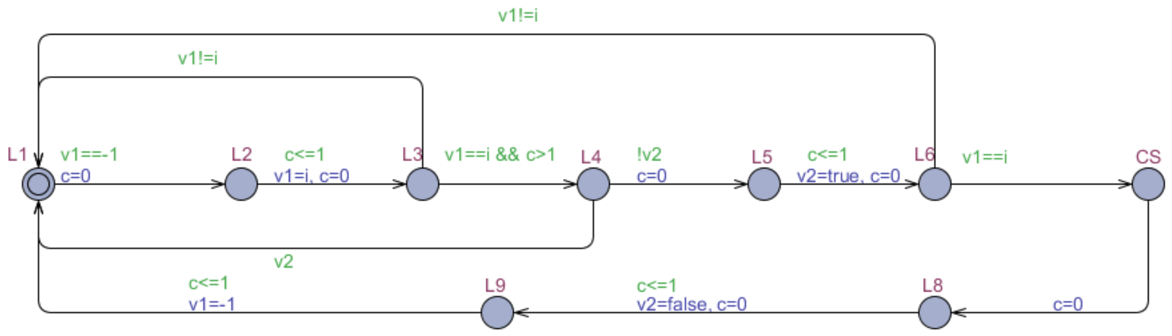


Figure 5: Lynch-Shavit's algorithm, taken from [16]

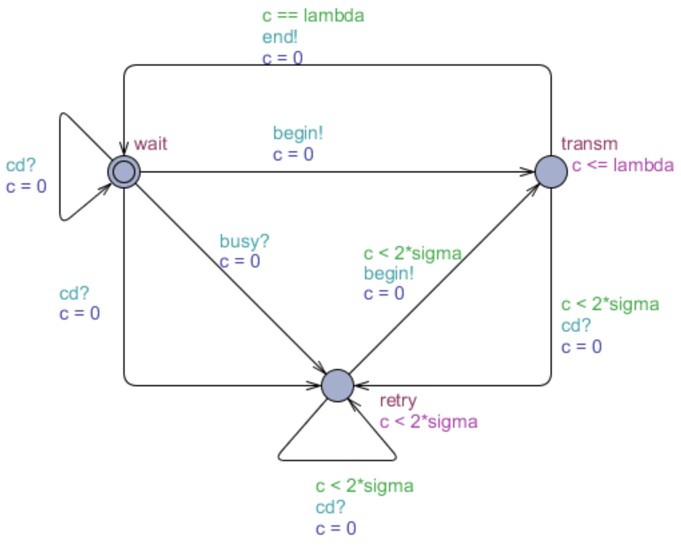


Figure 6: CSMA Client's automaton

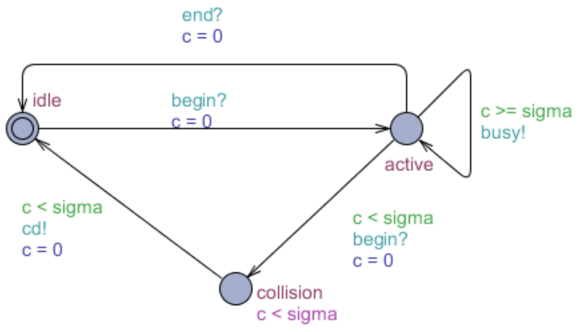


Figure 7: CSMA Bus' automaton

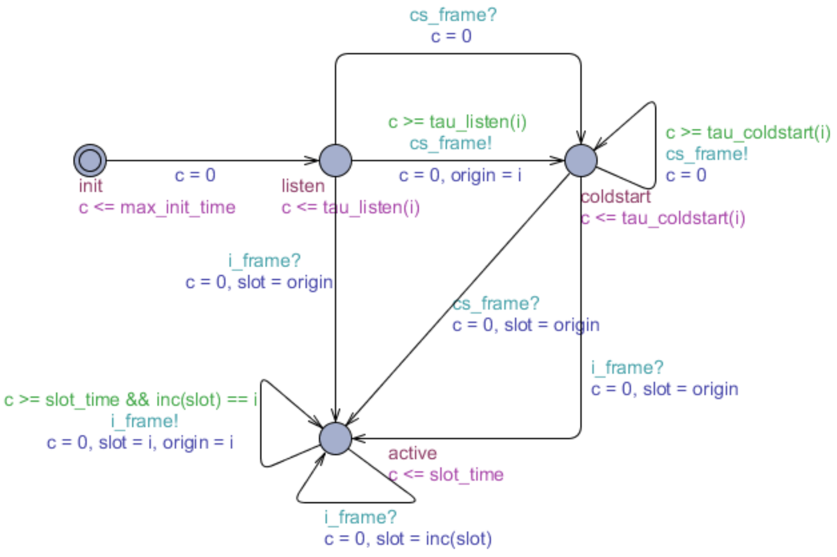


Figure 8: TTA Node's automaton