Symbolic Model Checking for Event-Driven Real-Time Systems

JIN YANG and ALOYSIUS K. MOK
The University of Texas at Austin
and
FARN WANG
Academia Sinica

In this article, we consider symbolic model checking for event-driven real-time systems. We first propose a Synchronous Real-Time Event Logic (SREL) for capturing the formal semantics of synchronous, event-driven real-time systems. The concrete syntax of these systems is given in terms of a graphical programming language called Modechart, by Jahanian and Mok, which can be translated into SREL structures. We then present a symbolic model-checking algorithm for SREL. In particular, we give an efficient algorithm for constructing OBDDs (Ordered Binary Decision Diagrams) for linear constraints among integer variables. This is very important in a BDD-based symbolic model checker for real-time systems, since timing and event occurrence constraints are used very often in the specification of these systems. We have incorporated our construction algorithm into the SMV v2.3 from Carnegie-Mellon University and have been able to achieve one to two orders of magnitude in speedup and space saving when compared to the implementation of timing and event-counting functions by integer arithmetics provided by SMV.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; *specification techniques*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computational logic*

General Terms: Languages, Verification

Additional Key Words and Phrases: Binary decision diagrams, real-time verification, symbolic model checking, synchronous real-time event logic

This article was supported in part by a research grant from the Office of Naval Research under ONR contract number N00014-94-1-0582 and in part by a research grant from the Texas Advanced Research Program. An extended abstract appears in Proceedings of the 14th IEEE Real-Time Systems Symposium.

Authors' addresses: J. Yang and A. K. Mok, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78721; email: {jinyang; mok}@cs.utexas.edu; F. Wang, Institute of Information Science, Academia Sinica, Nankang, Taipei 11529, Taiwan, Republic of China; email: farn@iis.sinica.edu.tw.

1. INTRODUCTION

Real-time systems are different from traditional finite-state systems in that they often have to meet hard real-time constraints in addition to functional correctness requirements. Many efforts have been carried out in recent years to automate the verification of real-time systems. An early work in this area is by Jahanian, Mok, and Stuart as a part of the SARTOR project [Mok 1985], which is to build an environment for the development of correct real-time software. They extended the language STATECHART [Harel 1986] with constructs for specifying timing constraints whose semantics is given in RTL (Real Time Logic) [Jahanian and Mok 1986], a first-order logic based on Presburger arithmetic which is specialized for specifying timing properties of event-based real-time systems. Jahanian and Stuart [1988] and Stuart [1990] proposed and subsequently implemented an algorithm for verifying timing properties of real-time systems using computation graphs. In a more recent development, Alur et al. [1990] and Henzinger et al. [1991] have considered extending temporal logic by introducing a "freeze" operator. Alur et al. [1990] successfully applied the model-checking technique for finite-state machines [Clarke and Emerson 1981; Clarke et al. 1986] to the verification of real-time systems based on the region graph approach.

However, the application of model checking is limited by the state explosion problem, which is even more severe in the presence of timing predicates. For this reason, many researchers have focused on symbolic model-checking methods [Burch et al. 1990; Clarke et al. 1991; Coudert et al. 1989; Emerson and Clarke 1980; Sistla 1982]. Symbolic model checking has proven to be very successful for non-real-time systems because of the effectiveness of Binary Decision Diagram (BDD) representation of the state transition relation. In recent years, the symbolic model-checking approach has been applied to real-time system verification. Among many efforts are Henzinger et al. [1991], Yang et al. [1993], Alur et al. [1993], and Campos et al. [1994].

In this article, we consider the symbolic verification of event-driven real-time systems over the discrete time domain. We choose to use the nonnegative integers for the domain of time because our work is aimed at verifying computer systems, which are discrete time devices. Except at integral time boundaries, change in internal state of discrete time devices is unobservable and indeed should not affect the environment external to the device. We also choose an event-based model to emphasize what we intend to be observable about a system. In our framework, an event is meant to denote I/O behavior or a significant change in internal system states. Events occur instantaneously at discrete time points, and the same event may occur multiple times (i.e., have multiple instances) in the course of a computation. An event-driven real-time system must respond to certain events with appropriate actions and within certain hard deadlines. Most control systems can be viewed as event-driven real-time systems. We shall adopt a graphical language, Modechart [Jahanian and Mok 1986], to

provide a concrete syntax for event-driven real-time systems. For formal specification and verification, we propose a logic, Synchronous Real-Time Event Logic (SREL). SREL can be viewed as a variation of Asynchronous Real-Time Event Logic (AREL) [Wang and Mok 1992]. SREL captures the semantics of synchronous systems where the occurrence of events is observable only at integral (discrete) time boundaries. In SREL computations, there is no temporal precedence among event instances that have the same time of occurrence. We shall show a symbolic model-checking algorithm for SREL. Since the representation of timing and event occurrence constraints is critical in the BDD implementation of a symbolic model-checking algorithm for real-time systems, we propose an efficient BDD algorithm for constructing BDDs for linear constraints among integer variables. We have incorporated this scheme into SMV v2.3 [McMillan 1992] from Carnegie-Mellon University and have been able to save one to two orders of magnitude in time and space compared to the integer implementation in the original system.

Unlike the work by Alur et al. [1993], in which they deal with general hybrid real-time systems, we focus on a restricted class of real-time systems with only linear constraints and over the domain of integers. This gives us the benefit of reducing the problem complexity and finding more efficient verification algorithms and at the same time retaining the coverage of a wide range of real-time systems. Our approach reasons symbolically about timing constraints among event occurrences which are very important in real-time systems but have been largely ignored in the past. Such a capability is also important for many hardware verification problems, e.g., when we want to match outputs of a pipeline with the inputs of the pipeline.

To the best of our knowledge, our algorithm for building BDDs for linear constraints is the first efficient solution to overcome the state explosion problem during the BDD-building process for such constraints. It is often the case that state explosion only happens when the intermediate results are being constructed. In 1995, Clark et al. [1995] proposed the Hybrid Decision Diagram that combines the Binary Moment Diagram (BMD), the BDD, and the Multiterminal BDD (MTBDD) to combat the complexity in arithmetic operations. Like our approach, the effectiveness of their approach relies on the fact that the relations among integer variables can be reduced to a set of linear constraints. Although their algorithm can deal with a wider range of integer operations such as multiplication, our approach is faster and more efficient to solve linear constraints because it is **hard-coded** and does not need to go through BMD and MTBDD.

The rest of the article is organized as follows. In Section 2, we propose a computation model for event-driven real-time systems. In Sections 3 and 4, we define the syntax and semantics of SREL. We show that although the satisfiability problem for SREL is undecidable in general, we observe that most practical real-time systems obey certain bounded progress conditions. The bounded progress conditions enable us to verify properties of most practical systems. In Section 5, we introduce the real-time system program-

ming language Modechart. We then define its semantics in terms of computation models. In Section 6, we show how to build a quotient computation model for a modechart, given an SREL formula to be verified, and propose a model-checking algorithm. In Section 7, we show how to symbolically model check a quotient computation model against an SREL formula. In Section 8, we propose a fast algorithm for building BDDs for linear constraints, which is very important in BDD-based implementation of symbolic model-checking procedures. We show some experiment results. We conclude the article in Section 9.

2. A COMPUTATION MODEL FOR EVENT-DRIVEN REAL-TIME SYSTEMS

From the control perspective, we view a real-time system as operating in *modes* [Jahanian and Mok 1986; Jahanian and Stuart 1988]. A mode is a control state of the system. The system changes from one mode to another when either some events trigger the transition, or some timing condition is satisfied. An *event* is an instantaneous change in the system that may happen repeatedly over time. It could be an external signal, or a toggling of a state variable. A mode can also be defined by a pair of events: one indicates entering the mode, and the other indicates leaving the mode. An instance of an event serves as a maker in time. Thus, the timing properties of the system can be easily and accurately specified as the timing constraints among instances of events. A well-known example is the following railroad crossing control example from Leveson and Stolzy [1985], Jahanian and Stuart [1988], and Stuart [1990].

Example 2.1 (*Railroad Crossing Control System*)

> The gate at a guarded railroad crossing is to be software controlled, and since the gate cannot control the train, a real-time solution is needed. There is an early warning signal at a distance from the crossing that gives notice to the gate controller that a train is approaching, and it is known that it takes the train at least 300 time units to reach the crossing from the signal. It is also known that the time required to lower the gate is between 20 and 50 time units. The controller itself can detect the departure of the train, and it requires between 20 and 100 time units to raise the gate. It is also known that trains are scheduled so that it takes at least 100 time units from the time a train leaves the crossing until the next train reaches the early warning signal [Stuart 1990].

The behavior of such a real-time system can be characterized by the set of all possible sequences of event instances that happen over time. Figure 1 shows such a sequence.

The horizontal axis in Figure 1 represents time. A name following (followed by) an arrow denotes a mode entry (exit) event. An event with a subscript denotes an instance of the event. We treat the event instances that happen at the same time as truly concurrent, although there might be some causality relation among these instances. Such a granularity is justified, since we are concerned about the real-time constraints over the integer time domain.

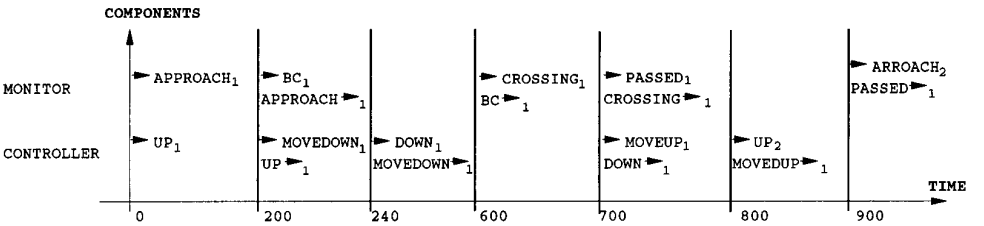

Fig. 1. An event sequence of the railroad crossing system.

Formally, we view a real-time system as a triple $S = (E, \Gamma, \mathcal{R})$ where (1) E is a finite set of boolean variables called *events*, (2) Γ is a finite set of variables called *timers* with values over N, the set of nonnegative integers, and (3) $\mathcal{R} : E \rightarrow 2^{\Gamma}$ *is a reset* function that associates a set of timers with each event.

A *time point* p of the system at real time t is defined by three assignments $\mathcal{O}_p : E \rightarrow Boolean$, $\mathcal{I}_p : E \rightarrow N$, and $\mathcal{T}_p : \Gamma \rightarrow N$. For every event $e \in E$, $\mathcal{O}_p(e)$ indicates whether the event is happening at the time point, and $\mathcal{I}_p(e)$ is the number of instances of the event that have happened since the beginning. For every timer $\tau \in \Gamma$, $\mathcal{T}_p(\tau)$ is the time value in the timer. We do not allow an event to happen more than once at any time point.

Definition 2.2 (*Computation Model*). The *computation model* for a real-time system $S = (E, \Gamma, \mathcal{R})$ *is a tuple* $M = (S, \Xi)$ where Ξ is a set of infinite time traces such that

(1) Every time trace $\pi = p_0 p_1 \ldots$ satisfies the following two conditions:
   (a) *Initial Condition*: For every event $e \in E$, $\mathcal{I}_{p_0}(e) = 1$ if $\mathcal{O}_{p_0}(e) = true$. Otherwise, $\mathcal{I}_{p_0}(e) = 0$. For every timer $\tau \in \Gamma$, $\mathcal{T}_{p_0}(\tau) = 0$.
   (b) *Successor Condition*: For every $i > 0$, (1) for every event $e \in E$, $\mathcal{I}_{p_i}(e) = \mathcal{I}p_{i-1}(e) + 1$ if $\mathcal{O}_{p_i}(e) = true$; $\mathcal{I}_{p_i}(e) = \mathcal{I}_{p_{i-1}}(e)$ otherwise; and (2) for every timer $\tau \in \Gamma$, $\mathcal{T}_{p_i}(\tau) = 0$ if $\mathcal{O}_{p_i}(e) = true$ and $\tau \in \mathcal{R}(e)$ for some $e \in E$; $\mathcal{T}_p(\tau) = \mathcal{T}_{p_{i-1}}(\tau) + 1$ otherwise.
(2) *History-Free*: Let $\pi = p_0 p_1 \ldots p_k \ldots$, $\pi' = q_0 q_1 \ldots q_l \ldots$ be two time traces in Ξ such that $p_k = q_l$. Then both $p_0 p_1 \ldots p_{k-1} q_l q_{l-1} \ldots$ and $q_0 q_1 \ldots q_{l-1} p_k p_{k+1} \ldots$ are also in Ξ.

The number of time traces in Ξ in the definition could be either finite or infinite. It is also clear that there is no loop in any time trace, meaning that no two time points in the trace are the same, since whenever a timer is reset to zero, an event must occur; and consequently the number of occurrences of the event must be incremented by one.

Since the set of traces is *history-free*, we can define Ξ in terms of an *initial-point* set P_0 and a *next-point* function $\mathcal{N} : P \rightarrow 2^P$, where P is the set of all time points, such that

—$P_0 \subseteq P$ contains the first time point of every time trace and
—for every point $p \in P$, a point $q \in P$ is in $\mathcal{N}(p)$ iff for any time trace in Ξ that contains p, q is the successor of p in the trace.

Based on the next-point function, we define a time point sequence $p_0 p_1 p_2 \ldots$ as a *p-trace* for a time point p if $p_0 = p$ and $p_{i+1} \in \mathcal{N}(p_i)$.

By the nature of the time trace set Ξ, we immediately conclude that the next-point function is infinite and does not imply a loop. Throughout the article, we shall interchangeably use the set of time traces and the tuple of the initial-point set and the next-point function to define the behavior of a real-time system.

3. SYNCHRONOUS REAL-TIME EVENT LOGIC (SREL)

SREL is specially designed for reasoning about timing constraints among event instances and can be seen as a variation of AREL [Wang and Mok 1992] for synchronous systems. For each event e, SREL uses predicate e to represent whether the event is happening, and it uses counter $\#e$ to represent the number of instances that have happened so far for the event. Thus, for instance, expression $e \wedge \#e = 5$ indicates that the 5th instance of event e is happening at the current moment.

Given a set of events E, let $\#E$ denote $\Sigma_{e \in E}(\#e)$, the total number of instances that have happened for the events in E. We call $k\#E$ a *counting term* of E with coefficient k, where k is a nonnegative integer. Intuitively, E defines a set of events that are of the same type. It is often the case that the same type of stimulus events require the same type of responses. The coefficients are used to compare sets of events that happen at different rates.

Let R be a finite set of nonnegative integer-valued variables called *registers*. Additionally, each register can have an undefined value *undef*. The formulas ϕ of SREL are inductively defined as follows.

Definition 3.1 (*SREL*)

$$\begin{aligned}
\phi ::= \; & e \mid \\
& k_1\#E_1 - k_2\#E_2 \sim c \mid k_1\#E_1 - r \sim c \mid \\
& \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \\
& \{r := k_2\#E_2\}.\phi_1 \mid \\
& AF[l, u]\phi_1 \mid EF[l, u]\phi_1,
\end{aligned}$$

where c is an integer constant; $r \in R$; $\sim$ is one of $<$, $\leq$, $=$, $\geq$, $>$; and l, u are two nonnegative integers such that $l \leq u$. We also allow u to be ∞.

We shall use $c \leq k_1\#E_1 - k_2\#E_2 \leq d$, $\phi_1 \Rightarrow \phi_2$, $AG[l, u]\phi_1$, and $EG[l, u]\phi_1$ as shorthands, respectively, for $k_1\#E_1 - k_2\#E_2 \geq c \wedge k_1\#E_1 - k_2\#E_2 \leq d$, $\neg\phi_1 \vee \phi_2$, $\neg EF[l, u]\neg\phi_1$, and $\neg AF[l, u]\neg\phi_1$. When the time range for an AF or EF operator is $[0, \infty]$, we shall omit the subscript. We shall also replace $\#E$ by $\#e$ when $E = \{e\}$.

The *snapshot operator* $\{(r := k_2\#E_2)\}$ saves the value of the counting term $k_2\#E_2$ into register r. The value will be kept in r unless r is set to some other value. Thus, $\{(r := k_2\#E_2)\}.\phi$ states that if we let r have value $k_2\#E_2$, ϕ will hold. We require that all SREL formulas do not contain *free* registers, i.e., any comparison involving a register has to be in the scope of some snapshot operator for the register.

Formula $AF[l, u]\phi_1$ requires that along every trace from the current time t_0, ϕ_1 will hold eventually at some future time t satisfying $l \leq t - t0 \leq u$. Formula $EF[l, u]\phi_1$ only requires the same property hold along some trace.

Now let us look at some specification examples using SREL.

Example 3.2 (*A Missile System*). A missile system consists of three missile firing units and two radar units. Event $fire_i$ denotes the event that a missile is fired by the ith firing unit. Event loc_j denotes the event that a target is located by the jth radar unit.

Let $Fire = \{fire_1, fire_2, fire_3\}$ and $Loc = \{loc_1, loc_2\}$. We can specify the following properties.

—The safety property: At any moment, the number of missiles fired is no more than the number of targets located so far, and the difference is within 2.

$$AG(0 \leq \#Loc - \#Fire \leq 2).$$

—The timeliness property: A missile must be fired within six time units once a target is found.

$$AG(\{r := \#Loc\}.AF[0, 6](\#Fire - r \geq 0)).$$

4. SATISFIABILITY AND BOUNDED PROGRESS CONDITION

Now we give a formal semantics to SREL. We define an *environment* $\mathscr{E}$ as an assignment to all the registers in $\mathscr{R}$, and let $\mathscr{E}(r)$ be the value of register r in $\mathscr{E}$. Environment $\mathscr{E}[r := k_2\#E_2]$ is identical to environment $\mathscr{E}$ except that r has the current value of the counting term $k_2\#E_2$. For simplicity, in the rest of the article, we shall assume that the real-time system $S = (E, \Gamma, \mathscr{R})$ we want to verify is understood. Therefore, we shall omit S from the computation model (S, Ξ) for S and simply write the model as Ξ.

Definition 4.1 (*Satisfiability*). Given a computation model $\Xi = (P_0, \mathscr{N})$ and an SREL formula ϕ, a time point p in Ξ *satisfies* ϕ under an environment $\mathscr{E}$, denoted by $(\Xi, p) \models_{\mathscr{E}} \phi$ or simply $p \models_{\mathscr{E}} \phi$ if the model is understood, if one of the following conditions holds.

—$p \models_{\mathscr{E}} e$ iff $\mathscr{G}_p(e) = true$;
—$p \models_{\mathscr{E}} k_1\#E_1 - k_2\#E_2 \sim c$ iff $k_1\Sigma_{e\in E_1}\mathscr{F}_p(e) - k_2\Sigma_{e\in E_2}\mathscr{F}_p(e) \sim c$;
—$p \models_{\mathscr{E}} k_1\#E_1 - r \geq c$ iff $k_1\Sigma_{e\in E_1}\mathscr{F}_p(e) - \mathscr{E}(r) \sim c$;
—$p \models_{\mathscr{E}} \neg\phi_1$ *iff* $p \not\models_{\mathscr{E}} \phi_1$;

—$p \models_{\mathscr{E}} \phi_1 \vee \phi_2$ iff $p \models_{\mathscr{E}} \phi_1$ *or* $p \models_{\mathscr{E}} \phi_2$;
—$p \models_{\mathscr{E}} \{r := k_2 \# E_2\}.\phi$ iff $p \models_{\mathscr{E}[r:=k_2\#E_2]} \phi$;
—$p \models_{\mathscr{E}} AF[l, u]\phi_1$ iff for every p-trace $p_0 p_1 \ldots$, i in range $[l, u]$, $p_i \models_{\mathscr{E}} \phi_1$;
—$p \models_{\mathscr{E}} EF[l, u]\phi_1$ iff there is a p-trace $p_0 p_1 \ldots$, i in range $[l, u]$, $p_i \models_{\mathscr{E}} \phi_1$.

We say model $\Xi = (P_0, \mathcal{N})$ *satisfies* ϕ iff for environment $\mathscr{E}_{undef}$ that assigns *undef* to every register and any time point $p_0 \in P_0$, $p_0 \models_{\mathscr{E}} \phi$. Formula ϕ is *satisfiable* iff there exists a computation model Ξ which satisfies ϕ.

THEOREM 4.2 (SATISFIABILITY PROBLEM). *The satisfiability problem for SREL is undecidable.*

PROOF. The theorem is proved by applying a reduction from the halting problem of two-counter turing machines [Lewis 1979]. Given any two-counter turing machine, we construct an SREL formula such that the formula is satisfiable iff the two-counter turing machine reaches the final state.

Consider a two-counter turing machine $2CT$ with two counters C_a, C_b and $n + 1$ states, $s_0, \ldots, s_n$ where s_0 is the initial state, and s_n is the final state. Initially, C_a and C_b have values c and 0, respectively, for some positive integer c. We introduce an event e_i for each state s_i. Then $2CT$ is in exactly one state at any moment and can be represented by SREL formula $\phi_S = AG(\bigvee_i (e_i \wedge \bigwedge_{j \neq i} \neg e_j))$.

We use three events e_a, e_b, e_o to simulate the counters. More specifically, $\#e_x - \#e_o$ records the value of counter C_x for $x = a, b$. The initial state of $2CT$ is represented by SREL formula $\phi_I = e_0 \wedge \#e_a = \#e_o + c \wedge \#e_b = \#e_o$.

Now we consider the set of transitions in $2CT$. Let (s_i, s_j) be such a transition. Given a counter x, let $\bar{x}$ denote the other counter.

—*Case* 1: the transition increases the value of counter x by 1. Then the predicate representing the next state is $\phi_{(i,j,x,+1)} = e_j \wedge e_x \wedge \neg e_{\bar{x}} \wedge \neg e_o$.
—*Case* 2: the transition decreases the value of counter x by 1. Then the predicate representing the next state is $\phi_{(i,j,x,-1)} = e_j \wedge \neg e_x \wedge e_{\bar{x}} \wedge e_o$.
—*Case* 3: the transition goes to s_j if x is positive in s_i, and it goes to s_k otherwise. Then the predicate representing the next state is $\phi_{(i,j,k,x,>0)} = \neg e_a \wedge \neg e_b \wedge \neg e_o \wedge (\#e_x > \#e_o \wedge e_j \vee \#e_x \leq \#e_o \wedge e_k)$.

Therefore, the SREL formula for the transition relation in $2CT$ is

$$\phi_T = AG(\bigvee_i (e_i \wedge AF[1, 1]((\bigvee \phi_{(i,j,x,+1)}) \vee (\bigvee \phi_{(i,j,x,-1)}) \vee (\bigvee \phi_{(i,j,k,x,>0)})))),$$

and the SREL formula for the entire $2CT$ is

$$\phi = \phi_I \wedge \phi_S \wedge \phi_T \wedge EF(e_{n+1}).$$

Clearly, $2CT$ reaches the final state s_{n+1} iff the SREL formula ϕ is satisfiable. Since the satisfiability problem for $2CT$ is undecidable, so is it for SREL. □

Although the satisfiability problem is undecidable in general, most real-time systems obey certain *bounded progress conditions* as part of their specifications. More precisely, the occurrences of related events are usually balanced. For instance, whenever a stimulus event happens, its corresponding response event will happen within a bounded time period. Consequently, the difference between the number of occurrences of the stimulus event and that of the response event is usually bounded by a constant at any moment. Furthermore, if a property we want to verify involves a comparison between the numbers of occurrences of two events, then the two events are mostly likely related to each other in some way.

Hence, we define the following bounded progress condition for a real-time system.

Definition 4.3 (*Bounded Progress Condition*). A *bounded progress* (B.P. for short) condition for a real-time system is a conjunction of inequalities between counting terms:

$$\Phi = (\bigwedge_{i,j} (b_{i,j} \leq k_i \#E_i - k_j \#E_j \leq B_{i,j})),$$

where $b_{i,j} \leq B_{i,j}$ are two integer constants. The real-time system is *well behaved* with respect to the B.P. condition if it satisfies property $AG(\Phi)$. An SREL formula is *well confined* if, for any pair of counting terms $k_1 \#E_1$ and $k_2 \#E_2$ that is involved in a comparison directly or indirectly through a register in the formula, $b_{1,2} \leq k_1 \#E_1 - k_2 \#E_2 \leq B_{1,2}$ where bounds $b_{1,2}$, $B_{1,2}$ are derived from Φ.

In Example 3.2, the first property checks a B.P. condition for the missile system that involves stimuli events *Loc* and response events *Fire*. As we shall show in Section 6, the verification of a real-time system becomes tractable under the B.P. condition. In the rest of the article, we shall assume that the SREL formulas are all well confined by some B.P. conditions with respect to the models they will be checked against.

5. MODECHART

Modechart [Jahanian and Mok 1986] is a graphical language extended from STATECHART [Harel 1986] with the constructs for specifying time properties. Its visualized hierarchical structure together with a small set of well-defined constructs makes it very attractive for defining timing behaviors of real-time systems. For ease of representation, we shall use a restricted form of Modechart with only a three-level hierarchy. An example

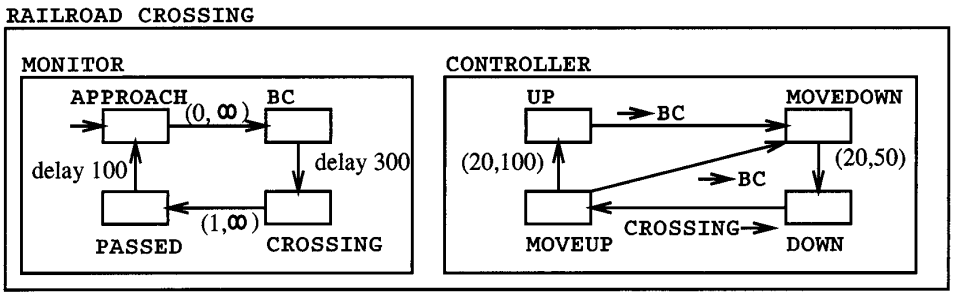

Fig. 2. A modechart for the railroad crossing system.

is shown in Figure 2, which formally specifies the railroad crossing control system in Section 2. The outermost box with name "railroad crossing" is the *parallel mode* representing the entire railroad crossing system. The second-level boxes with names "monitor" and "controller" are the *serial modes* representing the concurrent components in the system. The innermost boxes in each serial mode are the *atomic modes* representing the states of the serial mode. The arcs in each serial mode are *transitions*. A *triggering* condition, e.g., $\rightarrow BC$, on an arc means that once the condition holds when the serial mode is in the source atomic mode, the transition must be taken immediately. A *timing* condition of the form (r, d) means that the transition must be taken no earlier than r and no later than d once the source mode is entered.

The restricted form of Modechart is formally defined as follows.

Definition 5.1 (*Modechart*). A *modechart* G is a *parallel mode* consisting of a finite set of *serial modes* $\{P_k | 1 \leq k \leq n\}$. Each mode P_k is a structure $P_k = \langle A_k, a_{k_0}, E_k, \mu_k \rangle$, where

(1) A_k is a finite set of *atomic modes*.
(2) $a_{k_0} \in A_k$ is the *initial mode*.
(3) $E_k \subseteq A_k \times A_k$ is a set of *transition edges*.
(4) μ_k is a function labeling each edge with either
   (a) a *triggering condition*, i.e., a disjunctive normal form whose literals range over the *event set* of the modechart, or
   (b) a *timing condition*, i.e., a pair (r, d) where $r, d \in N$ and $r \leq d$ are called *delay* and *deadline*, respectively.

The *event* set E_G of G consists of an *entry event* $\rightarrow a$ and an *exit event* $a \rightarrow$ for each atomic mode a.

To define the formal semantics for the modechart in a computation model (Definition 2.2), we first introduce a timer τ_i for each serial model P_i. The timer is associated with the entry event for every atomic mode in the serial mode and is reset when such an event occurs. Let us denote the set of timers by Γ_G and the reset function by R_G.

Similar to time points, a *state* s of the modechart G is defined by three assignments $\mathbb{O}_s : E_G \rightarrow Boolean$, $\mathcal{J}_s : E_G \rightarrow N$, and $\mathcal{T}_s : \Gamma_G \rightarrow N$. We say a state is *legal* if it satisfies the following conditions:

(1) Every serial mode is in exactly one atomic mode, i.e., for any serial mode P_k, there is exactly one atomic mode a_k in A_k such that $\mathcal{J}_s(\rightarrow a_k) = \mathcal{J}_s(a_k \rightarrow) + 1$, and for any other atomic mode $a'_k \in A_k$, $\mathcal{J}_s(\rightarrow a'_k) = \mathcal{J}_s(a'_k \rightarrow)$.
(2) No deadline is missed, i.e., if a serial mode P_k is in atomic mode a_k, and the largest deadline on a transition from a_k is d_{max}, then, $\mathcal{T}_s(\tau_k) \leq d_{max}$.

We say a transition $\langle a_k, a'_k \rangle$ with a timing condition (r, d) is *enabled* in legal state s if P_k is in a_k and $r \leq \tau_k \leq d$. A transition $\langle a_k, a'_k \rangle$ with a triggering condition is *enabled* in s if P_k is in a_k and if the condition evaluates to be true in s.

Definition 5.2 (*Run Model*). The *run model* of modechart G is a pair (s_{init}, χ) where

—s_{init} is the *initial state* of G, i.e.,

(1) For every event e, if it is an entry event to an initial mode e, $\mathbb{O}s_{init}(e) = true$, $\mathcal{J}s_{init}(e) = 1$. Otherwise,

$$\mathbb{O}s_{init}(e) = false,\ \mathcal{J}s_{init}(e) = 0.$$

(2) For every timer τ_k, $\mathcal{T}s_{init}(\tau_k) = 0$.

—χ is the next-state relation from legal states to legal states defined as follows.

(1) A legal state s' is reached from a legal state s by taking an enabled mode transition $\langle a_k, a'_k \rangle$ in serial mode P_k if (1) $\mathcal{T}_{s'}(\tau_k) = 0$, and $\mathcal{T}_{s'}(\tau_l) = \mathcal{T}_s(\tau_l)$ for all $l \neq k$, (2) for every event e, if $e \in \{a_k \rightarrow, \rightarrow a'_k\}$, $\mathbb{O}_{s'}(e) = true$, $\mathcal{J}_{s'}(e) = \mathcal{J}_s(e) + 1$; and otherwise $\mathbb{O}_{s'}(e) = \mathbb{O}_s(e)$, $\mathcal{J}_{s'}(e) = \mathcal{J}_s(e)$.
(2) s' is reached from s by taking a time tick transition if (1) no transition with a triggering condition is enabled in s, (2) for all serial modes P_k, $\mathcal{T}_{s'}(\tau_k) = \mathcal{T}_s(\tau_k) + 1$, and (3) for every event e, $\mathbb{O}_{s'}(e) = false$, $\mathcal{J}_{s'}(e) = \mathcal{J}_s(e)$.

An interesting property about this definition is that the next-state relation *preserves monotonicity* for events at any time point, i.e., once an event becomes true, it will remain true until time passes. Therefore, once a triggering condition is true, it stays true unless time passes.

Now we build the computation model for the modechart. We say a legal state is a *time point* if it can reach another legal state by taking a time tick.

Definition 5.3 (*Computation Model for Modechart*). The *computation model* for G is $\Xi_G = (P_0, \mathcal{N})$ where

—P_0 is the set of *initial* time points, i.e., $p \in P_0$ iff p can be reached from s_{init} by taking mode transitions only;

—$\mathcal{N}$ is the next-point relation from time points to time points such that $\langle p, p' \rangle \in \mathcal{N}$ *iff* p' can be reached from p by taking a time tick transition followed by zero or more mode transitions.

We define a modechart to be *normal* if no event can happen more than once at any time point, i.e., for any $\langle p, p' \rangle \in \mathcal{N}$ and event e, $\mathcal{F}_{p'}(e) \leq \mathcal{F}_p(e) + 1$. From now on, we assume all the modecharts are normal.

6. MODEL CHECKING SREL AGAINST MODECHART

In this section, we present an algorithm for model checking SREL against Modechart. In the next section, we shall describe how to implement the algorithm symbolically.

In order to have a model-checking algorithm, we have to be able to represent the computation model finitely. Hence, we shall first build a finite quotient model for the computation model, given a modechart and an SREL formula to be verified. Then we shall apply our model-checking algorithm to the quotient computation model.

To obtain a finite representation for the computation model of a real-time system, we need to solve two problems. First, a timer may have infinitely many values. We adopt the region graph approach from the dense time domain [Alur et al. 1990]. For each timer, we use a fixed constant $\hat{D}$ to represent all the values beyond D, where D is the largest bound in any timing constraint concerning the timer.

Second, an event can happen infinitely often. To solve this problem, we notice that SREL formulas only compare the differences among event occurrences, whether they are in the same time point or in the different time points. Given an SREL formula, we introduce a *difference* $\Delta_{1,2}$ for every pair of terms $k_1\#E_1$, $k_2\#E_2$ that are compared either directly through a comparison or indirectly through a snapshot operator and a comparison. We replace $k_1\#E_1 - k_2\#E_2$ by $\Delta_{1,2}$ in every comparison $k_1\#E_1 - k_2\#E_2 \sim c$. We replace every comparison involving a register of form $\{r := k_2\#E_2\}.(\ldots k_1\#E_1 - r \sim c \ldots)$ *by* $\{r_{1,2} := \Delta_{1,2}\}.(\ldots r_{1,2} \sim c \ldots)$ where $r_{1,2}$ is a *difference* register. Note that in the rewritten formula, $r_{1,2}$ no long holds a constant. Once it gets the initial value in the snapshot operator, it will increment by k_1 whenever an event in E_1 occurs.

Recall that in Section 3 we introduced a B.P. condition to put bounds on the term differences for the events in a real-time system. Such a condition enables us to only consider a finite number of values for each term difference or difference register. More specifically, let $b_{1,2}$ and $B_{1,2}$ be, respectively, the lower bound and the upper bound explicitly or implicitly encoded in the B.P. condition for difference $\Delta_{1,2}$.

—The only values for $\Delta_{1,2}$ that we care about are in $[b_{1,2} - k_1|E_1|, B_{1,2} + k_2|E_2|]$. We say a legal state s is *tolerable* with respect to ϕ if $b_{1,2} - k_1|E_1| \leq k_1\Sigma_{e\in E_1}\mathcal{F}_s(e) - k_2\Sigma_{e\in E_2}\mathcal{F}_s(e) \leq B_{1,2} + k_2|E_2|$ for every $\Delta_{1,2}$ in

ϕ. The reason we choose a looser bound than [$b_{1,2}$, $B_{1,2}$] is that we allow the difference to be temporarily out of the bound in intermediate states as long as it can make it back in the subsequent time point. Once the difference is out of the loose bound in any state, them the system is definitely not well behaved, and an error should be reported.

—Similarly, for each difference register $r_{1,2}$, its value should never be below $b_{1,2} - k_1|E_1|$. Furthermore, for all the values above $C_{1,2}$ where $C_{1,2}$ is the largest constant in all the comparisons involving $r_{1,2}$, we can represent them by a fixed constant $\hat{C}_{1,2}$, since $r_{1,2}$ monotonically increases, and its exact value is no longer interesting once it goes beyond $C_{1,2}$.

Now let us build a quotient computation model, given a modechart G with a B.P. condition and an SREL formula ϕ. To make the quotient computation model smaller, we only include those event variables that appear either in some triggering conditions or in ϕ. We also only include those differences that appear in the formula. Further, let us introduce a mode variable $mode_k$ for each serial mode P_k.

Our first step is to build a quotient run model. A *quotient state* is an assignment to all the mode and event variables, the timers for serial modes, and the differences in ϕ. Each quotient state represents an equivalence class for the tolerable states. In addition, we introduce a special quotient state $\sqcap$ to represent all intolerable states.

Definition 6.1. The equivalence class for a tolerable legal state s is the following quotient state $[s]$:

—For each serial mode P_k, if $\mathcal{J}_s(\rightarrow a_k) = \mathcal{J}_s(a_k \rightarrow) + 1$, then $mode_k = a_k$ is in $[s]$.

—For each timer τ, let D be the largest time bound for τ. If $\mathcal{T}_s(\tau) = c \leq D$, then $\tau = c$ is in $[s]$; otherwise $\tau = \hat{D}$ is in $[s]$.

—For each event e in some triggering condition or in ϕ, if $\mathcal{O}_s(e) = true$, then $e = true$ in $[s]$; otherwise $e = false$.

—For each difference $\Delta_{1,2}$, if $k_1\Sigma_{e\in E_1}\mathcal{J}_s(e) - k_2\Sigma_{e\in E_2}\mathcal{J}_s(e) = c$ where $b_{1,2} - k_1|E_1| \leq c \leq B_{1,2} + k_2|E_2|$, then $\Delta_{1,2} = c$ is in $[s]$.

From this definition, we can immediately conclude the following:

LEMMA 6.2

(1) *For any timer τ, a timing condition $[r, d]$ is satisfied in a tolerable state s iff it is satisfied in the quotient state $[s]$.*

(2) *For any comparison $\Delta \sim c$ in the formula, it is true in s iff it is true in $[s]$.*

(3) *An event e is true in s iff it is true in $[s]$.*

Furthermore, we also have the following:

LEMMA 6.3. *Let s_1, s_2 be tolerable legal states such that $[s_1] = [s_2]$. Then for every successor s'_1 of s_1, there is a successor s'_2 of s_2 such that $[s'_1] = [s'_2]$ and is reached by taking the same transition, and vice versa.*

PROOF. Since $[s_1] = [s_2]$, every serial mode is in the same atomic mode in both states. Based on Lemma 6.2, we can easily conclude that a mode transition is enabled in s_1 iff it is enabled in s_2. We shall prove the lemma by a case analysis.

Case 1. s'_1 is reached from s_1 by taking a mode transition. Then there is a state s'_2 that is reached from s_2 by the same mode transition. By Definition 5.2 and Definition 6.1 we have $[s'_1] = [s'_2]$.

Case 2. s'_1 is reached from s_1 by a time tick transition. Then since s_1 and s_2 have the same set of enable mode transitions, time can also advance from s_2. Let s'_2 be reached from s_2 by a time tick transition. By the same definitions as in Case 1, $[s'_1] = [s'_2]$. □

Applying the lemma to the computation model $\Xi_G = (s_{init}, \mathcal{N})$, we can immediately conclude the following:

COROLLARY 6.4. *Let p_1, p_2 be two tolerable time points in the computation model such that $[p_1] = [p_2]$. Then for every successor p'_1 of p_1, there* is a successor *p'_2 of p_2 such that $[p'_1] = [p'_2]$, and vice versa.*

Let $[\Xi]_G = ([P_0], [\mathcal{N}])$ be the quotient computation model as defined by the equivalence relation. We expand it to include an assignment Υ to all the difference registers in every quotient time point. Initially, a difference register $r_{1,2}$ has either an undefined value *undef* or the value in $\Delta_{1,2}$. For two quotient states $[s]$, $[s']$ such that $[s'] \in [\mathcal{N}]([s])$, we have a transition from $([s], \Upsilon)$ to $([s'], \Upsilon')$ if the following two conditions hold:

(1) If $\Upsilon(r_{1,2}) = undef$, then $\Upsilon'(r_{1,2}) = undef$ or $\Upsilon'(r_{1,2}) = \Delta_{1,2}$.
(2) If $\Upsilon(r_{1,2})$ has a defined value, then $\Upsilon'(r_{1,2}) = \Upsilon(r_{1,2}) + k_1 \cdot d$ where d is the number of variables in $E_1 \cap \{e : \mathcal{C}_{[s']}(e) = true\}$.

Let χ be the directed graph representing the expanded quotient computation model with each node representing an expanded quotient state and each edge representing the next-point relation between two quotient states. If some time point with some $\Delta_{1,2}$ being out of range $[b_{1,2}, B_{1,2}]$ is reachable from an initial time point, then we report the violation and terminate. Otherwise, our model-checking algorithm is the following inductive procedure to label each node with the set of SREL subformulas in ϕ that are true in the node.

—$\phi = e$: label all the nodes in which e is true.
—$\phi = \Delta_{1,2} \sim c$: label all the nodes in which $\Delta_{1,2} \sim c$.
—$\phi = r_{1,2} \sim c$: label all the nodes in which $r_{1,2} \sim c$.
—$\phi = \neg\phi_1$: label all the nodes that are not labeled with ϕ_1.
—$\phi = \phi_1 \vee \phi_2$: label all the nodes that are either labeled with ϕ_1 or ϕ_2.

—$\phi = \{r_{1,2} := \Delta_{1,2}\}.\ \phi_1$: for any node with $r_{1,2} = \Delta_{1,2}$ that is labeled with ϕ_1, label this node and all the nodes identical to it except that $r_{1,2}$ has a different value.
—$\phi = AF[l, u]\phi_1$: label every node satisfying that along every path from the node, there is a node whose distance from the node is within $[l, u]$ and is labeled with ϕ_1.
—$\phi = EF[l, u]\phi_1$: label every node satisfying that there exists a path of length in $[l, u]$ from the node whose end node is labeled with ϕ_1.

Before we end this section, we present the following main two theorems.

THEOREM 6.5. *For any environment $\mathscr{E}$ and a time point p, $p \models \phi$ iff ϕ is labeled in $([p], \Upsilon)$ where $\Upsilon(r_{1,2}) = k_1\Sigma_{e\in E_1}\mathcal{J}_p(e) - \mathscr{E}(r)$ for any difference register in ϕ.*

PROOF. We prove the theorem by induction.

—For cases where $\phi = e$ or $\phi = \Delta_{1,2} \sim c$, we can immediately conclude the theorem based on the definition of a quotient state.
—For the case where $\phi = r_{1,2} \sim c$, we can also immediately conclude the theorem by the way Υ is constructed.
—For the case where $\phi = \neg\phi_1$ and where the theorem holds for ϕ_1, we have $p \models_{\mathscr{E}} \phi_1$ *iff* ϕ *is not labeled in* $([p], \Upsilon)$. Therefore, $p \models_{\mathscr{E}} \neg\phi_1$ *iff* $\neg\phi$ is marked in $([p], \Upsilon)$.
—For the case where $\phi = \phi_1 \vee \phi_2$ and where the theorem holds for ϕ_1 and ϕ_2, the theorem can be derived from the semantic definition for $p \models_{\mathscr{E}} \phi_1 \vee \phi_2$ and the way to label $\phi_1 \vee \phi_2$ in our model-checking algorithm.
—For the case where $\phi = \{r_{1,2} := \Delta_{1,2}\}.\phi_1$ and where the theorem holds for ϕ_1, let us first assume $p \models_{\mathscr{E}} \phi$. Then by the semantic definition, $p \models_{\mathscr{E}[r:=\Delta_{1,2}]} \phi_1$. Hence, ϕ_1 is labeled in $([p], \Upsilon')$ where Υ' is identical to Υ except that $\Upsilon'(r_{1,2}) = \Delta_{1,2}$. By the labeling procedure, ϕ is also labeled in $([p], \Upsilon)$. Now let us assume ϕ is labeled in $([p], \Upsilon)$. Then ϕ_1 must be labeled in $([p], \Upsilon')$ according to the labeling procedure. This implies that $p \models_{\mathscr{E}[r_{1,2}:=\Delta_{1,2}]} \phi_1$ and consequently $p \models_{\mathscr{E}} \{r_{1,2} := \Delta_{1,2}\}.\phi_i$.

—For the case where $\phi = EF[l, u]\phi_1$ and where the theorem holds for ϕ_1, we first note that each transition in the computation represents a time tick. Let us first assume $p \models_{\mathscr{E}} \phi$. Then, there is a path $p_0 = p$, $p_1, \ldots p_k$ with $l \le k \le u$ such that $p_k \models_{\mathscr{E}} \phi_1$. Now consider the path $([p], \Upsilon), ([p_1], \Upsilon_1), \ldots, ([p_k], \Upsilon_k)$ in the expanded computation model. For any different register $r_{1,2}$, we have $\Upsilon_i(r_{1,2}) = k_1\Sigma_{e\in E_1}\mathcal{J}_{p_i}(e) - \mathscr{E}(r)$ for all $0 \le i \le k$ by the construction of Υ. Therefore, ϕ_1 is labeled in $([p_k], \Upsilon_k)$. By the marking procedure, ϕ is marked in $([p], \Upsilon)$. The other direction can be proved similarly.
—The case for $\phi = AF[l, u]\phi_1$ can be proved similarly. □

THEOREM 6.6. *The time complexity for checking a well-confined SREL formula against a well-behaved model* Ξ *is in* $|\phi| \cdot |[\Xi]|$ *where* $[\Xi]$ *is the quotient graph for* Ξ *and* $|[\Xi]| = (\Pi_{i,j}(B_{i,j} - b_{i,j} + k_1|E_1| + k_2|E_2|)) \cdot (\Pi_{i,j}(C_{i,j} + 1)) \cdot (\Pi_k(D_k + 2)) \cdot (\Pi_k|A_k|) \cdot (2^{|E|})$ *where* E *is the set of events in the quotient model.*

The complexity is based on the fact that our labeling algorithm is linear to the size of ϕ and linear to the size of the quotient graph.

7. SYMBOLIC MODEL CHECKING

As we proved in the previous section, the worst-case complexity makes the explicit graph-labeling algorithm impractical. Among many research efforts in recent years, the symbolic model-checking approach has emerged as a promising solution to effectively relieve the state explosion problem. In this section, we show how to apply the symbolic approach to the model-checking problem for SREL and Modechart. It is a straightforward three-step process based on the technique presented in the previous section.

The first step is to symbolically build a quotient run model, given an SREL formula ϕ and a well-behaved modechart G under a certain B.P. condition. This step involves building a predicate for the initial quotient state $[s_{init}]$ and a predicate for the next-state function $[\chi]$. Let E be the set of events that are either in the triggering conditions in G or in ϕ, $E_k \subseteq E$ the set of events in serial mode P_k, $E_{\langle a,a' \rangle}$ the set of events associated with transition $\langle a, a' \rangle$, i.e., $E_{\langle a,a' \rangle} = \{\rightarrow a', a \rightarrow\}$. Let o_e be the event occurrence variable for event e. For a variable x, let x' denote the copy of x in the next state.

The predicate for the initial quotient state is the following predicate:

$$s_{init} ::= \bigwedge_k (mode_k = a_{k_0} \wedge \tau_k = 0) \wedge$$

$$\bigwedge_{e \in E \cap \{\rightarrow a_{k_0}\}} o_e \wedge \bigwedge_{e \in E \setminus \{\rightarrow a_{k_0}\}} (\neg o_e) \wedge$$

$$init_d \wedge init_{dr}$$

where predicate $init_d$ and $init_{dr}$ describe the initial values in the differences and difference registers, respectively.

The next-state relation is constructed using the following rules, of which Rule 7.1 through Rule 7.3 capture all the mode transitions and in which Rule 7.4 captures a time tick transition. For ease of presentation, we shall simply write $x' = x + c$ to represent $(x + c < l \wedge x' = \check{l} \vee l \leq x + c \leq u \wedge x' = x + c \vee x + c > u \wedge x' = \hat{u})$ where l, u are, respectively, the lower and upper bounds for x.

Rule 7.1. *A mode transition* $\langle a_k, a'_k \rangle$ *in mode* P_k *with condition* ψ *is captured by the following two predicates:*

$$trans_{\langle a_k, a'_k \rangle} ::= (mode_k = a_k) \wedge \psi_1 \wedge (mode'_k = a'_k) \wedge (\tau'_k = 0) \wedge$$

$$(\wedge_{e \in E_k \cap E_{\langle a_k, a'_k \rangle}} o'_e) \wedge (\wedge_{e \in E_k \setminus E_{\langle a_k, a'_k \rangle}} o'_e = o_e) \wedge$$

$$(\wedge_{\Delta_{i,j}} (\Delta'_{i,j} = \Delta_{i,j} + k_i \cdot |E_{\langle a_k, a'_k \rangle} \cap E_i| - k_j \cdot |E_{\langle a_k, a'_k \rangle} \cap E_j|))$$

$$(\wedge_{r_{i,j}} (r'_{i,j} = r_{i,j} + k_i \cdot |E_{\langle a_k, a'_k \rangle} \cap E_i|))$$

and

$$legal_{\langle a_k, a'_k \rangle} ::= \neg(mode_k = a_k) \vee \neg\psi_2$$

where $\psi_1 = \tau_k \geq r$; $\psi_2 = \tau_k \geq d$ if $\psi = (r, d)$; $\psi_1 = \psi_2 = \psi$ otherwise.

Predicate $trans_{\langle a_k, a'_k \rangle}$ defines when and how a mode transition is taken. Predicate $legal_{\langle a_k, a'_k \rangle}$ forces the transition to be taken immediately if the triggering condition is satisfied or if P_k is about to miss its deadline.

Rule 7.2. The set of the mode transitions in a serial mode P_k is captured by the following predicate:

$$trans_k ::= \vee_{\langle a_k, a'_k \rangle \in P_k} trans_{\langle a_k, a'_k \rangle}$$

Rule 7.3. The set of the mode transitions in G is captured by the following predicate:

$$trans ::= \vee_k (trans_k \wedge (\wedge_{x \notin P_k} x' = x))$$

A time tick of a single timer τ_k is captured by the following predicate:

Rule 7.4.

$$tick_k ::= (\tau_k \leq b_{\tau_k}) \wedge (\tau'_k = \tau_k + 1) \vee (\tau_k > b_{\tau_k}) \wedge (\tau'_k = \tau_k)$$

where b_τ is the largest constant referred by τ_k.

Rule 7.5. The time tick transition is captured by the following predicate:

$$tick ::= (\wedge_{\langle a, a' \rangle} legal_{\langle a, a' \rangle}) \wedge (\wedge_k \tau'_k = \tau_k + 1)$$
$$\wedge (\wedge_{e \in E} \neg o'_e) \wedge (\wedge_{\Delta_{i,j}} \Delta'_{i,j} = \Delta_{i,j}) \wedge (\wedge_{r_{i,j}} r'_{i,j} = r_{i,j})$$

Combining Rule 7.3 and Rule 7.4, we finally get the next rule:

Rule 7.6. The next-state relation for G is the following predicate:

$$\chi ::= trans \vee tick$$

A complete symbolic representation for the railroad crossing system is given in the following paragraphs. Here we consider the simplest case, in which we are only interested in the two triggering events $\rightarrow BC$ and $CROSSING \rightarrow$, and we do not care about the differences between event occurrences.

The initial condition for the system is the following:

$$[s_{init}] ::= (P_{MON} = APPROACH) \wedge (\tau_{MON} = 0) \wedge (P_{CON} = UP) \wedge (\tau_{CON} = 0) \wedge (\rightarrow BC = false) \wedge (CROSSING \rightarrow = false).$$

The mode transition relation for the monitor is the following:

$$\begin{aligned} trans_{MON} ::= {} & (P_{MON} = APPROACH) \wedge (\tau_{MON} \geq 0) \wedge (P'_{MON} = BC) \wedge (\tau'_{MON} = 0) \wedge (\rightarrow BC' = true) \wedge (CROSSING \rightarrow' = CROSSING \rightarrow) \vee \\ & (P_{MON} = BC) \wedge (\tau_{MON} \geq 300) \wedge (P'_{MON} = CROSSING) \wedge (\tau'_{MON} = 0) \wedge (\rightarrow BC' = \rightarrow BC) \wedge (CROSSING \rightarrow' = CROSSING \rightarrow) \vee \\ & (P_{MON} = CROSSING) \wedge (\tau_{MON} \geq 1) \wedge (P'_{MON} = PASSED) \wedge (\tau'_{MON} = 0) \wedge (\rightarrow BC' = \rightarrow BC) \wedge (CROSSING \rightarrow' = true) \vee \\ & (P_{MON} = PASSED) \wedge (\tau_{MON} \geq 100) \wedge (P'_{MON} = APPROACH) \wedge (\tau'_{MON} = 0) \wedge (\rightarrow BC' = \rightarrow BC) \wedge (CROSSING \rightarrow' = CROSSING \rightarrow). \end{aligned}$$

The mode transition relation for the gate controller is the following:

$$\begin{aligned} trans_{CON} ::= {} & (P_{CON} = UP) \wedge (\rightarrow BC = true) \wedge (P'_{CON} = MOVEDOWN) \wedge (\tau'_{CON} = 0) \vee \\ & (P_{CON} = MOVEDOWN) \wedge (\tau_{CON} \geq 20) \wedge (P'_{CON} = DOWN) \wedge (\tau'_{CON} = 0) \vee \\ & (P_{CON} = DOWN) \wedge (CROSSING \rightarrow = true) \wedge \end{aligned}$$

$$(P'_{CON} = MOVEUP) \wedge (\tau'_{CON} = 0) \vee$$

$$(P_{CON} = MOVEUP) \wedge (\tau_{CON} \geq 20) \wedge (P'_{CON} = UP) \wedge$$

$$(\tau'_{CON} = 0) \vee$$

$$(P_{CON} = MOVEUP) \wedge (\rightarrow BC = true) \wedge$$

$$(P'_{CON} = UP) \wedge (\tau'_{CON} = 0).$$

The mode transition relation for the entire system is the following:

$$trans ::= trans_{MON} \wedge (P'_{CON} = P_{CON}) \wedge (\tau'_{CON} = \tau_{CON}) \vee$$

$$trans_{CON} \wedge (P'_{MON} = P_{MON}) \wedge (\tau'_{MON} = \tau_{MON}) \wedge$$

$$(\rightarrow BC' = \rightarrow BC) \wedge (CROSSING \rightarrow' = CROSSING \rightarrow).$$

The time tick transition is captured by the following predicate:

$$tick ::= (P_{CON} = movedown \Rightarrow \tau_{CON} \leq 50) \wedge$$

$$(P_{CON} = moveup \Rightarrow \tau_{CON} \leq 100) \wedge$$

$$(P_{CON} = up \Rightarrow \rightarrow BC = false) \wedge$$

$$(P_{CON} = down \Rightarrow CROSSING \rightarrow = false) \wedge$$

$$(P_{CON} = moveup \Rightarrow \rightarrow BC = false) \wedge$$

$$(\tau_{MON} > 300 \wedge \tau'_{MON} = \tau_{MON} \vee \tau_{MON} \leq 300 \wedge \tau'_{MON} = \tau_{MON} + 1) \wedge$$

$$(\tau_{CON} > 100 \wedge \tau'_{CON} = \tau_{CON} \vee \tau_{CON} \leq 100 \wedge \tau'_{CON} = \tau_{CON} + 1) \wedge$$

$$(\rightarrow BC' = false) \wedge (CROSSING \rightarrow' = false) \wedge$$

$$(P'_{MON} = P_{MON}) \wedge (P'_{CON} = P_{CON}).$$

Therefore, the next-state relation for the system is the following:

$$\chi ::= trans \vee tick.$$

The second step in our symbolic model-checking approach is to compute the quotient computation model from the quotient run model. Let X be the set of all variables in the model. Let X', X'' be the two copies of X in different states. We recall that a transition in the computation model is a time tick in the run model followed by a sequence of mode transitions until a time

Procedure: symbolic model checking

$e(X) := o_e$;
$(\Delta_{i,j} \sim c)(X) := \Delta_{i,j} \sim c$;
$(r_{1,2} \sim c)(X) := r_{1,2} \sim c$;
$(\neg\phi_1)(X) := \neg(\phi_1(X))$;
$(\phi_1 \vee \phi_2)(X) := \phi_1(X) \vee \phi_2(X)$;
$(\{r_{i,j} := \Delta_{i,j}\}.\phi)(X) := (r_{i,j} = \Delta_{i,j})(X) \Rightarrow \phi(X)$;
$(EF[l,u]\phi)(X) := compute_E F(EF[l,u]\phi(X))$;
$(AF[l,u]\phi)(X) := compute_A F(AF[l,u]\phi(X))$;
end.

Fig. 3. Symbolic model-checking procedure.

Procedure: compute EF

Case 1: $l = 0$ and $u = 0$: return $\phi(X)$;
Case 2: $l = 0$ and $u \neq \infty$: return $\phi(X) \vee (\exists X'(\mathcal{N}(X, X') \wedge EF[l, u-1]\phi(X')))$;
Case 3: $l = 0$ and $u = \infty$: return $\mu EF(X).\phi(X) \vee \exists X'(\mathcal{N}(X, X') \wedge EF(X'))$;
Case 4: $l > 0$: return $\exists X'(\mathcal{N}(X, X') \wedge EF[l-1, u-1]\phi(X'))$;
end.

Fig. 4. Compute $EF[l, u]$.

point is reached. Hence, we first compute the transitive closure for *trans* as the following fixed point:

$$trans^*(X, X') ::= \mu trans^*(X, X').((\wedge_{x \in X} x' = x) \vee (\exists X''(trans^*(X, X'') \wedge trans(X'', X))))$$

The condition for a quotient state to be a time point is the following predicate:

$$legal ::= \wedge_{\langle a, a' \rangle} legal_{\langle a, a' \rangle}.$$

Therefore, the symbolic quotient computation model is captured by the following two predicates:

$$P_0(X) ::= \exists X' s_{init}(X') \wedge trans^*(X', X)$$

and

$$\mathcal{N}(X, X') ::= legal(X) \wedge (\exists X''.tick(X, X'') \wedge trans^*(X'', X')) \wedge legal(X').$$

Finally, we apply the following symbolic model-checking procedure to the symbolic quotient computation model. The procedure is based on the one in Emerson et al. [1989]. The procedure recursively constructs a predicate $\phi(X)$ to denote the set of all time points in the model in which ϕ holds. To verify whether the model satisfies ϕ, we check whether $P_0(X) \Rightarrow \phi(X)$ is a tautology. The model-checking procedure is presented in Figure 3. The recursive function $computeEF(EF[l, u]\phi(X))$ is in Figure 4.

The recursive function *computeAF*($AF[l, u]\phi(X)$) is identical to function *computeEF* except that all existential quantifiers X' are replaced by universal quianfiers X' and conjunctions "$\wedge$" are replaced by implications "$\rightarrow$".

Before we conclude this section, we would like to emphasize that in the actual implementation it is not necessary to compute the next-point relation explicitly. When we want to find the preimage of a predicate $\phi(X)$, for example, we can simply compute, by a fixed-point computation, the set of symbolic states reachable to the states in $\phi(X)$ through zero or more mode transitions following a time tick.

8. A FAST ALGORITHM FOR BUILDING OBDDS FOR LINEAR CONSTRAINTS

The implementation of our symbolic model-checking approach is based on the OBDD-based symbolic verifier SMV from CMU. Ordered binary decision diagrams (OBDDs) are a canonical form representation for boolean formulas [Bryant 1986]. They are often substantially more compact than tranditional normal forms and can be manipulated very efficiently. An OBDD is similar to a binary decision tree except that it is a directed acyclic graph (DAG) obtained by merging identical subgraphs into a single graph, and it imposes a strict total order on the occurrence of variables as one traverses the graph from root to leaf. The SMV model checker is based on a description language for finite-state machines which can be annotated by specifications expressed in CTL. The model checker accepts a finite-state machine in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies the CTL specifications.

Our implementation can be viewed as an extension of SMV to a timed domain. Although SMV already has a full set of arithmetic operations for subrange integers, the implementation is extremely inefficient.

Since our approach involves many linear constraints of form $x + c \sim y + d$ over subrange integers, it is important to build the OBDDs for such constraints efficiently both in time and in space.

The integer type in SMV is treated with almost no difference from an ordinary enumeration type. For an enumeration type variable, the definition consists of a bit vector to represent the variable together with a complete ordered binary tree to map integer values to their binary representations. Whenever a subrange integer variable is declared, this definition scheme causes an exponential blow-up both in time and in space. Figure 5 shows such an example.

The same inefficiency lies in building OBDDs for constraints among subrange integer variables. Given such a constraint, SMV exhaustively enumerates all possible combinations of values for variables involved in the constraint and determines the set of combinations that makes the constraint true. As a result, the intermediate results during the building could have forbiddingly large sizes, and the building process is extremely time consuming. Figure 6 shows the construction of an expression $x' = (x + 1) \bmod 8$.

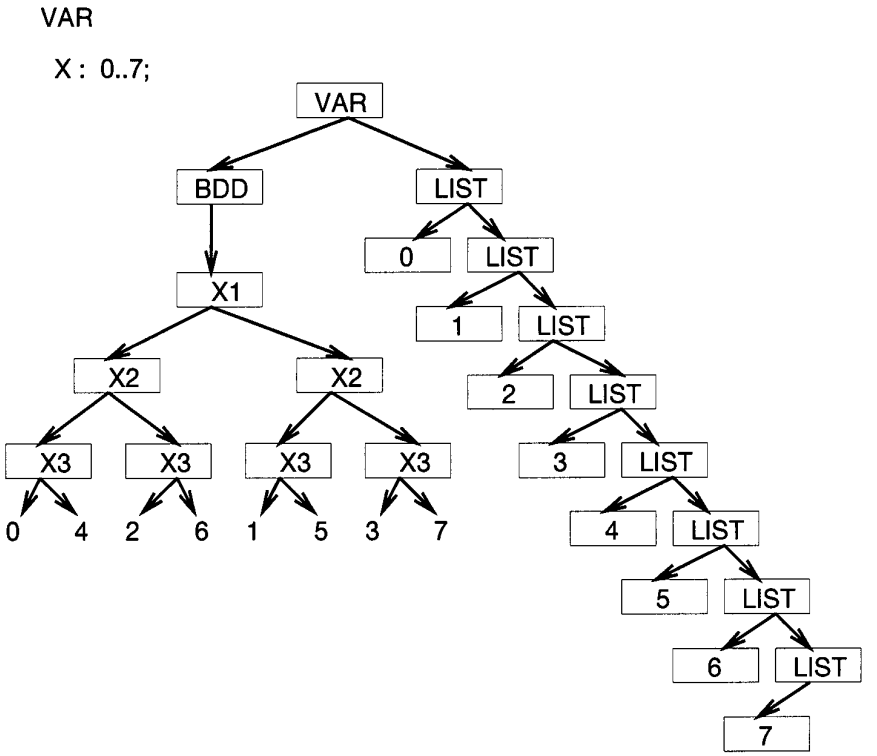

Fig. 5. A variable declaration.

We observe that for our purposes the constraints are only a subset of linear constraints. At most two variables are involved in any constraint. Hence, it is possible to devise a very efficient algorithm for building OBDDs for such linear constraints. For ease of presentation, we assume that all variables only have nonnegative integer values.

In our scheme, we do not need to build the explicit mapping from integer values to their binary representations because we insist on the standard binary encoding for integer values. Thus, declaring a variable takes no time.

In the following, we shall present the OBDD construction algorithms for simple constraints of form $a + b \sim c$ where a, b, c can be either variables or constants. We shall then discuss how to build OBDDs for complex linear constraints that involve more variables and coefficients. At the end, we shall show some experimental results.

Let $x = x_1x_2 \ldots x_n$, $y = y_1y_2 \ldots y_n$, $z = z_1z_2 \ldots z_n$ be three n-bit integer variables. Let $c = c_1c_2 \ldots c_n$ be an n-bit constant. Given a variableor a constant v, let $v[k, n]$ denote the last $n - k + 1$ bits of v. For thethree variables, we use interleaving variable ordering among them with the highest bit first, and the ordering z_k, x_k, y_k at each bit for all k.

First, let us consider constraint $x = y + z$. Let $eq(i, k)$ denote the predicate for $x[k, n] = y[k, n] + z[k, n]$, and the carry-bit of $y[k, n] +$

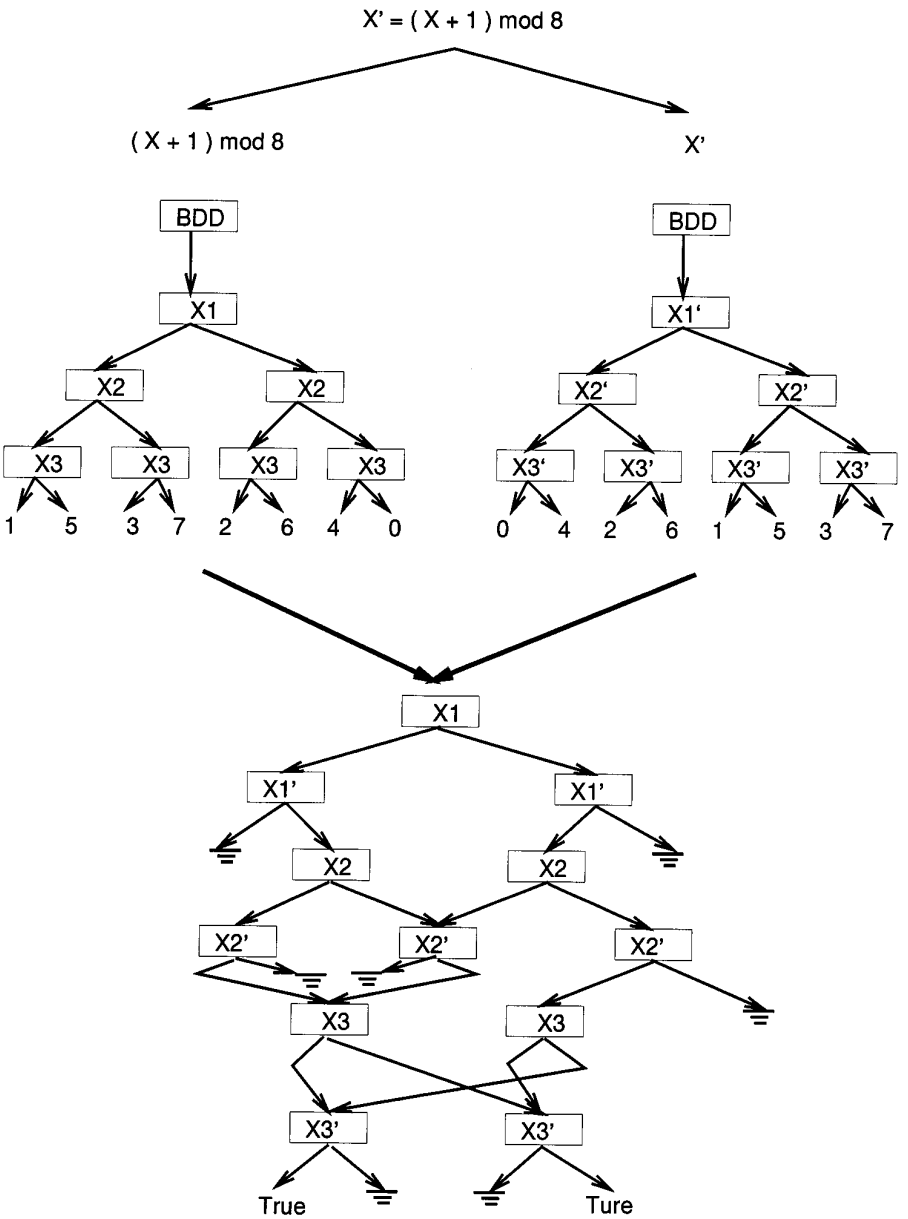

Fig. 6. The construction process of a constraint.

$z[k, n]$ is i where $i = 0, 1$. Let $eq(0, n + 1) = true$, $eq(1, n + 1) = false$.Then we have, for $k \leq n$,

$$eq(0, k) = (\overline{z_k} \wedge x_k \equiv y_k \vee z_k \wedge x_k \wedge y_k) \wedge eq(0, k + 1) \vee$$

$$(\overline{z_k} \wedge x_k \wedge \overline{y_k}) \wedge eq(1, k + 1),$$

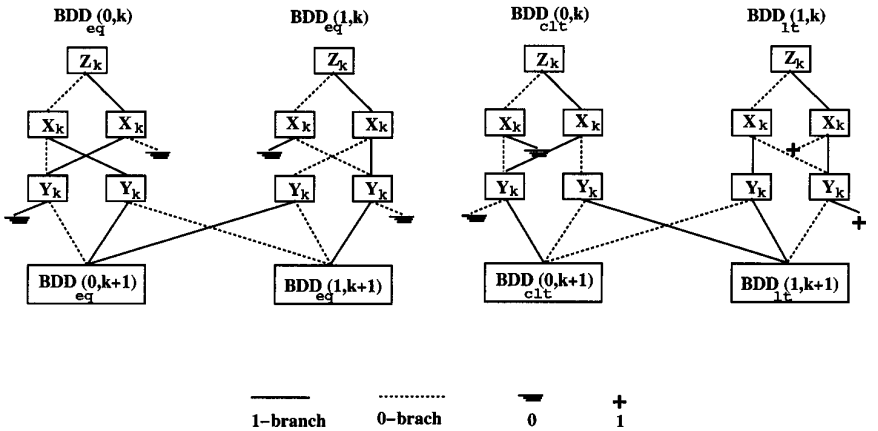

Fig. 7. Illustration of the BDD construction scheme.

$$eq(1, k) = (z_k \wedge \overline{x_k} \wedge y_k) \wedge eq(0, k + 1) \vee$$

$$(\overline{z_k} \wedge \overline{x_k} \wedge y_k \vee z_k \wedge x_k \equiv y_k) \wedge eq(1, k + 1).$$

Figure 7 shows how to build the BDD for $eq(i, k)$ from the BDD for $eq(i, k + 1)$. In the figure, $BDD_{eq}(i, k)$ denotes the BDD for $eq(i, k)$.

LEMMA 8.1. *The BDD for $x = y + z$ has a size $\leq 10n$ and can be built in time $O(n)$.*

PROOF. There is at most 10 nodes at each bit. Since the final BDD for $x = y + z$ is $BDD_{eq}(0, 1)$, and there are n bits, the size of the final BDD is $\leq 10n$. Furthermore, the time for building such a complete BDD is $O(n)$. □

COROLLARY 8.2. *There is a BDD of size $\leq 5n$ for $x = y + c$ which can be built in time $O(n)$.*

PROOF. If we replace z by c in $x = y + z$ in Figure 7, then only one branch from node z_k is needed. Therefore, at most five nodes are needed for each bit. Therefore the corollary holds. □

Now let us consider a constraint of the form $x < y + z$. Let $clt(k)$ denote the predicate for $x[k, n] < (y[k, n] + z[k, n]) \ mod \ 2^{n-k+1}$, and the carry-bit of $y[k, n] + z[k, n]$ is 1. Let $lt(k)$ denote $x[k, n] < y[k, n] + z[k, n]$. Let $clt(n + 1) = lt(n + 1) = false$. Then we have, for $k \leq n$,

$$clt(k) = (\overline{z_k} \wedge \overline{x_k} \wedge y_k \vee z_k \wedge x_k \equiv y_k) \wedge clt(k + 1) \vee$$

$$(z_k \wedge \overline{x_k} \wedge y_k) \wedge lt(k + 1),$$

Table I. The Test Result for $x = (x + 1) \bmod 2^i$

	Original Verifier			Modified Verifier		
i	Time (sec.)	Transition	Total	Time (sec.)	Transition	Total
12	689	59 + 1	12393	0.27	59 + 1	262
13	2838	66 + 1	24710	0.18	64 + 1	291
14	>10284	?	?	0.22	69 + 1	321
15	?	?	out of memory	0.18	74 + 1	352
16	?	?	out of memory	0.23	79 + 1	384

Table II. The Test Result for the Railroad Crossing Example

	Original Verifier			Modified Verifier		
max	Time (sec.)	Transition	Total	Time (sec.)	Transition	Total
300	6.93	426 + 1	3098	0.42	337 + 1	1996
3000	661	582 + 1	13280	0.43	439 + 1	2660

$$lt(k) = (\overline{z_k} \wedge \overline{x_k} \wedge y_k \vee z_k \wedge \overline{x_k} \vee z_k \wedge x_k \wedge y_k) \vee$$
$$(\overline{z_k} \wedge x_k \wedge \overline{y_k}) \wedge clt(k+1) \vee$$
$$(\overline{z_k} \wedge x_k \equiv y_k \vee z_k \wedge x_k \wedge \overline{y_k}) \wedge lt(k+1).$$

The right part of Figure 7 shows how to build the BDDs for $clt(k)$ and $lt(k)$ from the BDDs for $clt(k + 1)$ and $lt(k + 1)$. In the figure, $BDD_{clt}(0, k)$ is the BDD for $clt(k)$, and $BDD_{lt}(1, k)$ is the BDD for $lt(k)$.

LEMMA 8.3. *The BDD for $z < y + z$ has a size $\leq 10n$ and can be built in time $O(n)$.*

PROOF. At most 10 nodes are introduced for each bit. Since the BDD for $x = y + c$ is $BDD_{lt}(1)$, and there are totally n bits, the lemma holds. □

COROLLARY 8.4. *There is a BDD of size $\leq 5n$ for $x < y + c$ which can be built in time $O(n)$.*

For other types of comparison operators, we can apply very similar ideas and prove the same results. Therefore, we omit the descriptions for these algorithms.

It is not difficult to see from Figure 7 that if we swap the order among x_k, y_k, and z_k for some k, the size of the graph at each of these bits is still a constant, and therefore we can still obtain BDDs with sizes linear to the number of bits in these variables.

Now, let us consider how to deal with a linear constraint with many variables and constant coefficients. We can first unfold the constraint so that it does not include coefficients. Then we break such an unfolded constraint into a conjunction of constraints each of which contains at most three variables by introducing dummy variables. After we obtain the BDD

for each small constraint, we can repeatedly and two BDDs that share a common dummy variable and existentially quantify out the dummy variable. The final result will be the BDD for the original constraint.

We have incorporated the OBDD construction algorithm for linear constraints into SMV and tested our implementation on two examples. The first example is the construction of expression $x = (x + 1) \ mod \ 2^i$. The second example is the construction of the transition relation for the railroad crossing example described in a previous section. These two examples are tested on a SparcStation 1 with 20MHz clock and 16MB memory. The results are shown in Tables I and II.

A value listed in column *transition* is the number of BDD nodes representing a transition relation plus the number of BDD nodes representing an invariant. In our testing examples, all invariants are simply true, and therefore the corresponding BDDs are of size 1. A value listed in column *total* is the total number of BDD nodes used in a BDD construction, including those consumed by variable definitions. A question mark means that the value is not available. For these two test cases, we observe that the modified verifier achieves one to two orders of magnitude savings in time and significant savings in space.

9. CONCLUSIONS

We proposed in this article a logic called SREL for the verification of synchronous event-driven real-time systems and described a symbolic model-checking procedure for it. We showed a fast BDD construction scheme for linear constraints that demonstrate order-of-magnitude improvement over SMV. Such a BDD construction scheme has its own general interest in other linear constraint systems as well. We are continuing work to gain more efficiency by exploiting the semantics of synchronous systems where transient states are not observable.

Much theoretical and practical work needs to be done. Currently, we are refining our symbolic model checker for SREL based on SMV.

REFERENCES

ALUR, R., COURCOUBETIS, C., AND DILL, D. 1990. Model-checking for real-time systems. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science.* IEEE, New York.

ALUR, R., HENZINGER, T. A., AND HO, P. 1993. Automatic symbolic verification of embedded systems. In the *14th Annual Real-Time System Symposium.* IEEE, New York, 2–11.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35,* 8 (Aug.).

BURCH, J. H., CLARKE, E. M., DILL, D., HWANG, L., AND MCMILLAN, K. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science.* IEEE, New York.

CAMPOS, S., CLARKE, E., MARRERO, W., MINEA, M., AND HIRAISHI, H. 1994. Computing quantitative characteristics of finite-state real-time systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium.* IEEE, New York.

CLARKE, E. M. AND EMERSON, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logic of Programs.* Lecture Notes in Computer Science, vol. 131. Springer-Verlag, Berlin.

CLARKE, E. M., BURCH, J., GRUMBERG, O., LONG, D., AND MCMILLAN, K. 1991. Automatic verification of sequential circuit designs. In *Conference Proceedings of the Royal Society of London.*

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. Program. Lang. Syst. 8,* 2.

CLARKE, E. M., KHAIRA, M., AND ZHAO, X. 1995. Hybrid decision diagrams. In *ICCAD 95.* IEEE Computer Society Press, Los Alamitos, Calif.

COUDERT, O., BERTHET, C., AND MADRE, J. 1989. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the 1st Workshop on Computer-Aided Verification.* Lecture Notes in Computer Science, vol. 407. Springer-Verlag, Berlin.

EMERSON, E. A. AND CLARKE, E. M. 1980. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the 7th ICALP.* Lecture Notes in Computer Science, vol. 85. Springer-Verlag, Berlin.

EMERSON, E. A., MOK, A. K., SISTLA, A. P., AND SRINIVASAN, J. 1989. Quantitative temporal reasoning. In the *1st Annual Workshop on Computer-Aided Verification.*

HAREL, D. 1986. Statecharts: A visual formalism for complex systems. Tech. Rep., The Weizmann Institute of Science, Rehovot, Israel.

HENZINGER, T. A., NICOLLIN, X., SIFAKIS, J., AND YOVINE, S. 1991. Symbolic model checking for real-time systems. In *Proceedings of the 7th Symposium on Logics in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif.

JAHANIAN, F. AND MOK, A. K. 1986. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng. SE-12.*

JAHANIAN, F. AND STUART, D. 1988. A method for verifying properties of Modechart specifications. In *Proceedings of the 9th Real-Time Systems Symposium.* IEEE, New York.

LEVESON, N. G. AND STOLZY, J. L. 1985. Analyzing safety and fault tolerance using time Petri nets. In *TAPSOFT: Joint Conference on Theory and Practice of Software Development.* Springer-Verlag, Berlin.

LEWIS, H. R. 1979. *Unsolvable Classes of Quantificational Formulas.* Addison-Wesley, Reading, Mass.

MCMILLAN, K. 1992. The SMV system: Draft. Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. Available as ftp://emc.cs.cmu.edu/pub/smvmanual.r2.2.ps.Z.

MOK, A. K. 1985. SARTOR—A design environment for real-time systems. In *Proceedings of the 9th IEEE COMPSAC.* IEEE, New York.

SISTLA, J. 1982. A unified approach for studying the properties of transition systems. *Theoret. Comput. Sci. 18.*

STUART, D. 1990. Implementing a verifier for real-time systems. In *Proceedings of the 11th IEEE Real-Time Systems Symposium.* IEEE, New York.

WANG, F. AND MOK, A. K. 1992. Asynchronous real-time event logic. In *Proceedings of the 1992 International Computer Symposium* (Taiwan, R.O.C.).

YANG, J., MOK, A. K., AND WANG, F. 1993. Symbolic model checking for event-driven real-time systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium.* IEEE, New York.

Received May 1995; revised July 1996; accepted October 1996