Efficient Real-Time Model Checking using Tabled Logic Programming and Constraints*

Giridhar Pemmasani, C.R. Ramakrishnan, and I.V. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, U.S.A.
E-mail: {giri,cram,ram}@cs.sunysb.edu

Abstract. Logic programming based tools for real-time model checking are beginning to emerge. In a previous work we had demonstrated the feasibility of building such a model checker by combining constraint processing and tabulation. But efficiency and practicality of such a model checker were not adequately addressed. In this paper we describe XMC/dbm, an efficient model checker for real-time systems using tabling. Performance gains in XMC/dbm directly arise from the use of a lightweight constraint solver combined with tabling. Specifically the timing constraints are represented by difference bound matrices which are encoded as Prolog terms. Operations on these matrices, the dominant component in real-time model checking, are shared using tabling. We provide experimental evidence that the performance of XMC/dbm is considerably better than our previous real-time model checker and is highly competitive to other well known real-time model checkers implemented in C/C++. An important aspect of XMC/dbm is that it can handle verification of systems consisting of untimed components with performance comparable to verification systems built specifically for untimed systems.

1 Introduction

In the recent past several techniques for verification based on logic programming have been developed. For example, constraint logic programming has been used for the analysis and verification of hybrid and real-time systems [23, 11]. Partial deduction techniques as well as CLP have been used for model checking infinite-state systems [7, 6, 17, 18]. New logic program transformations have been devised for verifying parameterized systems [21, 10] and more recent work includes verification of security protocols [5].

Tools based on these techniques, such as our XMC system are beginning to emerge. The XMC system is an efficient and flexible model checker for finite-state systems [19] implemented using the XSB tabled logic programming system [24]. The verification environment provided by the XMC system includes system and

* This work was supported in part by NSF grants EIA-9705998, CCR-9876242, and IIS-0072927.

property specification languages, a simulator, model checker and a justifier for explaining the model checking results. The practicality of the XMC model checker has been demonstrated on several large-scale verification problems [20].

XMC is a model checker for untimed systems. But many reactive systems operate under real-time requirements. Model checking such systems requires constraint processing machinery, an observation that was the basis for several CLP based techniques [11, 7, 6, 18]. In [9] we developed XMC/RT for model checking real-time systems using CLP with tabulation. It was implemented by adding a generic constraint solver for linear constraints over the reals to XSB, thereby deriving a constraint logic programming system with tabulation.

XMC/RT demonstrated the feasibility of building model checkers using logic programming technologies for real-time systems that can be competitive with those implemented in other languages such as C/C++. However to make it as usable a system as XMC, there were performance issues that remained to be addressed. Most notably, constraint handling in XMC/RT was not tightly integrated to the XSB system. In particular it used the POLINE polyhedra package [12], a generic constraint solver library routines (written in C++) for linear arithmetic constraints over reals. There are several drawbacks with this approach. First of all, this package had to be interfaced such that the constraints are stored and manipulated by the constraint solver, but handles to these constraints may be present on the Prolog side. This made any low-overhead management of storage for the the combined system difficult. Secondly, constraints over linear arithmetic are more general than those that arise in real-time model checking. The more general constraint solver imposes additional overheads. Lastly, delegating all the constraint processing to an external handler (such as POLINE) makes it difficult, if not impossible, to do low-level optimizations on the constraint operations. A tightly integrated light-weight constraint solver would facilitate such optimizations. It will significantly improve the time as well as memory utilization of a real-time model checker and thereby enhance its scalability to handle large verification problems. Moreover a beneficial fallout of a tight integration is that it provides a unified environment for model checking systems consisting of both timed as well as untimed components.

In this paper we describe the design and implementation of XMC/dbm, a real-time model checker that eliminates some of the problems in XMC/RT. The main idea is to use a representation for constraints that can be efficiently encoded and manipulated as Prolog terms. In XMC/dbm the timing constraints are represented using *Difference Bound Matrices (DBMs)* [8], an idea gainfully explored in Uppaal [16]. Constraints on clocks can be concisely represented in DBMs and efficiently manipulated. Most of the expensive computation in real-time model checking centers around manipulating such constraints. Hence sharing such computations via caching becomes critical for performance. A tabled logic programming system offers a natural platform for implementing such a model checker. Specifically, in XMC/dbm the computations over DBMs are tabled. Combining this with other generic as well as Prolog-specific optimizations we obtain considerable performance gains in time and space. XMC/dbm's performance is considerably better than XMC/RT and is competitive with other well known real-time model checkers such as Uppaal [15] and HyTech [13].

Traditionally timed and untimed systems have been viewed as distinct verification domains. An important aspect of XMC/dbm (inherited from XMC/RT) is that it can handle verification of systems consisting of timed as well as untimed components. It is accomplished without unduly compromising the performance of model checking untimed systems. Finally it is interesting to note that so far the superior termination properties of a tabled logic programming system have been exploited for implementing model checkers. XMC/dbm demonstrates that the naturalness of caching that "comes for free" in such systems can also play an equally important role.

In terms of related work, several researchers, as noted earlier, have also used (constraint) logic programming for the verification of real-time and other infinite-state systems. The distinguishing aspect of the current work is that instead of exploiting the expressiveness of constraint logic programming to expand the set of systems that can be modeled, our primary emphasis is on building an efficient, usable and unified system that scales to real-world verification problems.

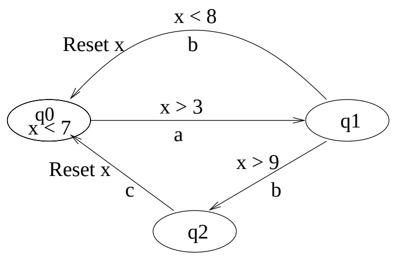
The rest of this paper is organized as follows. In Section 2.1 we present a brief discussion on timed automata, timed modal mu-calculus and model checking of timed automata with timed modal mu-calculus. The encoding of DBMs and related operations as logic programs, as well as optimizations on these operations are described in Section 3. We present experimental evaluation of XMC/dbm in Section 4 and the impact of optimizations used in the implementation in Section 5. Concluding remarks appear in Section 6.

2 Preliminaries

2.1 Timed Automata

Timed safety automata (TSA) [1] model real-time systems by augmenting finite-state automata with real-valued clocks. A TSA has a finite set of control *locations* (corresponding to the states in a finite-state automaton) and a set of *transitions* between these locations, and a finite set of *clocks* associated with the automaton. A *state* of an automaton is characterized by a control location (the "current" location) and a valuation of clocks. A transition may be annotated with an *action label* (drawn from a finite set of symbols) which is used for synchronization between multiple TSAs; a *guard* which is a constraint over clock valuations that must be satisfied for that transition to be enabled; and a set of clocks that are *reset* to zero when that transition is taken. Each location is associated with a location *invariant*, a constraint on clock valuations which must be satisfied when an automaton remains at that location. Constraints on clocks (location invariants and guards) in a TSA are a conjunction of base constraints of the form $x \odot k$ and $x - y \odot k$, where x and y are clocks, $\odot$ is one of $\{<, \leq, >, \geq\}$ and k is an integer constant.

Transitions are instantaneous: no time elapses when a transition is taken; however, a TSA may choose to remain at a location for an arbitrary period of time as long as the location invariant is satisfied. A real-time system is modeled as a synchronous parallel composition of TSAs. All the clocks in a system progress at a constant rate. We assume that the real-time system is non-Zeno: the system does not remain in the same location indefinitely.



(a)

```
invariant(state(q0,X)) :-
   X < 7.
trans(state(q0,X), a, state(q1,X)) :-
   X > 3. % Guard
trans(state(q1,X0), b, state(q0,X1)) :-
   X0 < 8, X1 = 0. % Guard, Reset
trans(state(q1,X), b, state(q2,X)):-
   X > 9. % Guard
trans(state(q2,_), c, state(q0,X)) :-
   X = 0. % Reset
```
(b)

Fig. 1. An example Timed Safety Automaton (a) and its representation as a constraint logic program (b).

TSA can be represented as a constraint logic program by encoding the transition relation. For instance, the TSA given in Figure 1(a) is represented by the constraint logic program in Figure 1(b). Each state of the system is encoded by the term `state(`l`, `$\overline{C}$`)` where l denotes the location and $\overline{C}$ denotes zero or more clocks. Transitions without guards or resets are represented simply as facts. For transitions without resets the clock valuations at the source and destination states remain unchanged.

TSAs can be augmented with discrete variables, which are non-clock, finite-domain variables; note that this does not increase the expressive power of TSAs. When two or more TSAs are composed in parallel, the locations of the composed system will be tuples, each element of the tuple denoting the location of the corresponding component TSA. The clocks of the composed TSA will be the union of the set of clocks in each component TSA. It is useful to consider as a transition the evolution of a system where it stays in the same location while time elapses. We call such special transitions as *delay* transitions.

The constraint-based representation of TSAs permit us to talk uniformly about a single valuation of clocks (where each clock is mapped to a unique real number) as well as a set of valuations that show the same behavior. A set of clock valuations that can be represented by as a conjunction of a finite number of clock constraints is called a *zone*. Note that by definition, a zone is a convex region of the clock valuation space.

2.2 Timed Modal Mu-Calculus

Timed modal mu-calculus [22] adds time modalities $\langle\epsilon\rangle$ and $[\epsilon]$ and resettable clocks to the modal mu-calculus. We use the following syntax for timed modal mu-calculus formulas (from [9]):

```
F -->  tt | ff | atomic(C)
     | and(F, F) | or(F, F) | neg(F)
     | diam(A, F) | box(A, F)
     | epsdiam(F) | epsbox(F) | reset(Z, F)
     | form(X)
```

where `tt` and `ff` denote constants *true* and *false* respectively; `atomic(C)` is a base formula where `C` is an atomic proposition or a constraint over system and formula clocks; `and`, `or` and `not` are standard logical connectives; `diam(A, F)` and `box(A, F)` are dual modal operators from classical modal mu-calculus (`diam(A,F)` asserts that there exists a transition labeled `A` after which `F` holds; `box(A,F)` asserts that `F` holds after every transition labeled `A`); `epsdiam(F)` (after some delay `F` holds) and `epsbox(F)` (after every delay `F` holds) are dual time modalities; `reset(Z,F)` defines a new clock `Z` local to formula `F`; and `form(X)` refers to a mu-calculus variable `X` defined using fixed point equations of the form `X += F` (least fixed point) or `X -= F` (greatest fixed point).

For example, consider the TSA in Figure 1 in location `q1` with clock $X = 7$ (denoted by `state(q1, 7)`). A `b` transition is possible in this state, and hence the formula `diam(action(b), tt)` holds at `state(q1, 7)`. It follows that `epsdiam(diam(action(b), tt))` also holds at `state(q1, 7)` (i.e., with zero delay). Now consider `state(q1, 8)`. The formula `diam(action(b), tt)` no longer holds, but `epsdiam(diam(action(b), tt))` holds at `state(q1, 8)` since a `b` transition is possible after a delay of > 1 second. It also follows that `epsbox(diam(action(b), tt))` does not hold at `state(q1, 8)`.

The model of a timed modal mu-calculus formula is defined with respect to structures called dense labeled transition systems that can be derived from TSAs. The model of a formula is a set of states in the given structure where the formula holds, and is defined inductively based on the syntax of the formula. Following XMC and XMC/RT we encode these inductive rules as a tabled logic program to derive a model checker. In XMC, the states are represented as Herbrand terms. In case of real-time systems, we choose an appropriate constraint representation to denote a set of states. In XMC/RT, we chose to use the POLINE polyhedra package to represent and manipulate constraints. In this paper, we use Difference Bound Matrices (DBMs), which are themselves represented as Prolog terms, to denote sets of states, and construct an efficient solver for DBMs.

2.3 Model Checking the Timed Modal Mu-Calculus

We use the formulation of real-time model checking presented in [9] and recalled in Figure 2. In that model checker, we defined a predicate `models(`R`, `F`, `Rs`)` that, given a *zone* R finds the largest (finite) set of zones Rs that model F

```
models(SS, F, SR) :- union(R, models1(SS, F, R), SR).

models1(SS, neg(F), SR) :-   % negation (due to greatest fixed points)
        models(SS, F, NegSR), diff(SS, NegSR, SR).

models1(SS, box(Act, F), SR) :-  % universal transition modality
        split(SS, Act, LSS),
        member(S, LSS),
        findall(TS, trans(S, Act, TS), TSS),
        all_models(TSS, F, S, Act, SR).

all_models([], _, _, _, []).
all_models([SS0|Rest], F, S, Act, SR) :-
        models(SS0, F, SR0),
        inverse_trans(SR0, Act, S, SR1),
        all_models(Rest, F, S, Act, SR2),
        conjunction(SR1, SR2, SR).

models1(SS, epsbox(F), SR) :-    % universal time modality
        univ_elim(D, (    trans(SS, e(D), TS),
                          models(TS, F, TR),
                          inverse_trans(TR, e(D), SS, SRD)
                     ), SRD, SR).
```

Fig. 2. Encoding of the XMC/RT model checker (from [9]).

and are contained in R. Note that, for a real-time system made up of a parallel composition of TSAs each zone is a tuple of locations (each element of the tuple denoting the location the corresponding TSA is in) and a conjunction of clock constraints. As explained in [9], this formulation differs significantly from the finite-state model checker in XMC [20] where the binary `models` predicate simply checks if a given system state is in the model of a formula. The first argument to `models/3` in the real-time checker represents a *set* of states, not all of which may model the given formula (the second argument). We could assume that when the goal succeeds, the first argument will be narrowed to a set of states that do model the formula. However, for eliminating the universal quantifier over time delays that is introduced by the universal time modality we need the (complete) set of *all* states that model a given formula. We accomplish this by aggregating such a set (the third argument) using a constraint operation `union`.

Apart from `union` we use two basic constraint operations to manipulate zones in the definition of `models/3`: `diff/3` which computes the difference between two constraints and `conjunction` that computes the intersection of two constraints; a derived constraint operation `univ_elim` which is used to eliminate a universally quantified delay variable, implemented by using difference and projection operations on constraints; and two operations on constraints based on transitions of a timed automata: `split` which splits a given zone according to a transition label, and `inverse_trans` which finds the subset of a source zone that takes the automaton into a given target zone.

In addition to these operations used directly by the `models/3` predicate, the real-time model checker also uses a number of constraint operations to construct zones and compute global transitions from the given timed-automata specifications. Instead of directly using a constraint logic programming system, the formulation in [9] presents the model checker in terms of a tabled logic program with explicit references to constraint solving operations. This design decision was made due to two orthogonal reasons. First, tabled resolution is central to the high-level implementation of the model checker and there is (as yet) no single system that integrates constraint solving over reals with tabled resolution. Secondly, encoding delay transitions in a constraint language can introduce significant overheads due to introduction of the delay variable D and its subsequent elimination. In contrast, since all clocks move at the same rate, the effect of delay transitions on clock valuations can be realized by direct constraint manipulation. In this paper, we follow the overall design of [9] and describe a DBM-based representation and manipulation of clock constraints.

3 Difference Bounds Matrices

Timed safety automata restrict clock constraints to those of the form $x_1 - x_2 \sim c$ and $x_1 \sim c$, where x_1 and x_2 are clocks, c is an integer constant and $\sim$ is one of $\{<, \leq\}$. These constraints can be represented in a matrix form, called a *difference bound matrix* (DBM) [8] with the rows and columns labeled by clocks. We assume that clocks are named $x_1, x_2, \ldots$. Each element $M_{i,j}$ in a DBM is a pair $(b, \sim)$ representing the constraint $x_i - x_j \sim b$: i.e., an upper bound on the difference between x_i and x_j. To represent constraints of the form $x_i \sim c$, we use a special 0th "clock" x_0 whose value is fixed at 0; thus $x_i \sim c$, which is equivalent to $x_i - x_0 \sim c$ is represented by entry $M_{i,0}$ and constraint $c \sim x_j$, which is equivalent to $x_0 - x_j \sim -c$ is represented by entry $M_{0,j}$. When there is no upper bound on $x_i - x_j$, then the entry $M_{i,j}$ is set to $(\infty, <)$.

Consider a timed automata with two clocks x_1 and x_2 with constraints $x_1 \geq 6, x_1 < 9$. A DBM representing these constraints is give in Figure 3(a). In DBM 3(a), since $x_1 < 9$ and $0 \leq x_2$, we can infer that $x_1 - x_2 < 9$: i.e., the entry $M_{1,2}$ can be changed from $(\infty, <)$ to $(9, <)$, in effect tightening the upper bound on $x_1 - x_2$. This yields DBM in Figure 3(b). No further nontrivial inferences are possible in DBM 3(b). A DBM where each entry is the tightest possible constraint that can be inferred from that DBM is said to be in *canonical form.*

When a DBM is in canonical form, the effect of each entry has been have been fully propagated through the DBM. Hence entries in a canonical DBM can be manipulated independently. Consider adding constraints $3 \leq x_2 \leq 7$ to the DBM in Figure 3(b). The resulting DBM appears in Figure 4(a), and its canonical form is in Figure 4(b).

$$\underset{\text{(a)}}{\begin{pmatrix} (0,\leq) & (-6,\leq) & (0,\leq) \\ (9,<) & (0,\leq) & (\infty,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}} \qquad \underset{\text{(b)}}{\begin{pmatrix} (0,\leq) & (-6,\leq) & (0,\leq) \\ (9,<) & (0,\leq) & (9,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}}$$

Fig. 3. DBMs for constraints $6 \leq x_1 < 9$; DBM (b) is the canonical form of DBM (a).

$$\underset{\text{(a)}}{\begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (9,<) & (0,\leq) & (9,<) \\ (7,\leq) & (\infty,<) & (0,\leq) \end{pmatrix}} \underset{\text{(b)}}{\begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (9,<) & (0,\leq) & (6,<) \\ (7,\leq) & (1,\leq) & (0,\leq) \end{pmatrix}} \underset{\text{(c)}}{\begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (\infty,<) & (0,\leq) & (6,<) \\ (\infty,<) & (1,\leq) & (0,\leq) \end{pmatrix}}$$

Fig. 4. (a) DBM for constraints $6 \leq x_1 < 9 \wedge 3 \leq x_2 \leq 7$; (b) is the canonical form of DBM (a); DBM (c) represents the clock values after an arbitrary passage of time.

Canonical forms enable efficient implementation of several constraint operations such as delays. For instance, consider the set of clock valuations that can be reached after an arbitrary delay from a given zone. The passage of time eliminates the upper bounds on the clocks, but the relationship between any two clocks is preserved (since all clocks progress uniformly). If a DBM is in canonical form, this delay operation can be performed by simply eliminating the upper bounds on individual clocks (i.e., the 0-th column). An arbitrary passage of time from DBMs in Figures 4(a) and (b) results in the DBM is in Figure 4(c).

3.1 Constraint Operations with DBMs

We define the following constraint operations over DBMs:

- `add_constraints(Pi1, Constraints, Pi2)`: Given a DBM `Pi1`, generates a new DBM `Pi2` by adding further constraints on the clocks, specified by the list `Constraints`. This operation is implemented by iterating through each element of the DBM, and replacing a DBM entry if there is a tighter bound on that value given in `Constraints`. The operation neither expects, nor generates, DBMs in canonical form.
- `delay(Pi1, Pi2)`: Given a DBM `Pi1`, computes the DBM `Pi2` reachable from `Pi1` by incrementing all clocks by an arbitrary amount (arbitrary delay). This operation is implemented by setting the upper bounds on all clocks, specified in the 0-th column, to infinity. The argument `Pi1` must be in canonical form; the output, `Pi2`, will also in canonical form.
- `canonical_form(Pi1, Pi2)`: Canonical form of a DBM over n clocks can be found in $O(n^3)$ time by adapting an all-pairs shortest path algorithm. We use a modified version of the Floyd-Warshall algorithm [4], reordering an inner loop to work efficiently over the list-of-lists representation of DBMs.

- `apply_resets(Pi1, Clocks, Pi2)`: Given a DBM `Pi1`, generates a new DBM `Pi2` obtained by resetting the values of clocks specified in the list `Clocks` to zero. This operation assumes that the given argument `Pi1` is in canonical form. Reset can be done by setting the 0-th column and 0-th row entries of the given clocks to 0, but the result may not be in canonical form. It is easy to "adjust" the result to its canonical form as follows. For each clock x_i reset by this operation, we copy the 0-th column of the DBM to the i-th column, and the 0-th row to the i-th row. The resulting DBM will be in canonical form [25].
- `check_invariants(Constraints, Pi)`: Check if `Pi` $\wedge$ `Constraints` is satisfiable. This operation exploits the fact that invariant constraints specify only upper bounds of individual clocks and have no constraints that relate different clock values. If `Pi` is in canonical form, these upper bound constraints can be checked by simply verifying that the lower bounds specified by the DBM of `Pi` is consistent with the given upper bounds.
- `conjunction(Pi1, Pi2, PL)`: Given two DBMs `Pi1` and `Pi2`, computes their conjunction `Pi3`. Conjunction of two DBMs M and M' is a DBM M'' such that $M''_{i,j}$ contains the tighter of the two bounds specified by $M_{i,j}$ and $M'_{i,j}$. Hence conjunction is analogous to matrix addition, and can be done in $O(n^2)$ time where n is the number of clocks represented in the DBM.
- `diff(Pi1, Pi2, PL)`: Given a DBM `Pi1` and a DBM `Pi2`, computes the set of DBMs `PL` containing points in `Pi1` that are not in `Pi2`.
- `union(V, Goal, PL)`: Given a goal `Goal` that contains a variable `V`, `PL` is the canonical representation of all states s such that `Goal`$[s/V]$;
- `univ_elim(D, Goal, ZD, PL)`: Given `ZD`, a disjunction of constraints over *Vars* $\cup$ {`D`} representing the set of all solutions to `Goal`, `PL` is a disjunction of constraints over *Vars* such that $\forall v \in \mathbf{R}$ (`ZD` $\wedge$ `D` $= v$) $\Leftrightarrow$ `PL`.

3.2 Using DBM Operations

The constraint operations described above are used in the `models/3` predicate, as well as to generate transitions from the specifications of timed safety automata. We assume that, given the transition relation of each TSA, `global_trans/5` gives the transition relation of the parallel composition of the TSAs. The global transition relation specifies the source and destination locations (each is a tuple with components representing local locations), the action label, guards and resets on the transition. Neither the local nor the global transition relations manipulate constraints. The constraints are constructed, manipulated and interpreted by defining the `trans/3` relation used by the model checker as shown in Figure 5.

The goal `trans((L1,Pi1), A, (L2,Pi2))`, given the set of clock valuations at the source location `L1` in `Pi1`, gives a transition label `A` and the set of clock valuations *at entry* to the destination location `L2`. Both `Pi1` and `Pi2` are represented as DBMs. We assume that the DBM `Pi1` is in canonical form whenever `trans/3` is invoked. The precondition for the `delay` operation is that `Pi1` be in canonical form. Hence we obtain the canonical form requirement as a precondition for `trans/3`. The goal `invariants(L1,Inv1)` simply picks up the loca-

```
trans((L1, Pi1), A, (L2, Pi2)):-
        global_trans(L1, A, L2, Guards, Resets),
        delay(Pi1, Pi3),                   % Let all clocks progress
        invariants(L1, Inv1),              % as long as the
        add_constraints(Pi3, Inv1, Pi4),   % state invariants hold.
        add_constraints(Pi4, Guards, Pi5), % Enfirce the guards and
        canonical_form(Pi5, Pi6),          % reduce to Canonical Form
        apply_resets(Pi6, Resets, Pi2),    % before resetting clocks.
        invariants(L2, Inv2),              % Impose invariants on the
        check_invariants(Pi2, Inv2).       % destination state.
```

Fig. 5. Computing the transition relation using DBM operations.

tion invariants at L1. The invariants of the source location are enforced, using add_constraints before checking the guards on the transition.

Note that the DBM Pi4 at this point may not be canonical, and the operation to enforce the guards does not need Pi4 to be in canonical form. The resulting DBM Pi5 obtained may not be in canonical form either. However, the next operation, to reset clocks specified on the transition, needs the DBM to be in canonical form. Therefore we explicitly reduce the DBM Pi5 to canonical form. The operation apply_resets sets the output DBM, Pi2 in canonical form, which simplifies the last operation check_invariants. Moreover, we obtain the following pre- and post-conditions for computing trans: the input DBM must in canonical form, and the output DBM *will* be in canonical form. We can show by induction on derivations that if the DBM in the initial model checking query is in canonical form, the preconditions for every application of trans will be met.

In this sequence of DBM manipulations, we avoid unnecessary canonical form reductions, as observed in [25]. In previous implementations of real-time model checkers using DBM data structure, certain operations (such as apply_resets) were defined so that they preserve canonical forms. Computation of canonical form is a very expensive operation and avoiding them wherever possible will accrue gains in performance. What is novel in our implementation is that we not only deploy the above techniques to avoid unnecessary canonical form reductions, but also tolerate DBMs that are not in canonical form whenever possible. (See *Avoiding canonical form computation* in Section 5.)

3.3 Optimization of Constraint Operations

Among the constraint operations described above, canonical_form is the most expensive. We have already seen how the sequence of constraint operations can be carefully crafted so as to minimize the number of times canonical forms are explicitly constructed. Complementing this reduction, we lower the overheads for canonical form computation by exploiting the memo tables: by simply tabling canonical_form. As the performance figures presented in Section 4 show, the sharing of canonical form computations significantly reduces model checking time. (See *Sharing canonical form computation* in Section 5.)

We observed that the same DBMs occur in multiple calls to `models` but with different locations. Hence each occurrence is separately tabled. After tabling the canonical form reduction, DBMs may be stored in multiple tables, and in call as well as answer tries. Since the terms representing DBMs are often large, and since DBMs contain no logical variables, we maintain a separate dictionary of DBMs, assigning each DBM a unique index, and use this index in all the tables. The time performance worsens by about 10% due to the indirection involved in accessing a DBM. However the space reductions are significant enough that it is worth the trade-off in time. (See *Dictionary representation* in Section 5.)

4 Experimental Results

In this section, we report on the comparative performance of XMC/dbm in two broad categories: (i) with respect to other real-time model checkers (XMC/RT, Uppaal and HyTech) to measure the effectiveness of XMC/dbm in practice; and (ii) with respect to XMC for model checking untimed systems to measure the overheads due to the constraint operations.

4.1 Real-Time Model Checking with XMC/dbm

We use as benchmarks two example systems, namely, Fischer's mutual exclusion protocol and the bridge-crossing system. (Details can be found in [9].) Table 1 presents the time (in seconds) taken by XMC/dbm, XMC/RT and HyTech for verifying several properties of the two benchmark programs. The measurements were done on a Sun Enterprise 4000 running Solaris 5.2.6 with 2GB of memory.

We verified safety, possibility and liveness properties. The first two are specified using timed modal mu-calculus and the much simpler reachability formula (denoted "mu-calc" and "reach" respectively, in the table) whereas liveness can only be specified in mu calculus. An entry "-" in the table means either that the property cannot be specified in that verification system (e.g., liveness properties with the reachability checker) or that the results could not be collected (e.g., possibility property for any of the systems for 6-processor Fischer's protocol). Observe that not only is XMC/dbm faster (by up to a factor of three) over XMC/RT, it also scales better. Moreover, XMC/dbm is competitive to HyTech.

We also measured the performance of Uppaal version 3.2.4 for verifying the safety property of Fischer's protocol on Intel Xeon 1.7GHz system with 1GB memory running Linux Mandrake 8.0. We measured the CPU time given by the operating system since the tool does not provide timing information. For 6 process Fischer, Uppaal takes 121 seconds whereas XMC/dbm takes 73 seconds.

In XMC/RT, the loose coupling of the POLINE constraint solver with XSB precluded it from any reasonable memory management. Hence we do not compare its memory usage with XMC/dbm. For 4,5 and 6 process Fisher XMC/dbm requires 3.8, 28 and 386 MB respectively. HyTech needs 11, 93 and 678 MB respectively. Uppaal shows much better memory performance: e.g. for 6 processes

System	Property	XMC/dbm		XMC/RT		HyTech
		mu-calc	reach	mu-calc	reach	
Fischer 3 proc	Safety	0.61	0.49	0.84	0.46	0.99
	Possibility	3.7	3.1	4.13	4.08	-
	Liveness	0.68	-	1.54	-	-
Fischer 4 proc	Safety	5.2	4.27	11.4	9.0	8.7
	Possibility	62.1	36.8	105	201	-
	Liveness	11.8	-	33.4	-	-
Fischer 5 proc	Safety	46	39.7	-	-	74.62
	Possibility	681.1	412.6	-	-	-
	Liveness	211.3	-	-	-	-
Fischer 6 proc	Safety	468.8	407.8	-	-	562.27
	Possibility	-	-	-	-	-
	Liveness	-	-	-	-	-
Bridge Crossing	Safety	0.07	0.06	0.19	0.17	0.14
	Bounded Liveness	2.1	0.4	7.0	1.19	1.19
	Liveness	0.2	-	0.8	-	-

Table 1. Comparative performance of XMC/dbm on real-time systems.

it takes only 58 MB. Reducing the memory usage by using alternative encodings of DBMs is a topic of current study.

We measured the relative performance of XMC/dbm on the Philips audio control protocol, a real-life problem also widely reported in the literature [3]. The protocol specifies communication between devices over an interface bus without a central controller. We verified two prperties of the protocol in XMC/dbm. The safety property that the receiver never enters error state was verified by XMC/dbm in 18.4 seconds using 23 MB of memory. The other correctness property that the receiver receives entire message was verified in 35.3 seconds using 42 MB of memory. Both measurements were made on an Intel Xeon 1.7GHz system with 1GB memory running Linux Mandrake 8.0.

4.2 Verification of untimed systems

For measuring the overheads incurred by XMC/dbm when model checking untimed systems we use examples drawn from XMC's benchmark suite: `leader` which is a distributed leader election protocol, and `sieve` which implements a concurrent Sieve of Eratosthenes, were originally from the SPIN [14] example suite; and `metalock` is a protocol used to control access to object lock queues in Java, originally verified in XMC [2]. Table 2 shows the performance of XMC/dbm and XMC on these three examples. In the table the column "Time" shows running time in seconds and the column "Space" shows memory usage in MB. The performance measurements were done on an Intel Xeon 1.7MHz system with 1GB memory running Linux Mandrake 8.0.

System	Property	Size	Time		Space	
			XMC	XMC/dbm	XMC	XMC/dbm
Leader	one_leader	5	1.8	2.2	2.0	3.1
		6	10.2	12.4	5.6	8.8
		7	55.2	69.7	21.2	34.1
Sieve	ae_finish	(6,7,41)	0.9	1.1	1.7	2.6
		(6,8,43)	0.9	1.2	1.7	2.6
		(6,9,47)	1.0	1.3	1.8	2.7
Metalock	mutex	(1,4)	2.5	3.1	2.1	3.4
		(3,1)	1.6	2.2	2.5	3.9
		(2,2)	2.4	3.0	2.4	3.7

Table 2. Comparative performance of XMC/dbm on untimed systems.

In terms of time, XMC/dbm incurs an overhead of at most 30% for model checking untimed systems. The memory used by XMC/dbm is about 60% more than that of XMC. This is due to the `models/3` predicate in XMC/dbm where every state gets recorded twice for each formula: once as the first argument (in the query), and once as the third argument (in the answer). In contrast, using `models/2` in XMC, a state is saved only once for each formula, in the call table alone. In XMC, a query to `models` simply succeeds or fails, and hence nothing is stored in the answer tables.

5 Impact of Optimizations: Experimental Evaluation

Here we present the effect of optimizations described in Section 3:

Sharing Expensive Computations: Recall that computing canonical forms dominates the cost of model checking real-time systems. Therefore caching is critical for improving the performance. A unique aspect of XMC/dbm is to use XSB's tables for caching the results of canonical form computation. This has improved the performance of XMC/dbm by a factor of 2.5. For example, when caching of canonical form computation is verifying the reachability formula in for Fischer's protocol with 3–6 processors takes 0.49, 4.27, 39.7 and 407.8s resp., while it takes 0.76, 9.11, 103 and 1174 resp. when the optimization is disabled.

Avoiding Canonical Form Computation: Recall that a unique aspect of our implementation is that we can avoid canonical form computation immediately after applying the invariants at the source state. This optimization is effective whenever there are location invariants in the transition system. Fischer's protocol has only one location with an invariant, and the effect of this optimization is not measurable. In contrast, the Philips audio protocol specifies several location invariants, and the optimization improves verification times by a factor of nearly 2.5: 18.2s and 35.3s for the safety and correctness properties resp. with the optimization turned on, compared to 45.2s and 92.1s, resp., without optimization.

Dictionary of DBMs: Recall from Section 5 that storing DBMs in a dictionary structure can improve memory usage. With this optimization, we observe total memory usage of 1.6, 9.5 and 76 MB for verifying the safety property (by reachability) for Fischer's protocol with 4, 5 and 6 processes respectively. In contrast, without this optimization, the memory usage was 3.8, 28 and 386 MB respectively. Hence the dictionary structure reduces space requirement by more than a factor of three.

6 Conclusions

We have shown how a lightweight constraint solver combined with tabling and carefully optimized, can yield an efficient model checker for real-time systems. We showed, via experimental evaluation, that it is possible to implement such a model checker (XMC/dbm) without unduly degrading the performance of model checking untimed systems. Such a model checker forms a basis for creating an uniform environment to verify timed as well as untimed systems. The use of model checkers such as XMC/dbm will become important for verifying embedded systems which contain both real-time and non-real-time components. The application of XMC/dbm to such problems is the topic of future work.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Scotland, April 2000.
3. J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using Uppaal. In *Proceedings of the 8th International Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 244–256, New Brunswick, New Jersey, USA, 1996. Springer-Verlag.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1999.
5. G. Delzanno and S. Etalle. Transforming a proof system into prolog for verifying security protocols. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001.
6. G. Delzanno, S. Mukhopadhyay, and A. Podelski. Constraint-based model checking for timed systems with accurate widenings. Technical Report, Max-Planck-Institut für Informatik, 1999.
7. G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), LNCS*, volume 1579, pages 223–239, Amsterdam, March 1999.
8. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of CAV'89.* LNCS 407, 1989.

9. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: a recipe for model checking real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 2000.
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state processes using program transofmration. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001.
11. G. Gupta and E. Pontelli. A Constraint Based Approach for Specification and Verification of Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.
12. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.
13. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. "HyTech: A model checker for hybrid systems". *International Journal on Software Tools for Technology Transfer*, 1(2):110–122, October 1997.
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. K. G. Larsen, P. Pettersson, and W. Yi. Model checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in LNCS, pages 62–88, August 1995.
16. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.
17. M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001.
18. U. Nilsson and J. Lubcke. Constraint logic programming for local and symbolic model-checking. In *In Proc. of the Int'l Conf. on Computational Logic (CL2000)*, volume 1861. Springer-Verlag, 2000.
19. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 143–154, Haifa, Israel, July 1997. Springer-Verlag.
20. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV '00)*, pages 576–580. Springer-Verlag, 2000.
21. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV '01)*, volume 2102 of *LNCS*. Springer-Verlag, 2001.
22. O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification*, volume 939 of *LNCS*. American Mathematical Society, 1995.
23. L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP'96)*, volume LNCS 1102. Springer-Verlag, 1996.
24. XSB. The XSB logic programming system. Available at `www.cs.sunysb.edu/~sbprolog`.
25. S. Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.