A Tool for Automated Verification of Parameterized Timed Algorithms

Tristan Crolard **Evguenia Prokofieva**

December 2004

TR–LACL–2004–15

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Département d'Informatique
Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle, 94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01

Laboratory of Algorithmics, Complexity and Logic (LACL)
University Paris 12 (Paris Est)

Technical Report **TR–LACL–2004–15**

T. Crolard, E. Prokofieva.

A Tool for Automated Verification of Parameterized Timed Algorithms

A Tool for Automated Verification of Parameterized Timed Algorithms

Tristan Crolard and Evguenia Prokofieva

Laboratory of Algorithmics, Complexity and Logic
University Paris 12 – Val de Marne* (France)
{crolard,prokofieva}@univ-paris12.fr

Abstract. We present a tool developed for automated verification of parameterized real-time systems. Algorithm specification is provided as Timed Gurevich Abstract State Machines while requirements are expressed as formulas of the First Order Timed Logic (FOTL). In our framework, the semantics of an ASM is also defined as a FOTL-formula. Thus any verification problem amounts to proving the validity of some FOTL-formula. Our method is based on a decidable subclass of FOTL which models some "finiteness" properties of control systems.

1 Preliminaries

We present a tool developed for automated verification of parameterized real-time systems. For algorithm specification we chose Timed Gurevich Abstract State Machines, a particular case of ASM which is apt to specify synchronous real-time algorithms. For requirements specification we use a First Order Timed Logic (FOTL) which permits to express directly the initial specifications. In our framework, the semantics of an ASM is also defined as a FOTL-formula. So any verification problem amounts to proving the validity of some FOTL-formula. Although FOTL is undecidable, there exists decidable subclasses. Our method is based on such a decidable class which models some "finiteness" properties of control systems. The existence of a counter-model for a formula in this class is equivalent to the validity of a formula in the theory of real closed fields. This theory known to be decidable and moreover a quantifier elimination method (QE) can applied to get the result.

The tool consists in two parts. The first part is a translator whose purpose is to generate a FOTL formula describing the set of runs of a given ASM. The core of the tool implements elimination of functional and predicate symbols from a FOTL-formula together with quantifier elimination (we rely here on a well-tested implementation of QE in Redlog [DS97,AD99] which is part of a computer algebra system Reduce [Hea99]). This process is summarized in the following diagram:

* 61, Av. du Général de Gaulle, 94010 Créteil Cedex France

Syntax and semantics of FOTL. A First Order Timed Logic in this framework is is the theory of real addition and unary multiplications by rational numbers (which is known to be decidable [Wei99]) extended by functions with at most one time argument and with other arguments being of finite abstract sorts.

The vocabulary W of a FOTL consists of a finite set of *sorts*, a finite set of *function symbols* and *predicate symbols*. Some sorts are predefined, i.e. have fixed interpretations. Here the predefined sorts are the real numbers $\mathbb{R}$ and time $\mathcal{T} \equiv \mathbb{R}_{\geq 0}$ treated as a subsort of $\mathbb{R}$. The other sorts are finite. The notions of interpretation, model, satisfiability and validity are treated as in first order predicate logic modulo the preinterpreted part of the vocabulary.

Timed Gurevich Abstract States Machines. We consider a particular case of ASM, namely basic ASM that consists of a set of **If-Then**-operators (*rules*). The notations of ASM are self-explanatory so we just give the form of ASM we consider (our notation only slightly differs from that of [Gur95]).

Repeat
  ForAll $\omega \in \Omega$
    InParallelDo
      If $G_1(\omega)$ **Then** $A_1(\omega)$ **EndIf**
      If $G_2(\omega)$ **Then** $A_2(\omega)$ **EndIf**
      . . . . . .
      If $G_m(\omega)$ **Then** $A_m(\omega)$ **EndIf**
    EndDo
  EndForall
EndRepeat

Here each G_i is a *guard*, i.e. a formula without any free variable different from ω, and each A_i is a list of assignments (called *updates*) whose terms have no free variables different from ω. Each assignment has the form $f(T) := \theta$, where f is an internal function, θ is a term and T is a list of terms with the required type. For a precise definition of the semantics see [BS02].

2 Representing ASM runs as FOTL formulas

It turns out that one can characterize the set of total runs of an ASM by an FOTL formula (cf. [BS02]). We give here (in plain english) a list of properties, the conjunction of which characterizes the set of total runs of an ASM.

$\Psi_0(t)$**:** If no guard is valid at t then no guard is valid in a neighborhood of t.

$\Psi_1(t)$**:** If a guard is valid at t for some ω then no guard is valid in some neighborhood of t except t itself.
$\Psi_2(t)$**:** The value of an internal function at t is equal to its value at the left of t.
$\Psi_3(t)$**:** Values of internal functions do not change as long as related guards are false.
$\Psi_4(t)$**:** If some guard related to the internal function f is true at t, f is updated according to the update rule of this guard, and the update holds as long as the guards related to f at time t remain false.
$\Psi_5(t)$**:** If some guard related to the internal function f is true at t, values of f not updated remain the same as long as the guards related to f remain false.

Finally, the total runs are described by the formula $\Psi \equiv \widehat{Init}(0) \bigwedge_{i=0,\ldots,5} \forall\, t\, \Psi_i(t)$ (where *Init* describes the initial state of the ASM) as stated by the following theorem [BS02]:

Theorem 1. *Every model of Ψ is a total run of $\mathcal{A}$, and conversely, every total run of $\mathcal{A}$ is a model of Ψ.*

Automated Generation of Description of ASM Runs The translator[1] is composed of a parser, a type checker, a library of functions which actually perform the translation and a printer which can generate a file conforming to any specific syntax containing definitions and axioms (or lemmas) given in our abstract syntax (see below). Because of this modular design, it is quite simple to extend the syntax of ASM, to generate new lemmas or to use another system than Reduce or PVS.

- *The input file.* An example of input file for the Generalized Railroad Crossing Problem is given in Appendix A (the formal grammar can be found in [BCS00]). The input file containing the description of the ASM is three-fold. The first part contains the signature and the (optional) lists of static and internal functions. The second part contains the definitions of some functions (or predicates) together with the signature of logical variables. Eventually, the third part contains the rules of the ASM.
- *Abstract syntax.* Since this abstract syntax corresponds directly to the concrete syntax defined by the formal grammar, it does not require much comment. Note however that we do not distinguish between terms and formulas at the syntactic level (formulas are just boolean terms) and that the ASM may or not be parameterized.
- *Type checking.* We check that no function is declared both static and internal, that no external function is assigned in the ASM and finally we type check the definitions and the conditional rules. Note that since some provers (such as PVS) use stronger type systems (possibly generating TCC), a type error during the translation does not stop the processing (but should be considered as a warning). Predefined atomic types and the types of the predefined functions and predicates enumerated in section 2 are known by the

[1] The translator (which is currently written in Standard ML) was first developed for a previous experiment with PVS [BCS00]. The printer had just to be rewritten in order to generate a file conforming to Reduce syntax.

translator. Moreover, a very simple but convenient form of subtyping and overloading is provided.
– *The translation.* The implementation of the translation is very close to the definition given in section 2: a set of axioms is generated for each axiom scheme $\Psi_i(t)$ (where each defined symbols is replaced by its definition).

3 A decidable class of verification

In this section we briefly recall some definitions and results from [BS02]. We say that an interpretation f^* of $f : \mathcal{T} \times \mathcal{X} \to \mathcal{Z}$ is a *Finite Interpretation of complexity* (k, n) ((k, n)-FI) if we can divide $\mathcal{X}^*$, the interpretation of $\mathcal{X}$, into k classes and for each class $\mathcal{X}_i^*$, f_x^* is a same constant for all $x \in \mathcal{X}_i^*$ on each of the n intervals of $\mathcal{T}$. The *complexity of a FI of a vocabulary* V is the maximum of complexities of interpretations of all abstract functions, where $\max\{(k_0, n_0), (k_1', n_1')\} = (\max\{k_0, k_1'\}, \max\{n_0, n_1'\})$.

Theorem 2. *It is decidable whether a closed formula of FOTL has a model of complexity* (k, n).

The proof of this theorem provides an algorithm transforming a given formula F into a formula $\widetilde{F}$ with real and integers variables and addition such that $\widetilde{F}$ is valid iff F admits a model of complexity (k, n).

The next question is now what formulas satisfy the following *bounded complexity counter-model property* : "F has a counter-model iff F has a counter-model of a complexity (k, n) for some k,n".

Since in our framework, verification problems are formulas of the form $\Phi \to \Psi$, we introduce two notions that are 'almost' dual: finite satisfiability (for Φ) and finite refutability (for Ψ). Intuitively, finite satisfiability of a formula Φ says that every "finite piece" of any model of Φ is extendable to a model of F of a finite complexity. Finite refutability says that if a formula is refutable the contradiction given by a counter-model is consecrated on a finite piece of a fixed size. The class of implication of FOTL where the premise is finitely satisfiable and the conclusion is finitely refutable with a fixed complexity has a *bounded complexity counter-model property.*

Automated verification of the existence of counter-model of a given compexity The formula corresponding to a verification problem is $\Phi_{runs} \wedge \Phi_{env} \to \Psi_{req}$, where Φ_{runs} is a conjunction of axioms generated by the translator, Φ_{env} is a formula specifying the environment and Ψ_{req} is the property to prove. To apply our method, one have to find first the complexity (k, n) (for a case study we refer the reader to [BCP03]). When this is done the existence of a counter-model for this formula is equivalent of existence of a counter-model of complexity (k, n).

The second part of the tool provides an automated verification of the existence of a (counter-)model with a given complexity (k, n). Since the design of our tool is oriented towards dealing with verification problems, the following files should be supplied:

- The FOTL formula Φ_{runs} describing the ASM (which is generally the output of our translator)
- The FOTL formula Φ_{env} (resp. Ψ_{req}) specifying the environment (resp. the requirements)[2]
- The signature of abstract function symbols, the complexity (k, n) and eventually the list of parameters.

The main steps are:

1. We eliminate abstract function symbols by applying the algorithm associated with theorem 2 to the formula $\neg(\Phi_{runs} \wedge \Phi_{env} \rightarrow \Psi_{req})$. The resulting formula contains real and integer variables. These two sorts of variables are well separated: an atomic formula contains either only real variables or only integer variables (and even no addition in the latter case).
2. We have two strategies for elimination of integer variables. If the signature doesn't contain '=' over abstract sorts and any functional symbols with a result of abstract type, and if the partition of the abstract sorts is the same for all abstract functions, we can suppose that the interpretation of any abstract sort has cardinality k and that each class contains exactly one element. In this case integers existential (resp. universal) quantifiers can be replaced by finite disjunctions (resp. conjunctions). Otherwise, since the signature contains only order (and no addition) we have developed an algorithm for integer quantifier elimination which is simpler than in Presburger arithmetic.
3. We eliminate real quantifiers using Reduce. We take advantage here of the knowledge we have about the structure of this formula, actually it is of the form $\exists\pi(\wedge\phi_i)$, where π is a list of reals variables and each ϕ_i contains a few existential and universal real quantifiers. We eliminate consecutively all quantifiers from each ϕ_i. At each step we try to get the smaller result. If our problem has abstract sorts, it seems (from an empirical standpoint) that the optimal way is to take some innermost quantifier and then to apply QE to this subformula and so on. On the other hand, when we have only real variables there is no improvement with this method, and we can equally apply QE to the whole formula ϕ_i (but in the latter case, Reduce function *rlqews* gives a smaller result than *rlqe*). At the last step, we apply QE on the resulting formula which is in existential prenex form. Note that the number of quantifiers in this formula depends on how the problem was first specified as an ASM.

4 Conclusion and Future works

Let us first say a word about the current state of implementation. The core of the tool, which is the module that interacts with Reduce, is written in C++ and runs on any personal computer. This module can also invoke Reduce on a

[2] An example of requirement formula for the Generalized Railroad Crossing Problem is given in Appendix A.

distant platform (which is actually needed for dealing with large formulas). In this case, the interaction with Reduce relies on remote pipes and a shared file system. Our main distant server is a Sun sparc Ultra 10 with 1Gb memory, but when needed, we also used a Sun Fire 280 with 4Gb memory[3].

From a theoretical point of view any verification problem belonging to the decidable class mentioned above can be treated with our tool. In practice, QE fails when the formula contains a few dozen variables (as one can expect from its complexity). We have thus experimented different implementations of QE in Reduce in order to find out which method is more appropriated. We also implemented some heuristics to diminish the number of quantified variables depending on specific properties of the problem in hand. Besides, the number of quantified real variables introduced during the process strongly depends on the number of functions in the initial ASM. To overcome this problem, we intend to develop further some recent ideas concerning how to improve the ASM and thus stay within the applicability range of our approach.

References

[AD99] T. Sturm A. Dolzmann. Redlog user manual - edition 2.0, for redlog version 2.0. Fakultät für Mathematik und Informatik, Universität Passau, 1999. MIP-9905.

[BCP03] D. Beauquier, T. Crolard, and E. Prokofieva. Automatic verification of real time systems: A case study. In *Third Workshop on Automated Verification of Critical Systems (AVoCS'2003)*, Technical Report of the University of Southampton, Southampton (UK), April 2003.

[BCS00] D. Beauquier, T. Crolard, and A. Slissenko. A predicate logic framework for mechanical verification of real-time Gurevich Abstract State Machines: A case study with PVS. Technical Report 00–25, University Paris 12, Department of Informatics, 2000. Available at http://www.univ-paris12.fr/lacl/.

[BS02] D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: Basic properties and a decidable class. *Annals of Pure and Applied Logic*, 113(1–3):13–52, 2002.

[DS97] Andreas Dolzmann and Thomas Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(2):2–9, June 1997.

[Gur95] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–93. Oxford University Press, 1995.

[Hea99] Anthony C. Hearn. Reduce user's and contributed packages manual, version 3.7. Available from Konrad-Zuse-Zentrum Berlin, Germany, February 1999. http://www.uni-koeln.de/REDUCE/.

[Wei99] V. Weispfenning. Mixed real-integer linear quantifier elimination. In *Proc. of the 1999 Int. Symp. on Symbolic and Algebraic Computations (ISSAC'99)*, pages 129–136. ACM Press, 1999.

[3] We would like to thank J.-M. Moreno from the Computer Science Department of Paris 7 University for allowing us to experiment with this server.

A Example: the Generalized Railroad Crossing Problem

A.1 The input file describing the ASM

```
Signature Railroad

  WT : time;
  dclose : time;
  dmin : time;
  DL : Tracks -> time;
  NoDL : Tracks -> bool;
  Cmg : Tracks -> bool;
  DirOp : bool;
  SafeToOpen : bool;
  Init : bool;

  Static WT, dmin, dclose;
  Internal NoDL, DL, DirOp;

End

Variables

  x : Tracks;

Define

  WT : time = (dmin - dclose);
  SafeToOpen : bool = forall x (not Cmg(x) or NoDL(x) or DL(x) > CT);
  Init : bool = forall x (DirOp and NoDL(x) and DL(x)=0);

  Repeat
    Forall x in Tracks
      InParallelDo
        If (Cmg(x) and NoDL(x)) Then DL(x) := (CT + WT); NoDL(x) := false; EndIf
        If (not Cmg(x) and not NoDL(x)) Then NoDL(x) := true; EndIf
        If (DirOp and not SafeToOpen) Then DirOp := false; EndIf
        If (not DirOp and SafeToOpen) Then DirOp := true; EndIf
      EndDo
    EndForall
  EndRepeat

End
```

A.2 Requirement

```
Liveness: all t ((t >=0 impl (all x (not Cmg(t,x) or all t1 (t1>=t impl (all t2
           ((t2>=t1 and t>t2) impl (Cmg(t2,x) impl t<t1+WT)))))impl DirOp(t))))
```