Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems*

Farn Wang
Institute of Information Science, Academia Sinica
Taipei, Taiwan 115, Republic of China
+886-2-27883799 ext. 1717; FAX +886-2-27824814; farn@iis.sinica.edu.tw
Tools avaiable at: http://www.iis.sinica.edu.tw/~farn/ddd

Abstract

A new data-structure called DDD (Data-Decision Diagram) for the fully symbolic model-checking of real-time software systems is proposed. DDD is a BDD-like data-structure for the encoding of regions [2]. Unlike DBM which records differences between pairs of clock readings, DDD only uses one auxiliary binary variable for each clock. Thus the number of variables used in DDD is always linear to the number of clocks declared in the input system description. Experiment has been carried out to compare DDD with previous technologies.

1 Introduction

Fully symbolic verification of real-time systems is desirable with the promise of efficient data-sharing. We propose *Data Decision Diagram (DDD)* as the new data-structure for such a purpose. DDD is a BDD-like data-structure [5, 8] for the encoding of regions [2]. The ordering among fractional parts of clock readings is explicitly encoded in the variable ordering of DDD. To record sets of clock readings with the same fractional parts, we add one auxiliary binary variable per clock. Thus in DDD, the number of variables used is linear to the number of clocks. Previous technologies like DBM[10] and CDD[7] keep records of differences between pairs of clock readings and thus incur quadratic numbers of variables in their data-structures. Like BDD[8], DDD is also a minimum canonical form with respect to a given variable ordering. It is also efficient for representing unions of zones. Experiments have shown better space efficiency against previous technologies like DBM (Difference-Bounded Matrix)[10].

We assume that system behavior is described as a set of m symmetric processes, identified with integers 1 to m, running different copies of the same program. Each process can use *global* and *local* variables of type *clock*, *discrete*, and *pointer*. Pointer variables either contain value NULL (0 in fact) or the identifiers of processes. Thus in our representation, we allow complicate dynamic networks to be constructed with pointers. We restrict that each process can declare at most one local clock and there is no global clock. The ordering of fractional parts of clock readings is encoded in DDD with the following *normality condition*.

> *"For any two local clocks $x[i], x[j]$, with $1 \leq i < j \leq m$, of processes i and j respectively, either $x[i]$ does not have a greater fractional part than $x[j]$ does or $x[j]$ is greater than the biggest timing constant used in the input system description and specification."*

A state satisfying the normality condition is called a *normalized state*. With DDD technology, we only work with the normalized images of states in runs. After a clock reading advancement or a clock reset operation, we may have to permute the process indentifiers to maintain state normality. Thus our data-structure is perfect for symmetric systems with symmetric specifications. However, we shall see in section 3 that asymmetric systems can also be handled by DDD technology.

*The work is partially supported by NSC, Taiwan, ROC under grant NSC 89-2213-E-001-002.

Our innovation is that we use one bit for each clock to encode the ordering among the fractional parts of clock readings in the region construction [2]. Compared to the classic DBM [10], DDD provides data-sharing capability of fully symbolic manipulation. In a DBM-based model-checker, since matrices and BDD are two different types of data-structure, we are forced to use a pair of BDD and matrix to represent a region. As a result, the set of regions are forced to be represented as an explicit directed graph which grows exponentially with timing constant magnitude and concurrency size.

Newer technology of NDD[1] and CDD[7] use binary inequalities of the form $x[i] - x[j] \leq c$. NDD uses binary encoding for the possible values of c while CDD uses multiple value-ranges to record them. However, the number of variables in their data-structure is likely to be quadratic to the number of clocks used in the systems. The number of variables used in our DDD technology, on the other hand, is always linear to the number of local clocks.

Here is our presentation plan. Section 2 defines the language for system behavior description. Section 3 discusses representation for asymmetric systems. Section 4 formally presents our data-structure scheme. Section 5 shows how to maintain DDD's after clock reading advancements and clock reset operations. Section 6 compares DDD technology with previous ones by performing experiments on several benchmarks.

2 Real-time software systems

We assume a real-time software system to be composed of a set of concurrent processes running different copies of the same program. Given a system of m processes, the processes are indexed with integers 1 through m which are called *process identifiers*. *NULL* is actually integer 0 and is used as the special null process identifier in data-structure construction. The program is presented as a *timed automaton*[2] equiped with global and local variables of type clock, discrete, and pointer. A global variable can be accessed by all processes while a local variable can only be accessed by its declaring process. Clocks can hold reals, can be tested against integers, and can be reset to zero during transitions. Only one local clock per process is allowed and no global clock is allowed. Discrete variables can hold integer constants. Operations on discrete variables are comparisons and assignments with integer constants. A special local discrete variable *mode* is used to record the operation mode of the executing process. Pointer variables can hold NULL or process identifiers. Operations on pointers are comparisons and assignments with NULL or local process identifiers (the one of the executing process).

2.1 Syntax

Given a system of m processes with variable set X, we can define *global state predicates* and *local state predicates*. Global state predicates are used to specify initial conditions and safety properties and are presented with a process identifier attached to each local variable to distinguish which local references are for which processes. The syntax of a global state-predicate η is:

$$\begin{array}{rcl} \eta & ::= & \mathit{false} \mid x \sim c \mid y = \mathit{NULL} \mid y = c \\ & \mid & x[i] \sim c \mid y[i] = \mathit{NULL} \mid y[i] = c \mid \neg\eta_1 \mid \eta_1 \vee \eta_2 \end{array}$$

Variables appended with square brackets are local variables while those not are global variables. x is either a clock variable or a discrete variable. y is a pointer variable. c is a natural constant. $\sim$ is an inequality operator in $\{\leq, <, =, >, \geq\}$. i is a process identifier constant in $[1, m]$. $\neg$ and $\vee$ are Boolean negation and disjunction respectively. Parentheses can be used to disambiguate the syntax. Standard shorthands are $\mathit{true} \equiv \neg\mathit{false}$, $\eta_1 \wedge \eta_2 \equiv \neg((\neg\eta_1) \vee (\neg\eta_2))$, and $\eta_1 \rightarrow \eta_2 \equiv (\neg\eta_1) \vee \eta_2$.

Local state-predicates are used to describe invariance and triggering conditions in the automata. Their syntax is very much like that of global state-predicates except that all occurrences of process identifier constants are replaced by the symbolic process identifier p which is to be interpreted as the process identifier of the executing process.

In figure 1, we have an example automaton for Fischer's timed mutual exclusion algorithm. Here, we have a global pointer *lock* and a local clock $x[p]$ for process p with p as the symbol for the identifier of the executing process. On each transition (arc), we label the triggering condition and assignment sequence. Testing on and assignments to local discrete `mode` are omitted from the transitions for simplicity. σ and δ are two integer parameters to be substituted in implementation. Mutual exclusion to *mode* 3 is maintained when $\sigma < \delta$.

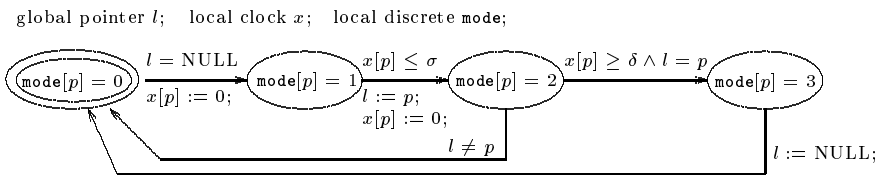


Figure 1: Fischer's timed mutual exclusion algorithm

The formal syntax of a real-time software system S is given as a triple $\langle A, m, I\rangle$ where A is a timed automaton equiped with various variables as mentioned in the above, m is the number of processes in the system, and I is a global state-prediate for the initial condition. A is formally presented as a timed automaton $A = \langle X, \lambda, Q, \mu, E, \tau, \pi\rangle$ with the following restrictions. X is the set of variables. λ maps each variable in X into one of the following five types: local clock, global discrete, local discrete, global pointer, and local pointer. Q is the set of operation modes (or control locations). We assume that the elements in Q are indexed from 0 to $|Q|-1$. μ is a function from $[0, |Q|-1]$ such that for all $q \in [0, |Q|-1]$, $\mu(q)$ is a local state-predicate describing the invariance condition at the q'th operation mode. Also we require that there is a special local discrete variable `mode` which always record the current operation mode index of a process.

$E \subseteq Q \times Q$ is the set of transitions. τ is a function from E such that for all $e \in E$, $\tau(e)$ is a local state-predicate describing the triggering condition of e. π is a function from E such that for all $e \in E$, $\pi(e)$ is a sequence of assignments α of the syntax form:

$$\begin{array}{lcl} \alpha & ::= & L := R; \\ L & ::= & x \mid x[p] \mid y \mid y[p] \\ R & ::= & c \mid NULL \mid p \mid x \mid x[p] \mid y \mid y[p] \end{array}$$

Such an assignment means that the value of R is assigned to variable L. The restriction is that constants or variables cannot be assigned to variables of different types. For example, if L is not of type pointer, then R cannot be $NULL$, p, or any pointer variables.

2.2 Semantics

Definition 1 **states** Suppose we are given a real-time software system $S = \langle A, m, I\rangle$ with $A = \langle X, \lambda, Q, \mu, E, \tau, \pi\rangle$. A state ν is a mapping from $X \times \{0, \ldots, m\}$ such that

- for every global pointer $x \in X$, $\nu(x, 0) \in \{\text{NULL}\} \cup \{1, \ldots, m\}$;
- for every global discrete $x \in X$, $\nu(x, 0) \in \mathcal{N}$;
- for every local clock $x \in X$ and $i \in \{1, \ldots, m\}$, $\nu(x, i) \in \mathcal{R}$;
- for every local pointer $x \in X$ and $i \in \{1, \ldots, m\}$, $\nu(x, i) \in \{\text{NULL}\} \cup \{1, \ldots, m\}$;
- for every local discrete $x \in X$ and $i \in \{1, \ldots, m\}$, $\nu(x, i) \in \mathcal{N}$. ||

For any $t \in \mathcal{R}$, $\nu + t$ is the state identical to ν except that for every local clock $x[i]$, $\nu(x, i) + t = (\nu + t)(x, i)$. For an atomic assignment $L := R;$ and a process identifier i, $\nu[L := R;, i]$ is the new state obtained by letting process i executing $L := R;$ in ν. It is identical to ν except that $(\nu[L := R;, i])(L, i) = \nu(R, i)$. Given a sequence β of assignments and a process identifier i, $\nu[\beta, i]$ is defined in the following inductive way.

- If β is empty, then $\nu[\beta, i] = \nu$.
- If $\beta = \alpha\beta'$, then $\nu[\beta, i] = (\nu[\alpha, i])[\beta', i]$.

Given a global state predicate η and a state ν, we say that ν satisfies η, in symbols $\nu \models \eta$, if and only if the following inductive restrictions hold.

- $\nu \not\models \mathit{false}$.
- $\nu \models x \sim c$ iff $\nu(x, 0) \sim c$.
- $\nu \models y = \text{NULL}$ iff $\nu(y, 0) = \text{NULL}$.
- $\nu \models y = c$ iff $\nu(y, 0) = c$.
- $\nu \models x[i] \sim c$ iff $\nu(x, i) \sim c$.

- $\nu \models y[i] = NULL$ iff $\nu(y,i) = \text{NULL}$.
- $\nu \models y[i] = c$ iff $\nu(y,i) = c$.
- $\nu \models \neg\eta_1$ iff it is not the case that $\nu \models \eta_1$.
- $\nu \models \eta_1 \vee \eta_2$ iff either $\nu \models \eta_1$ or $\nu \models \eta_2$.

The satisfaction relation between a local state predicate η and a state ν by process i, in symbols $\nu, i \models \eta$, can be similarly defined.

- $\nu, i \not\models \mathit{false}$.
- $\nu, i \models x \sim c$ iff $\nu(x,0) \sim c$.
- $\nu, i \models y = \text{NULL}$ iff $\nu(y,0) = \text{NULL}$.
- $\nu, i \models y = c$ iff $\nu(y,0) = c$.
- $\nu, i \models x[p] \sim c$ iff $\nu(x,i) \sim c$.
- $\nu, i \models y[p] = NULL$ iff $\nu(y,i) = \text{NULL}$.
- $\nu, i \models y[p] = c$ iff $\nu(y,i) = c$.
- $\nu, i \models \neg\eta_1$ iff it is not the case that $\nu, i \models \eta_1$.
- $\nu, i \models \eta_1 \vee \eta_2$ iff either $\nu, i \models \eta_1$ or $\nu, i \models \eta_2$.

Definition 2 **runs** Given a real-time software system $S = \langle A, m, I\rangle$ with $A = \langle X, \lambda, Q, \mu, E, \tau, \pi\rangle$, a *$\nu$-run* is an infinite sequence of state-time pair $(\nu_0, t_0)(\nu_1, t_1)\ldots(\nu_k, t_k)\ldots\ldots$ such that $\nu = \nu_0$, $t_0 t_1 \ldots t_k \ldots\ldots$ is a monotonically increasing real-number (time) divergent sequence, and for all $k \geq 0$,

- for all $t \in [0, t_{k+1} - t_k]$ and $1 \leq i \leq m$, $\nu_k + t, i \models \bigvee_{0 \leq q < |Q|}(\texttt{mode}[p] = q \wedge \mu(q))$; and
- either $\nu_k + (t_{k+1} - t_k) = \nu_{k+1}$; or there are $i \in \{1, \ldots, m\}$ and $e \in E$ such that $\nu_k + (t_{k+1} - t_k), i \models \tau(e)$ and $(\nu_k + (t_{k+1} - t_k))[\pi(e), i] = \nu_{k+1}$. ||

A run $\rho = (\nu_0, t_0)(\nu_1, t_1)\ldots(\nu_k, t_k)\ldots\ldots$ of S satisfies *safety* global state predicate η, in symbols $\rho \models \eta$, iff for all $k \geq 0$ and $t_k \leq t \leq t_{k+1}$, $\nu_k + t \models \eta$. We say S satisfies η, in symbols $S \models \eta$, iff for all ν-runs ρ such that $\nu \models I$, $\rho \models \eta$. In case that $S \not\models \eta$, we say S *violates* η.

2.3 Normalized runs and a permutation scheme

Given a constant $r \in \mathcal{R}$, $int(r)$ is the integer part of r while $frac(r)$ is the fractional part of r. Let C_S be the biggest integer constant used to compare with a local clock in the system description S. The normality condition is restated here:

> *"Suppose the local clock is x in a system $S = \langle A, m, I\rangle$. A state ν of S is normalized iff for any $1 \leq i < j \leq m$, either $\nu(x,j) > C_S$ or* $\text{frac}(\nu(x,i)) \leq \text{frac}(\nu(x,j))$*."*

Thus, in a normalized state, we can conceptually divide the process identifiers into several segments in the following pattern.

$$
\begin{array}{l}
\left.\begin{array}{l} 1, \\ \vdots \\ i_1, \end{array}\right\} \\
\left.\begin{array}{l} i_1+1, \\ \vdots \\ i_2, \end{array}\right\} \\
\quad\vdots \\
\left.\begin{array}{l} i_k+1, \\ \vdots \\ i_{k+1}, \end{array}\right\} \\
\left.\begin{array}{l} i_{k+1}+1, \\ \vdots \\ m \end{array}\right\}
\end{array}
\quad
\begin{array}{l}
\nu(x,1) \leq C_S \wedge \ldots \wedge \nu(x,i_1) \leq C_S \\
\wedge \quad \mathit{frac}(\nu(x,1)) = \ldots = \mathit{frac}(\nu(x,i_1)) \neq \mathit{frac}(\nu(x,i_1+1)) \\
\nu(x,i_1+1) \leq C_S \wedge \ldots \wedge \nu(x,i_2) \leq C_S \\
\wedge \quad \mathit{frac}(\nu(x,i_1+1)) = \ldots = \mathit{frac}(\nu(x,i_2)) \neq \mathit{frac}(\nu(x,i_2+1)) \\
\quad\vdots \\
\nu(x,i_k+1) \leq C_S \wedge \ldots \wedge \nu(x,i_{k+1}) \leq C_S \\
\wedge \quad \mathit{frac}(\nu(x,i_k+1)) = \ldots = \mathit{frac}(\nu(x,i_{k+1})) \\
\nu(x,i_{k+1}+1) > C_S \wedge \ldots \wedge \nu(x,m) > C_S
\end{array}
$$

The last segment contains identifiers of those processes whose local clock readings are greater than C_S. The processes with identifiers in a segment other than the last one all have the same fractional part in their clock readings which are no greater than C_S.

Definition 3 **normalized runs** Given a run $\rho = (\nu_0, t_0)(\nu_1, t_1)\ldots(\nu_k, t_k)\ldots\ldots$ of a real-time software system $S = \langle A, m, I\rangle$, a *normalized run* $\bar{\rho}$ of ρ is a mapping from $\mathcal{N} \times \mathcal{R}$ such that for every $k \in \mathcal{N}$ and $0 \leq t \leq t_k$, $\bar{\rho}(k, t)$ is a normalized state of $\nu_k + t$. ||

After each transition or clock readings advancement, a normalized state can be changed to an unnormalized one and there can be more than one identifier permutation whose application can maintain the normality of states. We propose the following permutation rules which can simplify our tool implementation.

1. When process i resets its local clock with a transition, we have to
   - change the identifier of process i to 1 (with global and local variables updated according to the transitions.); and
   - for every $1 \leq j < i$, change the identifier of process j to $j + 1$; and
   - for every $i < k \leq m$, keep the identifier of process k unchanged

   in the destination state to make it normalized. The permutation can be viewed as an identifier movement from i to 1 with displacement $1 - i$.
2. When there is no integer clock readings $\leq C_S$ in the source state and some clocks will advance from noninteger readings $< C_S$ to integer readings, we first have to identify the segment of identifiers of those processes with such clocks. This is the segment right preceding the last segment which contains identifiers of processes with local clock readings $> C_S$. Suppose, we find out $x[j], \ldots, x[k]$ are such clocks. Then in the destination state,
   - for all $j \leq i \leq k$, the identifier of process i is changed to $i - j + 1$; and
   - for all $1 \leq i < j$, the identifier of process i is changed to $i + k - j + 1$; and
   - for all $k < i \leq m$, the identifier of process i is unchanged

   to make it normalized. The permutation can be viewed as an identifier segment movement from $[j, k]$ to $[1, k - j + 1]$ with displacement $1 - j$.
3. When some clocks advance from integer to noninteger readings. we first have to detect if some of those clocks advance their readings from C_S to beyond C_S. Suppose, we find out that $x[j], \ldots, x[k]$ are such clocks. Then in the destination state,
   - for all $j \leq i \leq k$, the identifier of process i is changed to $i + m - k$; and
   - for all $1 \leq i < j$, the identifier of process i is unchanged; and
   - for all $k < i \leq m$, the identifier of process i is changed to $i + j - k - 1$

   to make it normalized. The permutation can be viewed as an identifier segment movement from $[j, k]$ to $[j + m - k, m]$ with displacement $m - k$.
4. In all other cases, the identifiers of all processes stay unchanged to satisfy normality.

However, there is one thing unclear in the above-mention permutation scheme. That is, in the third item, how do we know that those processes with clock readings $= C_S$ will appear with consecutive process identifiers ? That is the good thing about this permutation scheme and can be established by the following lemma.

LEMMA 1 *In the permutation scheme presented in the above, inside all segments of identifiers of processes whose clock readings are $\leq C_S$ and share the same fractional parts, the process identifiers are arranged according to monotonically increasing order of the local clock readings.*

Proof : True because everytime we reset a local clock, we change the identifier of the transiting process to 1. Thus the later a process resets its clock, the earlier its identifier appears in a segment. ||

Definition 4 **Symmetric global state predicate** Given a global state predicate η and a permutation θ of process identifier 1 through m, $\eta\theta$ is a new global state predicate obtained from η by renaming process identifiers according to θ. A global state predicate η for m processes is *symmetric* iff for any permutation θ of 1 through m, η equals to $\eta\theta$ according to commutation laws of Boolean algebra. ||

We want to establish the soundness of our DDD technology with the following lemma.

LEMMA 2 *: Given a state ν of a real-time software system $S = \langle A, m, I\rangle$ and a symmetric global state predicate η, for any normalized image ν' of ν, $\nu \models \eta$ iff $\nu' \models \eta$.*

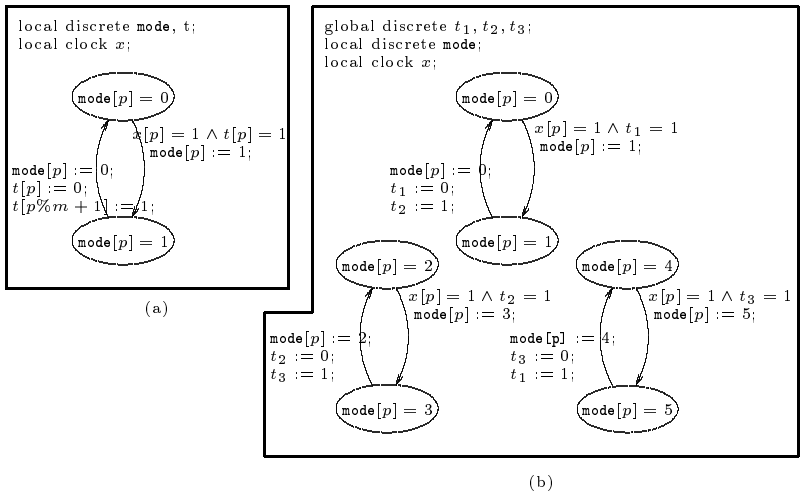


Figure 2: Ring network mutual exclusion algorithm

Proof : Suppose the process identifier permutation that changes ν to ν' is θ. Then $\nu \models \eta$ has the same truth value as $\nu' \models (\eta\theta)$. But $\eta\theta$ is equal to η according to commutation laws of Boolean algebra. Thus the lemma is proven. ||

3 Presentation beyond the framework

Since DDD technology only handles normalized states and normalized runs, it seems only appropriate for symmetric software systems and symmetric specifications. By "symmetric," we mean that for an executing process, all other process's identifers are the same. There is no next process identifiers like in a ring network. Also no special identifier constants served to distinguish leaders.

However, there is a simple translation scheme which makes DDD technology also capable of verifying asymmetric systems. Suppose we have a concurrent real-time software systems of m processes which are running different programs (automata) $A_1, \ldots, A_m$. The trick is in the dscription of automata, we can construct a union of the process automata to represent the system behavior. If the i'th process automaton, $1 \leq i \leq m$, has n_i operation modes, then in the union automaton A of $S = \langle A, m, I \rangle$, the operation modes of A_i will be indexed sequentially as

$$(\textstyle\sum_{1 \leq j < i} n_j), \quad (\sum_{1 \leq j < i} n_j) + 1, \quad \ldots, \quad (\sum_{1 \leq j < i} n_j) + n_i - 1$$

In this way, the asymmetricity among processes are totally distinguished by the range of values that local discrete `mode` can have even after arbitrary process identifier permutations.

For example, we have a ring network mutual exclusion algorithm in figure 2(a). Here t is the local discrete variable for token. Because in the upward transition local variable of a peer process is modeified ($t[p\%m+1] := 1;$), the system is not in the framework of our real-time software systems. After making the union of the process automata, we can construct the new automaton in figure 2(b) which is not connected and is partitioned into three connected components corresponding to the three process automata. Also the local token declaration is now redeclared as global. Of course, initial condition I has to be:

$$\texttt{mode}[1] = 0 \wedge x[1] = 0 \wedge \texttt{mode}[2] = 2 \wedge x[2] = 0 \wedge \texttt{mode}[3] = 4 \wedge x[3] = 0$$

As for the complexity incurred by the scheme in this section, if we have m processes, the total number of auxiliary binary variables used in the DDD is $m \log m$ and is very close to linear complexity w.r.t. the concurrency size.

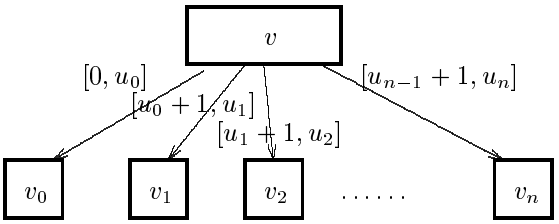


Figure 3: Data structure implementation of a node in DDD

4 Data Decision Diagram

DDD is a data-structure for representing set of normalized states. In the implementation aspect, it resembles CDD[7] and each node in DDD has the structure shown in figure 3. Such a node is used to evaluate the truth value of a formulus from variable v. The outgoing arcs are labeled with lower and upper bounds of **integer parts** (note, only integer parts) of values of variable v and direct to the DDD's for the subformulae true in the corresponding ranges of v's values. For example, if the second arc from left in figure 3, is labeled with $[7, 9]$, then subformulus represented by the DDD rooted at v_1 must be true when the integer part of v's value is in $[7, 9]$. The ranges lobeled on arcs from a DDD node are required to be disjoint.

The lower and upper bounds of outgoing arcs are chosen according to the following rules. For clock variables, we define constant $O_S = C_S + 1$ which symbolically denotes a clock reading greater than C_S. Thus the lower and upper bounds of outgoing arcs from DDD nodes of clock variables are chosen from elements in $\{0, \ldots, C_S\} \cup \{O_S, \infty\}$ which is sufficient for the regions construction in [2]. For discrete variables, the lower and upper bounds of outgoing arcs are chosen from those constants assigned to and compared with the variable in both the automaton and the specification. For pointer variables, the lower and upper bounds of outgoing arcs are chosen from *NULL* and integers 1 through m.

Given an automaton A with variable name set X, the variable set in the DDD for an m process system is

$$\{x \mid x \in X; x \text{ is global}\} \cup \{x[i] \mid x \in X; x \text{ is local}; 1 \leq i \leq m\} \cup \{\kappa[i] \mid 1 \leq i \leq m\}$$

Here κ is the name for an auxiliary local binary discrete variable name used to encode the fractional part orderings among clock readings. According to Alur et al's region graph construction [2], the state-equivalence relation for model-checking is determined by the following three factors:

- the discrete information of each state,
- the integer parts of clock readings $\leq C_S$.
- the ordering among the fractional parts of clock readings $\leq C_S$.

Our innovation is that we use one bit (κ) for each clock to encode the ordering among the fractional parts of clock readings in normalized states. For each clock, say $x[i]$ of process i, $\kappa[i]$ is true in a normalized state s if and only if $s(x[i]) \leq C_S$ and either

- $i = 1$ and $\mathit{frac}(s(x[i])) = 0$, i.e., $s(x[i])$ is an integer; or
- $i > 1$ and $\mathit{frac}(s(x[i-1])) = \mathit{frac}(s(x[i]))$, i.e., the fractional parts of $s(x[i-1])$ and $s(x[i])$ are the same.

With such definition of data-structure and appropriate permutations of process identifiers after clock reading advancements and clock reset operations, we are able to represent the regions of timed automata [2]. As for the other input variables, local or global, we simply copy them as the variables in our DDD. Thus given a realt-time software system $S = \langle A, m, I \rangle$ with $A = \langle X, \lambda, Q, \mu, E, \tau, \pi \rangle$, the number of variables used in our DDD is $O(m|X| + m)$.

For example, we may have 8 processes in a normalized state with $x[1] = 0, x[2] = 3, x[3] = 1.3, x[4] = 1.456, x[5] = 9.456, x[6] = 20.7, x[7] = 38, x[8] = 10\pi$ while $C_S = 13$. The readings of clocks and values of $\kappa[i]$'s are shown in the following.

$DDDop(op, D_x, D_y)$\{
(1) if op= AND,
(1) if D_x is *true*, return D_y;
(2) else if D_y is *true*, return D_x;
(3) else if either D_x or D_y is *false*, return *false*;
(2) else if op= OR,
(1) if either D_x or D_y is *true*, return *true*;
(2) else if D_x is *false*, return D_y;
(3) else if D_y is *false*, return D_x;
(3) else \{
(1) Construct a new DDD node D with $D.v = \min(D_x.v, D_y.v)$;
(2) if $D_x.v < D_y.v$, then for each outgoing arc $[l, u]D_c$ of D_x,
add an outgoing arc $[l, u]DDDop(op, D_c, D_y)$ to D.
(3) else if $D_x.v > D_y.v$, then for each outgoing arc $[l, u]D_c$ of D_y,
add an outgoing arc $[l, u]DDDop(op, D_x, D_c)$ to D.
(4) else for each outgoing arc $[l_x, u_x]D'_x$ of D_x and outgoing arc $[l_y, u_y]D'_y$ of D_y,
if $[\max(l_x, l_y), \min(u_x, u_y)]$ is nonempty,
add an outgoing arc $[\max(l_x, l_y), \min(u_x, u_y)]DDDop(op, D'_x, D'_y)$ to D.
(5) Merge any two outgoing arcs $[l, u]D_c, [u + 1, u']D_c$ of D into one $[l, u']D_c$
until no more merge can be done.
(6) if D has more than one outgoing arcs, return D,
else return the sole subformulus of D;
\}
\}

Table 1: Algorithm for computing $D_x \, op \, D_y$

i	1	2	3	4	5	6	7	8
$x[i]$	0	3	1.3	1.456	9.456	20.7	38	10π
$\kappa[i]$	*true*	*true*	*false*	*false*	*true*	*false*	*false*	*false*

5 Manipulations on DDD

5.1 Boolean operations

The Boolean operations on DDD's follow the same style in Bryant's BDD manipulations[5, 7, 8]. We present procedure $DDDop(op, D_x, D_y)$ in table 1 to illustrate the idea in the implementation of such an operation. For convenience, we use $[l, u]D$ to denote an outgoing arc whose lower bound is l, upper bound is u, and the subformulus DDD is D. Also, $D.v$ denotes the index, of D's variable, used in the variable-ordering of the DDD.

We should point out that the algorithm shown in table 1 is for simplicity and clarity of algorithm presentation and is not for efficiency. Our implementation is more efficient in that it records which pairs of D_x, D_y have already been processed. If a pair of D_x, D_y has already been processed with procedure $DDDop()$ before, then we simply return the result recorded in the first time when such a pair was processed.

5.2 Preserving normality after transitions and clock reading advancement

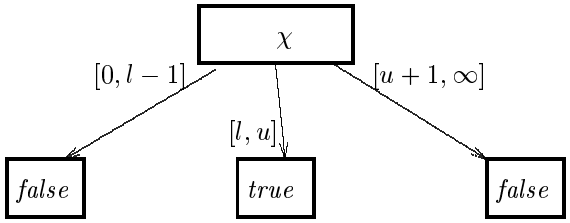
DDD are data-structures for normalized states of real-time software systems. Transitions and clock reading advancements may change normalized states into unnormalized states. This section tells us how to symbolically do necessary process identifier permutation to preserve the normality of states.

Given a local variable $v[i]$ of process i, we assume that $VarIndex(i, v)$ is the variable index of $v[i]$ in DDD. For convenience, we use process identifier 0 (NULL) for those global variables. That is $VarIndex(0, v)$ returns a valid

variable index in DDD only when v is a global variable. As can be seen from the process identifier permutation scheme in subsection 2.3, segment movements are performed. We can thus define the following function which computes the new identifier of process i after such a segment movement.

NewProcId$(i, j, k,$ *disp*$)$ /* process identifiers j through k are to be moved with displacement *disp*. */ {
(1) if $i = 0$, return 0; /* Global is always global. */
(2) else if $j \leq i \leq k$, return$(i +$ *disp*$)$;
(3) else if $i < j +$ *disp*, return(i);
(4) else if $j +$ *disp* $\leq i < j$, return$(i + k - j + 1)$;
(5) else if $k < i \leq k +$ *disp*, return$(i - (k - j + 1))$;
(6) else /* $k +$ *disp* $< i$ */ return(i);
}

Due to page-limit, we shall only describe the algorithm for symbolic manipulation of a procedure *ToInt*(R), in table 2 in page 10, which generates a new DDD describing the set of states obtained from those in R by advancing those clocks with biggest fractional parts to integers. In the algorithm presentation, for simple clarity, we use Boolean operation symbols like $\vee, \wedge$ to represent our procedure DDDop(). Also for an atom like $l \leq \chi \leq u$, it is implemented by constructing the following DDD.



Of course, when $[0, l-1]$ (or $[u+1, \infty]$) is empty, the corresponding arc disappears. Symbolic manipulation procedures for next state set after transitions and clock reading advancement from integers to fractionals can all be defined similarly.

6 Experiments

We have experimented to compare DDD technology with previously published performance data in two reports[4, 6] that compared performances of various model-checkers respectively on two versions of Fischer's timed mutual exclusion algorithm as shown in Figure 4. UPPAAL's version has bigger timing constants while Balarin's version allows repetitions.

UPPAAL is based on DBM technology and has been well accepted as one of the most efficient model-checkers for real-time systems. It has been used to successfully verify many communication protocols. Recently, CDD technology was proposed to enhance the performance of UPPAAL[7]. However, further reports are yet to be seen. In [6], UPPAAL was compared with many other model-checkers like HyTech's[3], Epsilon, and Kronos[13] on the automaton in figure 4(a). The experiments was reported to be performed on a Sparc-10 with 128 MB memory (real plus swap). All other tools fail when the number of processes reaches beyond 5 while UPPAAL can verify the algorithm on 8 processes.

We have implemented two version of DDD on an Pentium II 366 MHz IBM notebook with 256 MB memory (real plus swap) running Linux. The tools are avaiable at:

`http://www.iis.sinica.edu.tw/~farn/ddd`

The first version is plain while the second version employs the clock-shielding reduction technique reported in [12, 14]. Reduction clock-shielding replaces clock readings with ∞ in a state when along any runs from the sate, it is determined that such a reading will no longer be read unless the clock is reset. The performance is listed in the following table.

p

ToInt(R) /* R describes the state set before advancing those clock readings with biggest fractional parts to integers. */ {

(1) $D := \mathit{false}$;

(2) For $1 \le i \le m$ and $i \le j \le m$, do {

(1) Construct the condition K that

- there is no clock whose reading is an integer $\le C_S$; and
- processes i to j are those with biggest fractional parts in their clock readings.

(2) $D := D \vee \mathit{RecToInt}(K \wedge R, i, j)$;

}

(3) Return(D);

}

RecToInt(K, i, j) {

(1) If K is either *true* or *false*, return K;

(2) Get the process identifier k of $K.v$; /* $k = 0$ when v is global. */

(3) Generate the name χ of variable $K.v$ of process $\mathit{NewProcId}(k, i, j, 1-i)$);

(4) $K' := \mathit{false}$;

(5) switch on type of $K.v$ {

(6) case LOCAL_CLOCK:

(1) if $i \le k \le j$, then for each outgoing arc $[l, u]D'$ of K,
$K' := K' \vee (l + 1 \le \chi \le u + 1 \wedge \mathit{RecToInt}(D', i, j))$;
else for each outgoing arc $[l, u]D'$ of K,
$K' := K' \vee (l \le \chi \le u \wedge \mathit{RecToInt}(D', i, j))$;

(2) return(K');

(7) case $\kappa[k]$:

(1) if $\mathit{NewProcId}(k, i, j, 1-i) = 1$,

(1) then with outgoing arcs $[l_1, u_1]D_1, \ldots, [l_n, u_n]D_n$ of K,
return($\kappa[1] = \mathit{true} \wedge \bigvee_{1 \le h \le n} \mathit{RecToInt}(D_h, i, j)$);

(2) else for each outgoing arc $[l, u]D'$ of K,
$K' := K' \vee (l \le \kappa[\mathit{NewProcId}(k, i, j, 1-i)] \le u \wedge \mathit{RecToInt}(D', i, j))$;

(2) return(K');

(8) case LOCAL_Discrete:

(9) case GLOBAL_Discrete:

(1) for each outgoing arc $[l, u]D'$ of K,
$K' := K' \vee (l \le \chi \le u \wedge \mathit{RecToInt}(D', i, j))$;

(2) return(K');

(10) case LOCAL_POINTER:

(11) case GLOBAL_POINTER:

(1) for each outgoing arc $[l, u]D'$ of K and each $l \le h \le u$, {

(1) $g := \mathit{NewProcId}(h, i, j, 1-i)$;

(2) $K' := K' \vee (\chi = g \wedge \mathit{RecToInt}(D', i, j))$;

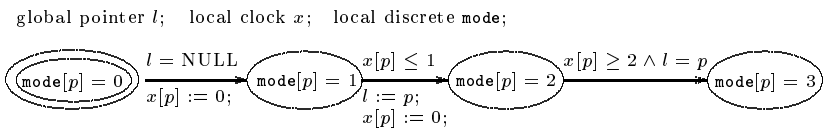
}

(2) return(K');

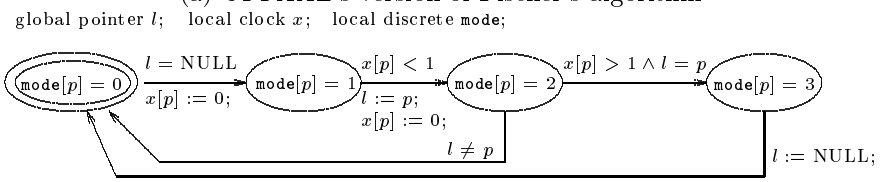
}

}

Table 2: Symbolic manipulation procedure for clock-reset transitions



(a) UPPAAL's version of Fischer's algorithm



(b) Balarin's version of Fischer's algorithm

Figure 4: Two versions of Fischer's algorithm for experiments

version	resources	2	3	4	5	6	7	8	9	10	11	12	13
no CS	time	0.19	1.22	6.5	27	105	320	888	2323	5556	N/A	N/A	N/A
	space	50	212	697	1959	4885	10966	22444	42858	77503	N/A	N/A	N/A
CS	time	0.17	0.98	4.4	16.5	50.3	134	325	724	1493	3002	5743	10152
	space	47	161	463	1134	2456	4810	8871	15726	26194	41930	64323	95389

"CS" means the version with clock-shielding reduction while "no CS" means the one without. The CPU time is measured in seconds. The space is measured in kilobytes and only includes those for the management of DDD's and 2-3 trees. "N/A" means "not available" which indicates that the corresponding experiment has not been performed.

The time consumption is considerable bigger than that of UPPAAL[6] considering the CPU clock rate difference. This is due to our implementation philosophy. We believe time is an unbounded resource while space is not. As can be seen from procedure ToInt(), no DDD for the relation between current state and next state is computed. In practice, such a relation in DDD can occupy a great many bytes. The next-state set DDD is computed by analysis on different situations of i and j. The sacrifice in CPU time pays off in the memory space efficiency. With twice the memory size used in [6], we are now able to verify the simplified Fischer's algorithm with 13 processes.

UPPAAL is a mature tool with a great arsenal of reduction technologies implemented. Our software at this moment only relies on minimal canonicality of DDD to gain performance. Please note that the exponent base in our data seems to decrease with respect to concurrency size. This may imply that fully symbolic manipulation is more suitable for large system verification. In the future, with more reduction technique implemented for DDD, we hope even more performance improvements will be observed. For example, the clock-shielding reduction indeed slowers down the state-space explosion problem exponentially. Still several simple reduction, like getting rid of FALSE terminal nodes in DDD, can be implemented in the future version of DDD to get constant factor of improvement.

In [4], weak and strong approximation technologies of symbolic verification are proposed and experiments are performed on algorithm in figure (b). We extend the performance data table in [4] to compare our tool with previous technologies.

#proc	strong	weak	KRONOS	Wong-Toi	DDD(no CS)
6	155sec	18sec	1174sec	74sec	26sec/1374k
7	398sec	48sec	M/O	164sec	67sec/2488k
8	986sec	116sec	M/O	375sec	142sec/4242k
9	2220sec	247sec	M/O	891sec	303sec/6858k
10	M/O	576sec	N/A	N/A	558sec/10659k
11	N/A	N/A	N/A	N/A	1034sec/15673k
12	N/A	N/A	N/A	N/A	1724sec/22251k
13	N/A	N/A	N/A	N/A	2889sec/30593k
14	N/A	N/A	N/A	N/A	4492sec/41019k
15	N/A	N/A	N/A	N/A	7047sec/53737k
16	N/A	N/A	N/A	N/A	10782sec/69126k
17	N/A	N/A	N/A	N/A	15330sec/87431k

The original table consists of the firt five columns and only reports the CPU time in seconds used by various tools. "M/O" means "out of memory." KRONOS is also based on DBM technology while Wong-Toi's tool is based on approximation. In our extension, each entry is composed of both CPU time (in seconds) and space (in kilobytes) used. The column extension is for time/space consumed without clock-shielding reduction. Balarin's experiment is performed on Sparc 2 with 128 MB memory while ours is performed on IBM Thinkpads with PII 366 MHz and 256 MB memory. The extended table shows that DDD indeed control state-space explosion better.

7 Conclusion

We propose to use one auxiliary binary variable for each clock in our new data-structure for fully symbolic model-checking of real-time software systems. Since we now have fewer variables in the fully symbolic manipulation, theoretically we can expect better verification performance. With better implementation of reduction techniques borrowed from BDD technology, we are hoping for further performance improvement.

References

[1] Asaraain, Bozga, Kerbrat, Maler, Pnueli, Rasse. Data-Structures for the Verification of Timed Automata. Proceedings, HART'97, LNCS 1201.

[2] R. Alur, C. Courcoubetis, D.L. Dill. Model Checking for Real-Time Systems, IEEE LICS, 1990.

[3] R. Alur, T.A. Henzinger, P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. in Proceedings of 1993 IEEE Real-Time System Symposium.

[4] F. Balarin. Approximate Reachability Analysis of Timed Automata. IEEE RTSS, 1996.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond, IEEE LICS, 1990.

[6] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, Wang Y. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. Hybrid Control System Symposium, 1996, LNCS, Springer-Verlag.

[7] G. Behrmann, K.G. Larsen, J. Pearson, C. Weise, W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. CAV'99, July, Trento, Italy, LNCS 1633, Springer-Verlag.

[8] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., C-35(8), 1986.

[9] E. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, Proceedings of Workshop on Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[10] D.L. Dill. Timing Assumptions and Verification of Finite-state Concurrent Systems. CAV'89, LNCS 407, Springer-Verlag.

[11] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic Model Checking for Real-Time Systems, IEEE LICS 1992.

[12] P.-A. Hsiung, F. Wang. User-Friendly Verification. Proceedings of 1999 FORTE/PSTV, October, 1999, Beijing. Formal Methods for Protocol Engineering and Distributed Systems, editors: J. Wu, S.T. Chanson, Q. Gao; Kluwer Academic Publishers.

[13] X. Nicolin, J. Sifakis, S. Yovine. Compiling real-time specifications into extended automata. IEEE TSE Special Issue on Real-Time Systems, Sept. 1992.

[14] F. Wang, P.-A. Hsiung. Automatic Verification on the Large. Proceedings of the 3rd IEEE HASE, November 1998.

[15] H. Wong-Toi. Symbolic Approximations for Verifying Real-Time Systems. Ph.D. thesis, Stanford University, 1995.