Journal of Embedded Computing 1 (2005) 39–56
IOS Press

Symbolic simulation of industrial real-time and embedded systems -experiments with the bluetooth baseband communication protocol

Farn Wang[a,*], Geng-Dian Huang[b] and Fang Yu[b]
[a]*Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan 106, Republic of China*
[b]*Institute of Information Science, Academia Sinica, Taipei, Taiwan 115, Republic of China*

Abstract. We introduce the symbolic simulation function implemented in our model-checker/simulator RED 4.0 for dense-time concurrent systems. By representing and manipulating state-spaces as logic predicates, the technique of symbolic simulation can lead to high performance by encoding even a dense amount of traces in traditional simulation into one symbolic trace. Symbolic simulation adds the dimension of *width* to a trace of state-spaces. By controlling the width of traces, we have a much better chance to find bugs using fewer traces.
Our main contribution is the design of symbolic simulation function in RED 4.0 for dense-time concurrent systems. In our tool, users can strongly control the width of traces and the generation of traces. We discuss how to generate traces using various policies, how to manipulate the state-predicate, and how to manage the trace trees. Moreover, we design a C-like language whose programs can be mechanically translated into the optimized communicating timed automata(CTA). Engineers can also put down comment-line assertions as specifications in their verification tasks. Finally, experiments using our simulator to verify the Bluetooth baseband protocol justify the usefulness of our tool. The tool is available at http://cc.ee.ntu.edu.tw/~farn/.

Keywords: Assertions, specification, state-based, event-driven, model-checking, verification

1. Introduction

Traditional simulation [10,16,21] uses memory to record the variable values in a state along a trace and makes it possible for engineers to visualize the behaviors of the system design even before the hardware prototypes are put into reality. For many decades, simulation has been the major tool for engineers to successfully guarantee the quality of system designs in early cycles of system development. But for new system designs, such as System-on-a-Chip (SOC) which has tens of millions of gates, there will not be enough time and manpower to run enough simulation traces of the system designs. The verification complexity incurred by system designs in the next few years will simply overwhelm the capability of traditional simulation technology.

Model-checking technology [2,14] has promised to mathematically prove the correctness of system design. The development of symbolic model-checking technology [7,13] has brought the complete verification of many non-real-time industrial projects into reality. The symbolic manipulation techniques do not record the exact values of variables explicitly. Instead, sets of states are succinctly represented and manipulated as logic constraints on variable values. Such succinctness not only saves the memory space in representation but also allows us to construct the representation of a huge (or even dense) set of states in a few symbolic manipulation steps.

However, even with such powerful techniques of symbolic manipulation, the verification task of real-time concurrent systems still demands tremendous resources beyond the reach of current technology. The

*Corresponding author. E-mail: farn@cc.ee.ntu.edu.tw.

ISSN 1740-4460/05/$17.00

reachable state-space representations in TCTL model-checking [2] tasks usually demand complexity exponential to the input system description sizes. Usually, verification tasks blow up the memory usage before finishing with answers.

In a sense, traditional simulation and model-checking represent two extremes in the spectrum. Traditional simulation is efficient because you only have to record the current state, but the number of traces to cover full functionality of a system is usually forbiddingly high. On the other hand, model-checking can achieve functional completeness in verification but usually requires a huge amount of computing resources. Thus it will be profitable to develop a technique that balances these two extremes.

Symbolic simulation [34] was originally introduced and proven valuable for the verification of integrated circuits. While traditional simulation runs along a trace of precise state recordings, symbolic simulation runs along a trace of symbolic constraints, representing (convex or concave) spaces of current states. Metaphorically, traditional simulation is a probe while the new symbolic simulation technique is a searchlight that can monitor multiple state-traces at the same time. With the proper searchlight caliber, we have much better chance of discovering the imminent risks and potential threats in an immense sky.

We have implemented a symbolic simulator for dense-time concurrent systems using GUI (Graphical User-Interface) and adequate facilities to generate and manage the traces. The simulator is now part of RED 4.0 (`http://cc.ee.ntu.edu.tw/~farn/`), a model-checker/simulator for real-time systems. The tool also has advanced capability of neumerical coverage estimation and is useful in evaluating the progress of verification tasks. More details can be found in [46]. In developing the symbolic simulator, we encountered the following challenges and opportunities.

1.1. The model adopted for real-time concurrent systems

In simulation, we construct a mathematical model for a system design (and the environment) using computer programs and observe how the model behaves in the virtual world. The semantics of the model effects how efficiently we can model the system/environment interaction and how efficiently we can compute the traces.

Symbolic simulation has achived much success in the verification of VLSI circuits, which are usually synchronous. We plan to extend the ability to the area of real-time concurrent systems, like communication protocols, embedded software, etc. For such systems, there exists no global clock so the synchronous discrete-time model can lead to imprecise simulation. In a real-world real-time concurrent system, each hardware module may have its own clock. The new SOC can have multi-clocks in the same chip. Based on all these considerations, we adopt the well-accepted timed automata [3] as our system model that have multiple dense-time clocks.

The input language of RED 4.0 allows the description of a timed automaton as a set of process automata that communicate with each other through synchronizers (called, input/output events through channels in [24]) and global variables. Users may use binary synchronizers to construct *legitimate global transitions* (to be explained in Section 3) from *process transitions*. RED also allows users to control the *searchlight caliber* to better monitor a user-given goal (or risk) condition along traces.

1.2. Constructing and managing traces

Traces can be constructed randomly or according to a policy. *Random traces* are computed with random number generators so as to avoid the bias of designers and verification engineers. Many people do not feel confident with a design until it has been verified with random traces. *Directed traces* are constructed with built-in or user-given policies. They can help guide simulators to program lines that are suspected of having bugs or whose functions need to be closely monitored. With directed traces, our simulator can more efficiently construct the traces that are of interest to verification engineers.

Symbolic simulation actually adds one more dimension to the issue of random vs. directed traces. Since we use complex logic constraints to represent state-spaces, we actually build traces of state-spaces, instead of a single precise state. That is, this method constructs many traces (even a dense amount of them) simultaneously. Symbolic simulation thus adds the dimension of *width* to a trace of state-spaces. In Section 5, we shall discuss how to control the width of traces using the options provided by our simulator.

1.3. Blending our symbolic simulator into industrial development cycles

Since engineers are trained to write programs in traditional programming languages, such as C, C++ and

Verilog, it is important that system designs are described in a format similar to these programming languages. Thus, we define a new language, called *Timed C* (*TC*) that has C-like syntax and OVL (Open Verification Library) assertions [9,29], that serves as an intermediate language from C programs to formal descriptions. TC is designed to bridge the gap between the engineering world and the verification research community. It supports most of the programming constructs in traditional C, such as sequences, while-loops, and switch-statements. It also provides syntax constructs to abstract unimportant details for mechanical translation to *Communicating Timed Automata* (*CTA*) [33,37–39]. Moreover, we have added new constructs to make it easy to describe event-driven behaviors like timeouts. TC is designed for efficient mechanical translation from C-programs into formal models of embedded systems.

OVL is a new initiative in the VLSI industry whose purpose is to unify commercial EDA (Electronic Design Automation) tools. It provides a set of predefined specification modules instantiated as assertion monitors. It is supported by EDA companies and was donated to Accellera (an electronic industry standards organization) in anticipation of making OVL an industry standard. With OVL, engineers can write assertions as comment lines in their HDL (Hardware Description Language [6,32]) programs. OVL was originally designed for the assertions of VLSI circuits, which are highly synchronous discrete-time systems. In our work, we extend OVL assertions to a dense-time model and hence our symbolic simulator provides users inserting assertions as comment lines within TC programs.

We have also implemented an efficient optimized compiler to generate the corresponding CTA and TCTL formula of TC programs that have OVL assertions [47]. Such an extension will allow embedded system engineers to take advantage of verification technology with minimum effort in their development cycles.

1.4. Organizations of this paper

In the following sections, we first review some related work (Section 2), describe our system models (Section 3), and give a brief overview of what we have achieved in our implementations (Section 4). Then we delve into more details of our achievements (Sections 5, 6). Before experimenting with the industrial project, we illustrate our input language TC with a small example(Section 7). We report our experiments with our implementations and the Bluetooth baseband protocol (Section 8). We were able to verify that under some parameter-settings, the protocol guarantee that one device will eventually discover the frequency of its peer device. The experiment is also interesting since we have not heard of any similar result on the full model-checking of the protocol.

2. Previous work

Symbolic Trajectory Evaluation(STE) [34], or *called symbolic simulation*, is the main alternative to symbolic model checking [7], in formal hardware verification. STE can be considered a hybrid approach based on symbolic simulation and model checking algorithms and can verify assertions, which express safety properties.

STATEMATE [22] is a tool set with a heavy graphical orientation and powerful simulation capability. Users specify systems from three points of view: structural, functional, and behavioral. Three graphical languages, includes module-charts, activity-charts, and state-charts, are supported for the three views. The STATEMATE provides simulation control language(SCL) to enable user to program the simulation. Breakpoints can also be incorporate into the programs in SCL. It may cause the simulation to stop and take certain actions. Moreover, the simulation trace is recorded in trace database, and can be inspected later. The users may view a trace as a discrete animation of state-charts.

The MT-Sim [10] provides simulation platform for the Modechart toolset(MT) [16], which is a collection of integrated tools for specifying and analyzing real-time systems. MT-Sim is a flexible, extensible simulation environment. It supports user-defined viewers, full user participation via event injection, and assertion checking which can invoke user-defined handlers upon assertion violation.

In [21], IOA language and IOA toolset, based on IO automaton, are proposed for designing and analyzing the distributed systems. The toolset can express designs at different levels of abstraction, generate source code automatically, simulate automata, and interface to existing theorem provers. The IOA simulator solves the nondeterminism in IOA language by user-defined determinator specification, random-number generator, and querying the user. IOA simulator provides paired simulation to check the simulation relationship between two automata. It simulates an automaton normally and executes another automaton according to user-defined step correspondence. It is useful in developing systems using levels of abstraction. In [10,16,21,22], tra-

ditional simulation is adopted, so the traces do not have the dimension of width.

UPPAAL [30] is an integrated tool environment for modeling, validation and verification of dense-time systems. It is composed of the system editor, the simulator, and the verifier. The behavior of simulated systems can be observed via the simulator, which can display the systems in many level of details. Besides, the simulator can load diagnostic trace generated by the verifier for further inspection. One technical difference between RED and UPPAAL is that RED uses a BDD-like data-structure, called CRD (Clock-Restriction Diagram) [37–40], for the representation of dense-time state-space while UPPAAL uses the traditional DBM (Difference-Bounded Matrix)[17]. A CRD can represent disjunction and conjunction while a DBM can only represent a conjunction. As a result, the traces do not have the dimension of width in UPPAAL. On the other hand, CRD is more convenient and flexible in manipulating the "*width*" of simulation traces with this advantage. Also in previous experiments [37,38,40], CRD has shown better performance than DBM w.r.t. several benchmarks of dense-time concurrent systems.

3. Communicating timed automata(CTA)

A *communicating timed automaton* (*CTA*) [33,37–39] is a set of finite-state automata, called *process automata*(*PA*), equipped with a finite set of clocks, which can hold nonnegative real-values, and synchronization channels. At any moment, each process automata can stay in only one *mode* (or *control location*). In its operation, one of the transitions can be triggered when the corresponding triggering condition is satisfied. Upon being triggered, the automaton instantaneously transits from one mode to another and resets some clocks to zero. In between transitions, all clocks increase their readings at a uniform rate. Process automata can communicate with one another through binary synchronizations. One of the earliest devices of such synchronizations are the input-output symbol pairs through a channel, in process algebra [24]. Similar synchronization devices have been used in the input languages to HyTech [4], IO Automata [28], UPPAAL [11], Kronos [18], VERIFAST [43], SGM [25, 41,42], and RED [35–39].

For convenience, given a set Q of modes and a set X of clocks, we use $B(Q, X)$ as the set of all Boolean combinations of inequalities of the forms $\texttt{mode} = q$ and $x - x' \sim c$, where mode is a special auxiliary variable, $q \in Q$, $x, x' \in X \cup \{0\}$, "$\sim$" is one of $\leqslant, <, =, >, \geqslant$, and c is an integer constant.

Definition 1. process automata A process automaton A is given as a tuple $\langle X, E, Q, I, \mu, T, \lambda, \tau, \pi\rangle$ with the following restrictions. X is the set of clocks. E is the set of synchronization channels. Q is the set of modes. $I \in B(Q, X)$ is the initial condition on clocks. $\mu : Q \mapsto B(\emptyset, X)$ defines the invariance condition of each mode. $T \subseteq Q \times Q$ is the set of transitions. $\lambda : (E \times T) \mapsto \mathcal{Z}$ defines the message sent and received at each process transition. When $\lambda(e, t) < 0$, it means that process transition t will receive $|\lambda(e, t)|$ events through channel e. When $\lambda(e, t) > 0$, it means that process transition t will send $\lambda(e, t)$ events through channel e. $\tau : T \mapsto B(\emptyset, X)$ and $\pi : T \mapsto 2^X$ respectively defines the triggering condition and the clock set to reset of each transition.

Definition 2. CTA (Communicating Timed Automata) A CTA of m processes is a tuple, $\langle E, A_1, A_2, \ldots, A_m\rangle$ where E is the set of synchronization channels and for each $1 \leqslant p \leqslant m$, $A_p = \langle X_p, E, Q_p, I_p, \mu_p, T_p, \lambda_p, \tau_p, \pi_p\rangle$ is a process automaton for process p.

A *valuation* of a set is a mapping from the set to another set. Given an $\eta \in B(Q, X)$ and a valuation ν of X, we say ν *satisfies* η, in symbols $\nu \models \eta$, iff it is the case that when the variables in η are interpreted according to ν, η will be evaluated *true*.

Definition 3. states Suppose we are given a CTA $S = \langle E, A_1, A_2, \ldots, A_m\rangle$ such that for each $1 \leqslant p \leqslant m$, $A_p = \langle X_p, E, Q_p, I_p, \mu_p, T_p, \lambda_p, \tau_p, \pi_p\rangle$. A state ν of S is a valuation of $\bigcup_{1\leqslant p\leqslant m}(X_p \cup \{\texttt{mode}_p\})$ such that

- $\nu(\texttt{mode}_p) \in Q_p$ is the mode of process i in ν; and
- for each $x \in \bigcup_{1\leqslant p\leqslant m} X_p$, $\nu(x) \in \mathcal{R}^+$ such that $\mathcal{R}^+$ is the set of nonnegative real numbers and $\nu \models \bigwedge_{1\leqslant p\leqslant m} \mu_p(\nu(\texttt{mode}_p))$.

For any $t \in \mathcal{R}^+$, $\nu + t$ is a state identical to ν except that for every clock $x \in X$, $\nu(x) + t = (\nu + t)(x)$. Given $\bar{X} \subseteq X$, $\nu\bar{X}$ is a new state identical to ν except that for every $x \in \bar{X}$, $\nu\bar{X}(x) = 0$.

Now we have to define what a legitimate synchronization combination is in order not to violate the widely accepted interleaving semantics. A *transition plan* is a mapping from process indices p, $1 \leqslant p \leqslant m$, to elements in $T_p \cup \{\perp\}$, where $\perp$ means no transition (i.e., a process does not participate in a synchronized transition). The concept of transition plan represents

which process transitions are to be synchronized in the construction of an LG-transition.

A transition plan is *synchronized* iff each output event from a process is received by exactly one unique corresponding process with a matching input event. Formally speaking, in a synchronized transition plan Φ, for each channel e, the number of output events must match with that of input events. Or in arithmetic, $\sum_{1\leqslant p\leqslant m;\Phi(p)\neq\perp}\lambda(e,\Phi(p))=0$.

Two synchronized transitions will not be allowed to occur at the same instant if we cannot build the synchronization between them. The restriction is formally given in the following. Given a transition plan Φ, a *synchronization plan* Ψ_Φ for Φ represents how the output events of each process are to be received by the corresponding input events of peer processes. Formally speaking, Ψ_Φ is a mapping from $\{1,\ldots,m\}^2\times E$ to $\mathcal{N}$ such that $\Psi_\Phi(p,p',e)$ represents the number of event e sent form process p to be received by process p'. A synchronization plan Ψ_Φ is *consistent* iff for all p and $e\in E$ such that $1\leqslant p\leqslant m$ and $\Phi(p)\neq\perp$, the following two conditions must be true.

- $\sum_{1\leqslant p'\leqslant m;\Phi(p')\neq\perp}\Psi_\Phi(p,p',e)=\lambda(\Phi(p))$;
- $\sum_{1\leqslant p\leqslant m;\Phi(p)\neq\perp}\Psi_\Phi(p',p,e)=-\lambda(\Phi(p))$;

A synchronized and consistent transition plan Φ is *atomic* iff there exists a synchronization plan Ψ_Φ such that for each two processes p,p' such that $\Phi(p)\neq\perp$ and $\Phi(p')\neq\perp$, the following transitivity condition must be true: there exists a sequence of $p=p_1,p_2,\ldots,p_k=p'$ such that for each $1\leqslant i<k$, there is an $e_i\in E$ such that either $\Psi_\Phi(p_i,p_{i+1},e_i)>0$ or $\Psi_\Phi(p_{i+1},p_i,e_i)>0$. The atomicity condition requires that each pair of meaningful process transitions in the synchronization plan must be synchronized through a sequence of input-output event pairs. A transition plan is called an *IST-plan* (*Interleaving Semantics Transition-plan*) iff it has an atomic synchronization plan.

Finally, a transition plan has a *race condition* iff two of its process transitions have assignment to the same variables.

Definition 4. runs Suppose we are given a CTA $S=\langle E,A_1,A_2,\ldots,A_m\rangle$ such that for each $1\leqslant p\leqslant m$, $A_p=\langle X_p,E,Q_p,I_p,\mu_p,T_p,\lambda_p,\tau_p,\pi_p\rangle$. A *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 $\nu_0\models I_p$ and $t_0t_1\ldots t_k\ldots$ is a monotonically increasing real-number (time) divergent sequence, and for all $k\geqslant 0$,

- for all $t\in[0,t_{k+1}-t_k]$, $\nu_k+t\models\bigwedge_{1\leqslant p\leqslant m}\mu_p(\nu_k(\mathtt{mode}_p))$; and
- either
  - $\nu_k(\mathtt{mode}_p)=\nu_{k+1}(\mathtt{mode}_p)$ and $\nu_k+(t_{k+1}-t_k)=\nu_{k+1}$; or
  - there exists a race-free IST-plan Φ such that for all $1\leqslant p\leqslant m$,
    - either $\nu_k(\mathtt{mode}_p)=\nu_{k+1}(\mathtt{mode}_p)$ or $(\nu_k(\mathtt{mode}_p),\nu_{k+1}(\mathtt{mode}_p))\in T_p$ and
    - $\nu_k+(t_{k+1}-t_k)\models\bigwedge_{1\leqslant p\leqslant m;\Phi(p)\neq\perp}\tau_p(\nu_k(\mathtt{mode}_p),\nu_{k+1}(\mathtt{mode}_p))$ and
    - $(\nu_k+(t_{k+1}-t_k))\mathtt{concat}_{1\leqslant p\leqslant m;\Phi(p)\neq\perp}\pi_p(\nu_k(\mathtt{mode}_p),\nu_{k+1}(\mathtt{mode}_p))=\nu_{k+1}$. Here $\mathtt{concat}(\gamma_1,\ldots,\gamma_h)$ is the new sequence obtained by concatenating sequences $\gamma_1,\ldots,\gamma_h$ in order.

We can define the TCTL model-checking problem of timed automata as our verification framework. Here we adopt the safety-analysis problem as our verification framework for simplicity. A safety analysis problem instance, $\mathrm{SA}(A,\eta)$ in notations, consists of a timed automaton A and a safety state-predicate $\eta\in B(Q,X)$. A is *safe* w.r.t. to η, in symbols $A\models\eta$, iff for all runs $(\nu_0,t_0)(\nu_1,t_1)\ldots(\nu_k,t_k)\ldots\ldots$, for all $k\geqslant 0$, and for all $t\in[0,t_{k+1}-t_k]$, $\nu_k+t\models\eta$, i.e., the safety requirement is guaranteed.

In Fig. 1, we have drawn three process automata, in a bus-contending systems. Two process automata are for senders and one for the bus. The circles represent modes while the arcs represent transitions, which may be labeled with synchronization symbols (e.g., `!start`, `?end`, `!collision`, ...), triggering conditions (e.g., $x\leqslant 5$), and assignments (e.g., $x:=0$;). Each transition (arc) in the process automata is called a *process transition*. For convenience, we have labeled the process transitions with numbers. In the system, a sender process may synchronize through channel `start` with the bus to start sending signal on the bus. While one sender is using the bus, the second sender may also synchronize through channel `start` to start placing message on the bus and corrupting the bus contents. When this happen, the bus then signals bus `collision` to both of the senders.

We adopt the standard interleaving semantics, i.e., at any instant, at most one *legitimate global transition* (*LG-transition*) can happen in the CTA. A process transition may not represent an LG-transition and may not be executed by itself. Only LG-transition can be executed. Symbols `start`, `end`, and `collision`, on the arcs, represent synchronization channels, which serve as glue to combine process transitions into LG-transitions. An exclamation (question) mark followed

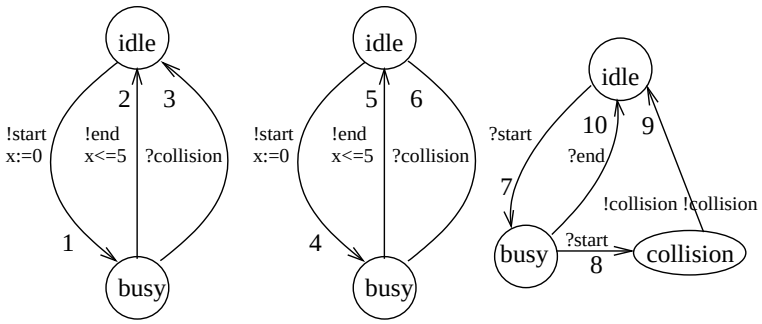

Fig. 1. The model of bus-contending systems.

by a channel name means an *output* (*input*) event through the channel. For example, `!start` means a sending event through channel `start` while `?start` means a receiving event through the same channel. Any input event through a channel must match, at the same instant, with a unique output event through the same channel. Thus, a process transition with an output event must combine with another process transition (by another process) with a corresponding input event to become an LG-transition.

Thus the synchronizers in our input language are primarily used to help users in decomposing their programs into modules and to help the simulators to glue process transitions in constructing LG-transitions. For example, in Fig. 1, process transitions 1 and 7 can combine to be an LG-transition. Also process transitions 3, 6, and 9 can make an LG-transition since two output events matches two input events through channel `collision`.

In the following, we illustrate how to reason in one step of our simulator engine to construct the state-predicate of the next-step. Intuitively, in one step, the system will progress in time and then execute an LG-transition. For example, we may have a current state-predicate

$$(p = 1 \wedge q = 2) \vee (q = 4 \wedge 1 \leqslant x < 3) \qquad \text{(P)}$$

and an LG-transition expressed as the following guarded command:

$$(p = 1 \wedge x > 5) \longrightarrow x := 0; p := 3; \qquad \text{(X)}$$

which means

"when $(p = 1 \wedge x > 5)$ is true with x as a clock, reset x to zero and assign 3 to p."

In a step of the simulation engine, we first calculate the new state-predicate obtained from states in (P) by letting time progress. This affects the constraint on clock x and yields

$$(p = 1 \wedge q = 2) \vee (q = 4 \wedge 1 \leqslant x) \qquad \text{(P')}$$

Then we apply the LG-transitions, selected by the users, to (P') to obtain the state-predicate representing states after the selected transitions. Suppose the only selected LG-transition is (X). Then the state-predicate at the next-step is

$$p = 3 \wedge x = 0 \wedge (q = 2 \vee q = 4)$$

Details can be found in [23].

4. Overview of our simulator

We have incorporated the idea in this report in our verification tool, RED 4.0, a TCTL model-checker/simulator [35–39]. The tool can be activated with the following command in Unix environment:

`$ red [options]` *InputFileName OutputFileName*

The options are

- `-Sp`: symmetry reduction for pointer data-structure systems [48]
- `-Sg`: Symmetry reduction for zones [20,39],
- `-c`: Counter-example generation
- `-s`: Simulator mode with GUI

Without option `-s`, the tool serves as a high-performance TCTL model-checker in backward analysis. When the simulation mode GUI is activated, we will see the window like Fig. 2 popping up. The GUI window is partitioned into four frames respectively of trace trees (on the upper-left corner), current state-predicates (on the bottom), command buttons (in the middle), and candidate process transitions (PT-frame, on the upper-right corner) to be selected and already been selected.

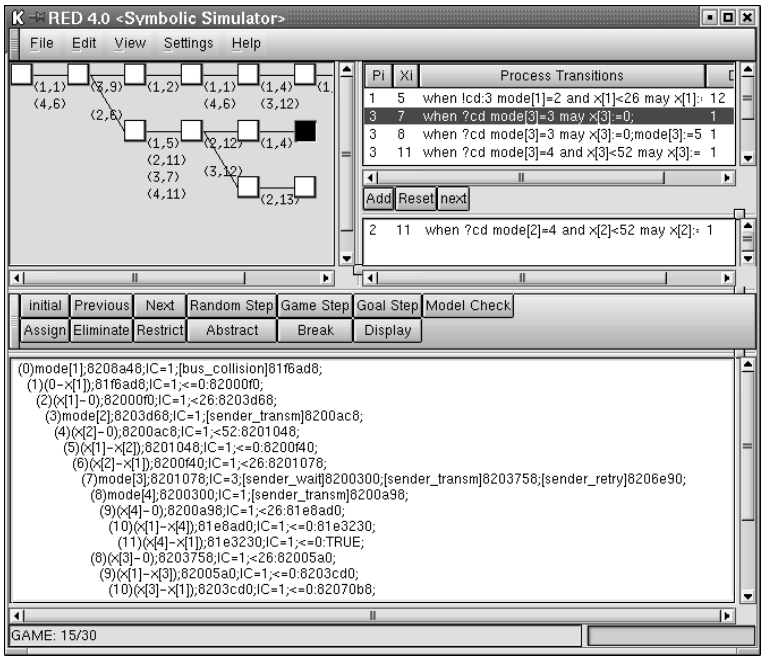

Fig. 2. The GUI of RED 4.0.

Users can construct LG-transitions by selecting process transitions step-by-step in the PT-frame. At each step, the PT-frame displays all process transitions that can be fired at the current state-predicate in the upper-half of the PT-frame. After the selection of a process transition , our simulator is intelligent enough to eliminate those process transitions not synchronizable with those just-selected ones from the display of PT-frame.

After the selection of many process transitions, the simulator steps forward and computes the new current state-predicate at the next step with the LG-transitions constructable from the selected process transitions. If there are many process transitions waiting to be selected at the time the simulator steps forward, all those process transitions will be selected. Since these process transitions may belong to different LG-transitions, the new current state-predicate may represent the result of execution of more than one LG-transitions. This capability to manipulate a state-space represented in a complex state-predicate in symbolic steps is indeed the strength of symbolic simulation.

The architecture of our implementation is shown in Fig. 3. We explain briefly its components in the following:

- *TC compiler*: The efficient compiler can parse the TC programs and OVL assertions, and then mechanically generate the optimized CTA and TCTL formulae.

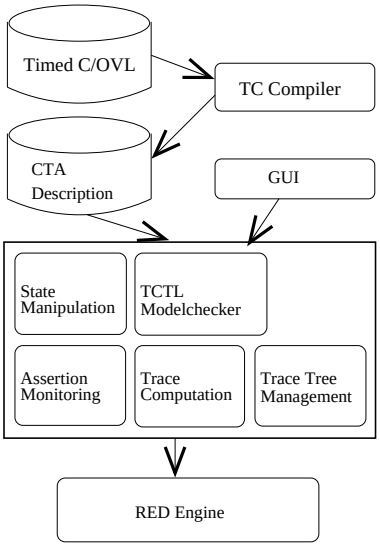

Fig. 3. The architecture of RED model-checker/simulator.

- *GUI* (*graphical user-interface*): A user-friendly window for easy access to the power of formal verification.
- *RED symbolic simulation engine*: This is actually the timed-transition next-step state-predicate calculation routine in forward analysis. Symbolic algorithm for this next-step state-predicate calculation routine is explained at the end of last section

and can also be found in [23].

- *assertion monitoring*: In the input language to the simulator, users can also specify a *goal predicate* for the traces. This goal predicate can be a risk condition, which the users want to make sure that it cannot happen. Or it can be a liveness condition, which the users want to see that it can happen. After each step of the simulation engine, our RED 4.0 will check if the inter of the goal predicate and the next-step state-predicate is nonempty. If it is, the sequence of LG-transitions leading from the initial state to this goal predicate can be displayed. Such a capability is indispensable in helping the users debugging their system designs.
- *trace computation*: This component uses user-guidance, randomness, and various policies to select LG-transitions, in the generation of traces by repetitive invoking the RED symbolic simulation engine. More details is given in section 5.
- *state manipulation*: This includes facilities to inject faults, to either relax or restrict the current state-space, and to set symbolic breakpoints.
- *trace tree management*: (See the frame at the upper-left corner.) This component is for the maintenance of the trace tree structure and movement of current state nodes in the tree. The simulator can step forward and backtrack according to the plain interaction. After a few times of these forward-backward steps, a tree of traces is constructed and recorded in our simulator to represent the whole history of the session. The node for the current state-predicate is black while the others are white. Users can also click on nodes in the trace tree and jump to a specific current state-predicate. On the arcs, we also label the set of pairs of processes and process transitions used in the generation of the next state-predicate.
- *RED symbolic TCTL model-checker*: The high performance backward analysis power of RED can be directly activated to check if the system model satisfies the assertion.

5. Trace computations

As mentioned in the introduction, symbolic simulation adds one new dimension of trace "width", which reflects the number of fired LG-transitions in each step in the construction of traces. With Red 4.0, users may choose from various options to construct traces with appropriate randomness, special search policy, and enough width. The options are:

- *plain interaction:* With selection of process transitions from the PT-frame and previous/next step commands, users have total control on how to select process transitions to make LG-transitions in the construction of the next-step state predicates along the current trace.
- *random steps:* The simulator could also randomly choose an LG-transition in each step. Users can command the autonomous execution of a given number of random steps.
- *game-based policy:* We use the term *"game"* here because we envision the concurrent system operation as a game. Those processes, which we want to verify, are treated as *players* while the other processes are treated as *opponents*. In the game, the players try to win (maintain the specification property) under the worst (i.e., minimal) assumption on their opponents.
  A process is a *player* iff its local variables appear in the goal state-predicate. Intuitively, the simulator constructs a trace segment with all possible reactions of the players in response to random behaviors of the opponents. With this option, we can observe the behavior of players' response to opponents' action. According to the well-observed discipline of modular programming [31], the behavioral correctness of a functional module should be based on minimal assumption on the environment. If we view the players as the functional module and the opponents as the environment, then this *game-based policy* makes a lot of sense.
  It can be useful when we try to verify the design of the player processes. In other words, at each step, the simulator is growing the trace with a width enough for one process transition from each opponent and all firable process transitions from players. Users can again command the autonomous execution of a few steps with this game-based policy.
- *goal-oriented policy:* This policy makes the simulator to generate fast traces leading to the goal states. This can be useful in debugging the system designs, when users have observed some abnormal states. The users can specify the abnormal states as the goal assertions.
  RED 4.0 achieves this by defining the *heuristic distance estimation* (*HD-estimation*) from one state to the other (to be explained in the following). Then process transitions which can the most significantly reduce the HD-estimation from any states in the current state-predicate to any states

in the goal state-predicate will be selected in the hope of a short trace to the goal states can be constructed.

The *HD-estimation* from one (global) state s to another s' is defined as follows. Suppose we have m processes and $s(p)$ is the mode in process p's automaton in state s. Then HD-estimation from s to s' is the sum, over all processes p, of the shortest path distance from $s(p)$ to $s'(p)$ in the graph (constructed with modes as nodes and process transitions as arcs) of process p's automaton. For each porcess p, the shortest path distance is gained from the backward breath-first algorithm.

For VLSI, usually people adopt the estimation of Hemming distance, which measures the number of bit-differences. But for dense-time concurrent systems, state-predicates are loaded with clock constraints and Hemming distance can be difficult to define in a meaningful way.

6. Manipulation of current state-predicate

Our simulator allows for the modification of the current state-predicate before proceeding to the next-step. The following methods can be used to manipulate the current state-predicate and affects the "width" of traces.

- `assign`: The simulator allows users to assign a new value to a state-variable. This can be used to change the behavior of the systems and insert faults.
- `eliminate`: By this method, users can eliminate all constraints w.r.t. a state-variable. This is equivalent to broadening the width of the trace on the dimension of the corresponding state-variable. We can observe the system behavior with less assumption on state-variables.
- `restrict`: In opposition to elimination, users can type in a new predicate and conjunct it with the current state-predicate. With this capability, we can narrow the width of the trace and focus on the interesting behaviors.
- `abstract`: As in the paragraph of game-based policy in page 7, we view the behavior of the target system as a game process and players, opponents can be identified. According to this, the simulator provides three abstract image functions to systematically abstract the current state-predicate. This is also equivalent to systematically broadening the width of the trace. The options for the abstract image functions are:
  - *Game-abstraction*: The game abstract image function will eliminate the state information of the opponents from its argument.
  - *Game-discrete-abstraction*: This abstract image function will eliminate all clock constraints for the opponents in the state-predicate.
  - *Game-magnitude-abstraction*: A clock constraint like $x - x' \sim c$ is called a *magnitude constraint* iff either x or x' is zero itself (i.e. the constraint is either $x \sim c$ or $-x' \sim c$). This abstract image function will erase all non-magnitude constraints of the opponents in the state-predicate.

Note that some of these methods can significantly simplify the representation of the current state-predicate. This also implies that the time and space needed to calculate the next-step state-predicates can be reduced. For example, we may have clocks x_1, x_2 as local clocks of processes 1 and 2 respectively. After applying the game-magnitude-abstraction image function to $x_1 \geqslant 4 \wedge x_2 \geqslant 3 \wedge (x_1 - x_2 \leqslant -2 \vee x_2 - x_1 \leqslant -1)$, we get $x_1 \geqslant 4 \wedge x_2 \geqslant 3$ and have changed a concave state-space down to a convex state-space. This kind of transformation usually can significantly reduce the time and space needed for calculating the next-step state-predicates.

7. Timed C

Today, model-checkers for real-time systems base on mathematical models, such as CTA, Petri net, hybrid automata, etc [8,11,18,25,37,38,42,44,49]. Since most programs for real-time systems are written imperative programming laguages such as C, being able to automatically translate C-programs into CTAs would make our symbolic simulator more attractive for practical use. Our input language TC serves as an intermediate language from C-programs to CTAs. Moreover, in addition to providing this C-like language for system description, an expressive method for writing specifications should also be provided. In traditional programming languages, assertions are inserted between code lines to ensure asserted properties during run time. TC supports assertions derived from a basic subset of OVL assertions in the form of comment lines. In the following sections, we breifly describe TC constructs and OVL assertion semantics using a simple example. Details regarding the automated translation procedures are given in [47].

7.1. Time C Constructs

The TC language adopts basic C-program constructs. A TC program construct **B** could be:

- an atomic assignment: e.g., $y = 3$;
- a sequence statement: e.g., $\mathbf{B}_1\mathbf{B}_2$
- a while-loop statement: e.g., while$(x < 3)\mathbf{B}$
- an if-else statement: e.g., if$(x < 3)$ $\mathbf{B_1}$else$\mathbf{B_2}$
- a switch-case statement: e.g., switch $(y)\{$case1 : $\mathbf{B_1}\ldots\}$

However, the traditional program constructs in C-like languages do not capture all the elements in the modeling of real-time concurrent systems. One deficiency is that there is no way to tell at what time the next statement should be executed. In other words, users cannot describe the deadlines, earliest starting time of the next statement after the execution of the current statement. Here we propose a new type of statement, the *interval* statement, in the forms of "$[c, d]$;", "$[c, d)$;", "(c, d);", "$(c, d]$;", where $c \in \mathcal{N}$ and $d \in \mathcal{N} \cup \{\infty\}$ such that $c \leqslant d$ and $(c, \infty], [c, \infty]$ are not allowed. An interval statement, say $[c, d]$, is not executed but serves as a glue to bind the execution times of its predecessor and successor statements. For example, a statement sequence like $\mathbf{B}_1\ [3, 5]; \mathbf{B}_2$ means that the time lap from the execution of the last atomic statement in $\mathbf{B}_1$ to the execution of the first statement in $\mathbf{B}_2$ is within [3,5]. From real-world C-programs, interval statements can be obtained by abstracting out the execution time of blocks or sequences of program statements. Accurate execution time can be obtained with techniques of WCET [19] analysis. In many embedded systems, a processor exclusively executes one process and the execution time of a straight-line program segment can be obtained by accumulating the execution time (from CPU data-book) of the machine instructions in the segment.

Event-handling is an essential element in modeling languages for real-time systems. With different events observed, the systems may have to take different actions. We design the new construct of

switch event{
case$\langle ss_1\rangle : B_1$break;
case$\langle ss_2\rangle : B_2$break;
...
timeout$[c, d] : B_t$break;
}

to capture this kind of system behaviors. $ss_1, ss_2, \ldots$ are sequences of synchronization labels, like ?receive, !send, etc. The construct means that the system will wait for any of the event combinations of $\langle ss_1\rangle, \langle ss_2\rangle, \ldots$ to happen and take the corresponding actions $B_1, B_2, \ldots$ respectively. But the system will only wait for a period no longer than d time units because of the timeout event which will happen between c and d time units. Finally we also allow programmers to use synchronizers in TC for the convenience of modeling of concurrent behaviors and construction of LG-transitions. For example, users can also write an atomic statement like "< ?ack !finish >;".

7.2. The railroad crossing example

The TC program in Table 1 models a simple railroad crossing system.

The system consists of two processes: monitor and gate_controller, both executing infinite while-loops. In the beginning, we declare two variables of enumerate type, as in Pascal. The first value in the enumerated value set is the initial value of the declared variables.

After sending out a synchronization signal !TRAIN_NEAR, train_status will be assigned value ATCROSSING in 100 to 300 time units. If in between two statements there is no interval statements, it is equivalent to the writing of interval $[0, \infty)$. Lines beginning with // are comments, in which we can write OVL assertions. In this program, there are two OVL assertions which are explained in Section 7.3.

7.3. OVL assertions

We allow four types of OVL assertions inserted in TC as below.

//assert_always(ϕ)
//assert_never(ϕ)
//assert_change#$(\mathcal{I}, f)ID(\phi_1, \phi_2)$
//assert_time#$(\mathcal{I}, f)ID(\phi_1, \phi_2)$

Here ϕ, ϕ_1, ϕ_2 are Boolean predicates on variable values. $\mathcal{I}$ is an interval. f is a special flag. ID is the name of the assertion.

We choose these four assertion types from OVL as examples because many other assertion types can be treated with similar technique, which we use for these four types. In the four assertion types, //assert_always(ϕ) and //assert_never(ϕ) specify some properties at the current state. The first type

Table 1
A TC program for the railroad crossing system

```
enum {NOT_ATCROSSING, ATCROSSING}   train_status;
enum {NOT_DOWN, DOWN}   gate_status;

process monitor() {
  while (1) {
//assert_change #([0,20], 1) A1(train_status == ATCROSSING, train_status == NOT_ATCROSSING)
    <!TRAIN_NEAR>;
    (100,300);
    train_status = ATCROSSING;
//assert_always(gate_status == DOWN)
    [5,10];
    train_status = NOT_ATCROSSING;
    [0,0];
    <!TRAIN_LEAVE>;
    [100,oo];
  }
}

process gate_controller() {
  while (1) {
    <?TRAIN_NEAR>;
    [20,50);
    gate_status = DOWN;
    <?TRAIN_LEAVE>;
    [0,50];
    gate_status = NOT_DOWN;
  }
}
```

$$//\texttt{assert\_always}(\phi)$$

means that "now ϕ must be true." For example, in Table 1, the second assertion in the while-loop of process monitor says that "now the gate must be down."

The second type

$$//\texttt{assert\_never}(\phi)$$

means that "now ϕ must not be true."

The other two assertion types specify some properties along all computations from the current state. f is a flag specific to `assert_change` and `assert_time`. When $f = 0$,

$$//\texttt{assert\_change}\#(\mathcal{I}, f)ID(\phi_1, \phi_2) \qquad (1)$$

means that from now on, along all traces, THE FIRST TIME WHEN ϕ_1 is true, from that ϕ_1-state on, ϕ_2 must change value once within time in $\mathcal{I}$. That is, every time this assertion is encountered, it will only be used once, when ϕ_1 is true, and then discarded.

When $f = 1$, assertion Eq. (1) means that from now on, along all traces, WHENEVER ϕ_1 is true, ϕ_2 must change value once within time in $\mathcal{I}$. That is, this assertion will be assured once and for all. For example, in Table 1, the first comment line in the while-loop of process monitor, is an `assert_change`, which says that when a train is at the crossing (`train_status == ATCROSSING`), then Boolean value of predicate `train_status == NOT_ATCROSSING` must change within 0 to 20 time units.

We have to make a choice about how to interpret "THE FIRST TIME" in a dense-time multiclock system. OVL assertions were originally defined to monitor *events* in VLSI circuits with the assumption of a discrete-time global clock [9]. In synchronous circuits, an atomic event can happen at a clock tick or sometimes can be conveniently interpreted as true in the whole period between two clock ticks. We believe the latter convenient interpretation is more suitable for this work because in concurrent systems, it is not true that all processes will change states at the tick of a "global clock." And this period between two ticks can be interpreted as a state in a state-transition system. According to this line of interpretation, we shall interpret assertion Eq. (1) as

"from now on, along all traces, in THE FIRST INTERVAL WITHIN WHICH ϕ_1 is true, from every state in that interval,

ϕ_2 must change value once within time in $\mathcal{I}$.
to better fit the need of dense-time concurrent systems. This choice of interpretation may later be changed to fit all domains of applications.

The last assertion

$$//\texttt{assert\_time}\#(\mathcal{I}, f)ID(\phi_1, \phi_2) \quad (2)$$

is kind of the opposite to `assert_change`. When $f = 0$, it means that from now on, along all traces, in THE FIRST INTERVAL WITHIN WHICH ϕ_1 is true, from every state in that interval, ϕ_2 must not change value at any time in $\mathcal{I}$. Similarly, when $f = 1$, assertion (2) means that from now on, along all traces, WHENEVER ϕ_1 is true, ϕ_2 must not change value at any time in $\mathcal{I}$.

8. Experiments on Bluetooth baseband protocol

In the following, we first give a brief introduction to the Bluetooth baseband protocol [26]. Then we present our model of baseband protocol in CTA in Subsection 8.2. The model will be used in two ways:bug-inserted and bug-free. We use two bug-inserted models in Subsection 8.3 and 8.4 respectively, and show how to quickly find the bugs with symbolic traces of Red 4.0. In Subsections 8.3, we also demonstrate how to generate traces to observe system behaviors step by step. Finally, in Subsection 8.5, we use the bug-free model to report the performance in complete verification of the Baseband protocol.

8.1. Bluetooth baseband protocol

Bluetooth is a specification for wireless communication protocols [26]. It operates in the unlicensed Industrial-Scientific-Medical (ISM) band at 2.4 GHz. Since ISM band is open to everyone, Bluetooth uses the frequency hopping spread spectrum (FHSS) and time-division duplex (TDD) scheme to cope with interferences. Bluetooth divides the band into 79 radio frequencies and hops between these frequencies. It is a critical issue for Bluetooth devices to discover the frequencies of other Bluetooth devices since FHSS and TDD scheme are used.

A Bluetooth unit that wants to discover other Bluetooth units enters an INQUIRY mode. A Bluetooth unit that allows itself to be discovered, regularly enters the INQUIRY SCAN mode to listen to inquiry messages. Figure 4 shows the INQUIRY and INQUIRY SCAN procedures. All Bluetooth units in INQUIRY and INQUIRY SCAN share the same hopping sequence, which is 32 hops in length. The Bluetooth unit in INQUIRY SCAN mode hops every 1.28 sec. Although a Bluetooth unit in INQUIRY mode also uses the same inquiry hopping sequence, it does not know which frequencies do receivers listen to. In order to solve this uncertainty, a Bluetooth unit in INQUIRY mode hops at rate of 1600 hop/sec, and transmits two packets on two different frequencies and then listens for response messages on corresponding frequency. Besides, the inquiry hopping sequence is divided into train A and B of 16 frequencies and a single train is repeated for Ninquiry (which is 256 in specification) times before a new train is used. In an error-free environment, at least three train switches must have taken place. Details can be found in [26];

8.2. The system model

Every Bluetooth unit has a system clock. When the clock ticks, the Bluetooth unit updates its internal timer and frequency. So in our model, there are two clocks, `tick_clk_scan` and `tick_clk_inq`, for INQUIRY SCAN (Fig. 5) and INQUIRY (Fig. 6) processes, respectively. For convenience, we have labeled the process transitions with numbers.

Every time unit, the processes loop through the modes to update the variables. For the INQUIRY SCAN procedure, there are two important variables, `inqscanTimer_` and `mode_scan`. Variable `inqscanTimer_`, which is a timer updated in transitions 6 to 9, is used to determine when to enter INQUIRY SCAN mode. Variable `mode_scan` records the current mode of the process performing the INQUIRY SCAN procedure, and its value may be INQUIRY_SCAN or STANDBY.

For the INQUIRY procedure, when the value of variable `clkmod`, in transitions 13 to 16, is less than 2, the process transmits packets. Otherwise, it listens for response messages. The process sends packets via synchronization channel in transitions 19 and 20. If a packet is received successfully, it means that the frequency, through which the packet is received, is discovered and the process goes to SUCCESS mode. Otherwise, in transitions 21 to 24, variables `id_sent`, `train_sent`, and `train_switch` are changed. Variable `id_sent` records the packets sent in current train; variable `train_sent` records the number of repeat of a single train; variable `train_switch` represents how many train switches have taken place.

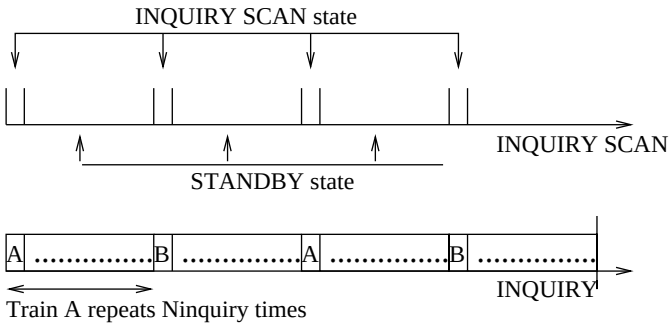

Fig. 4. Mode sequences of processes INQUIRY and INQUIRY SCAN in baseband protocol.

After three train switches, the process goes to TIME-OUT mode via transition 25.

Our task is to verify whether two Bluetooth units in complementary modes will hop to the same frequency before timeout, so that the INQUIRY and INQUIRY SCAN procedures can go on. One can think of a printer equipped with Bluetooth in INQUIRY SCAN mode. When a notebook equipped with Bluetooth has data to print, it will inquiry nearby printers. We anticipate that the notebook can learn the existence of the printer with the Bluetooth protocols.

8.3. Using "width" of simulation traces for advantage

In this subsection, a bug is inserted in the INQUIRY SCAN process in the model. We demonstrate how to properly control the "width" of symbolic traces to quickly discover the bug, and manipulate the state-space predicate to pseudo-correct the bug. In the end of the simulation, we use game-based policy to automatically trace to our goal states.

We use the step sequence shown in the second row of Table 2 to experiment with RED and the Baseband protocol. A pair like (p, x) in the row means that process p executes transition x. When several of these process transition execution pairs are stacked, it means that we select all these process transitions to broaden the trace width of simulation.

In our scenario with notebook and printer, the printer regularly enters the INQUIRY SCAN mode to listen to inquiry messages. The printer will periodically execute in mode INQUIRY SCAN and mode STANDBY in sequence (See the upper mode-sequence in Fig. 4). In the implementation of Baseband protocol, the alternation between these two modes is controlled with counter `inqscanTimer_`, which increments at every clock tick. When $\texttt{inqscanTimer\_} < \texttt{TwInqScan\_c}$ (`TwInqScan_c` is a macro constant defining the scan window size), the printer stays in mode INQUIRY_SCAN. At the time when $\texttt{inqscanTimer\_} = \texttt{TwInqScan\_c}$, the printer changes to mode STANDBY. When counter `inqscanTimer_` increases to macro constant `TinqScan_c` (the time span between two consecutive inquiry scans), it is reset to zero. We want to make sure that an INQUIRY SCAN process will periodically execute in the two modes of

$$\texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \wedge \texttt{mode\_scan} = \text{INQUIRY\_SCAN}$$

and

$$\texttt{inqscanTimer\_} \geqslant \texttt{TwInqScan\_c} \wedge \texttt{mode\_scan} = \text{STANDBY}$$

in sequence. Thus a risk condition saying that this sequence is violated in the following.

$$\left(\begin{array}{l} \left(\begin{array}{l} \texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \\ \wedge\, \texttt{mode\_scan} \neq \text{INQUIRY\_SCAN} \end{array} \right) \\ \vee \left(\begin{array}{l} \texttt{inqscanTimer\_} \geqslant \texttt{TwInqScan\_c} \\ \wedge\, \texttt{mode\_scan} \neq \text{STANDBY} \end{array} \right) \end{array} \right)$$

When the notebook starts to inquiry, the printer may be in mode INQUIRY_SCAN or mode STANDBY. With traditional simulation [10,16,21,22], a precise initial state, such as

$$\texttt{inqscanTimer\_} = 0 \wedge \texttt{mode\_scan} = \text{INQUIRY\_SCAN}$$

must be chosen to start the simulation. And the chosen initial state may either never reach the risk states or have a long way to do it. But in RED 4.0, we can start our simulation from the whole state-space represented by the following state-predicate.

$$\left(\begin{array}{l} \left(\begin{array}{l} \texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \\ \wedge\, \texttt{mode\_scan} = \text{INQUIRY\_SCAN} \end{array} \right) \\ \vee \left(\begin{array}{l} \texttt{inqscanTimer\_} \geqslant \texttt{TwInqScan\_c} \\ \wedge\, \texttt{mode\_scan} = \text{STANDBY} \end{array} \right) \end{array} \right)$$

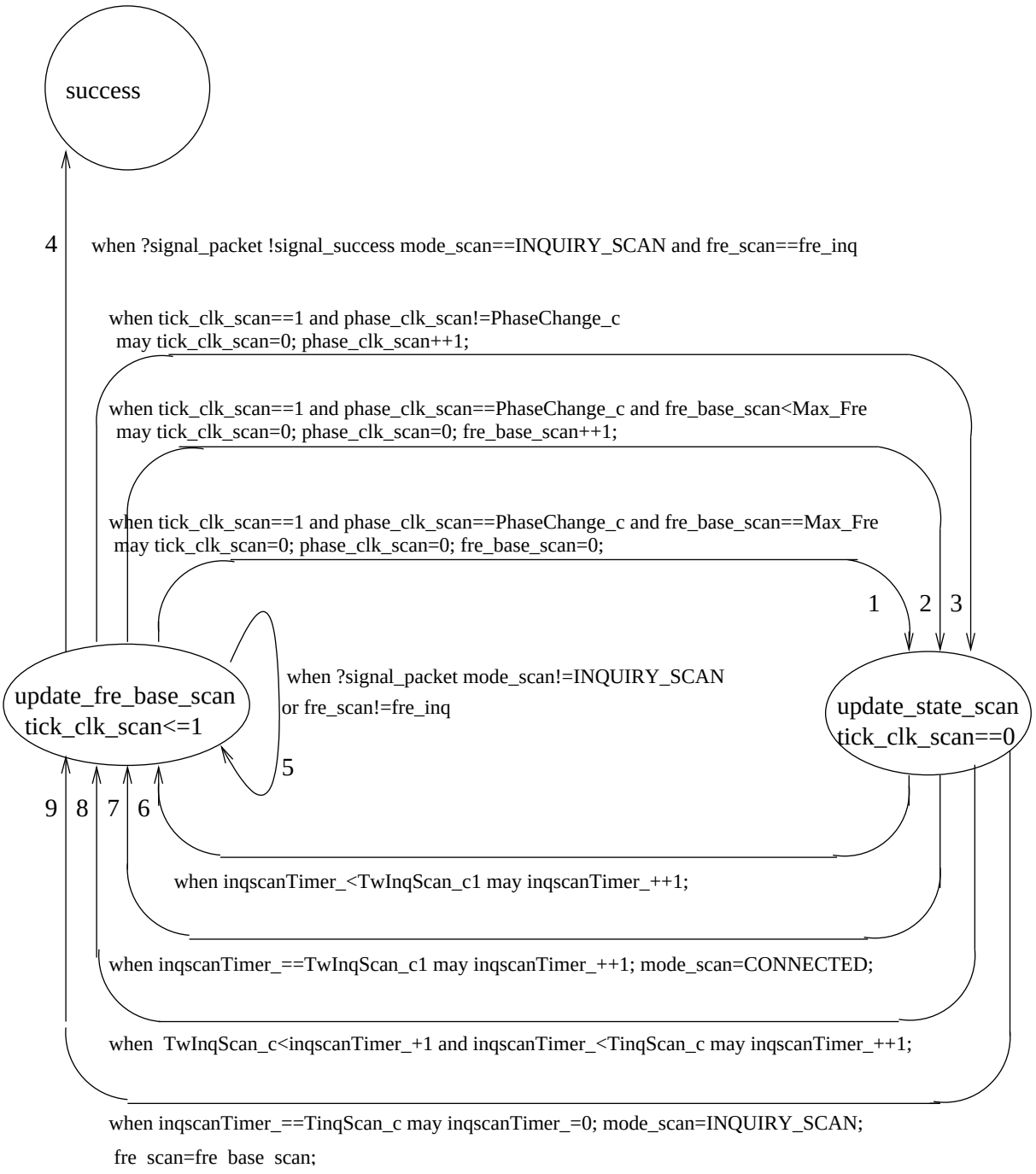

Fig. 5. Inquiry scan.

By starting simulation with this big state-space, we are actually using a great "width" of the symbolic trace and should have much better chance in detecting bugs.

By executing the first five steps in the sequence of Table 2, we simulate the model step by step to observe if the system acts according to our expectation. At the fifth step, we have four executable process transitions, including transitions 6, 7, 8, and 9 (see the arc labels in Fig. 5 of process INQUIRY SCAN. With RED 4.0, we can simulate all these possibilities in a single step.

Table 2
The step-by-step simulation

Step	1	2	3	4	5	6	7	8
Process transitions	(I,13)	(I,17)	(IS,5) (I,20)	(IS,1) (IS,2) (IS,3)	(IS,6) (IS,7) (IS,8) (IS, 9)	`restrict`	`assign`	`game-based policy`

I: process INQUIRY; IS: process INQUIRY_SCAN; (p, x): process p executing process transition x.

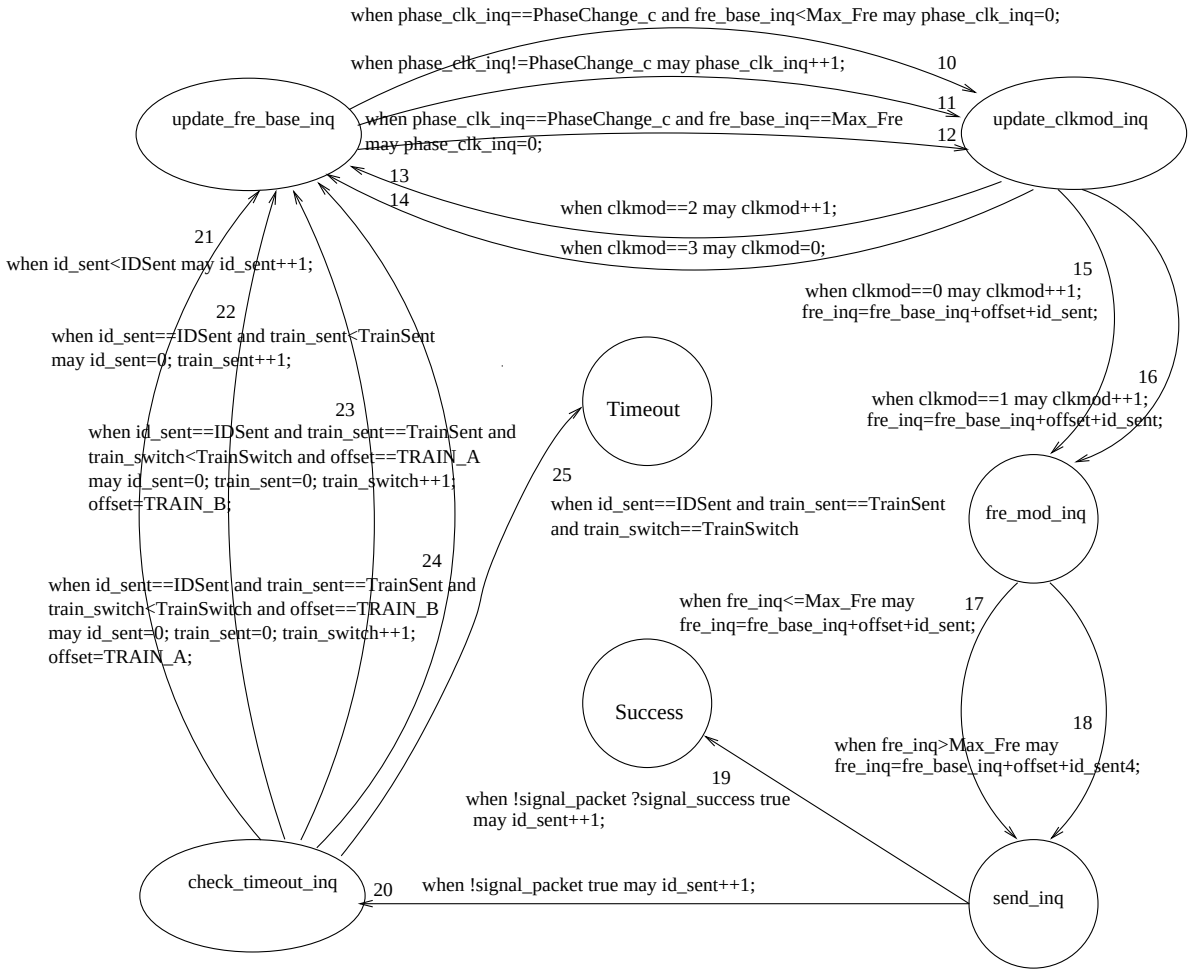

Fig. 6. Inquiry.

Now we want to demonstrate what we can do with the discovery of bugs. After the fifth step, we reach a risk state. Inspecting the trace, we find a bug in transition 7 (see Fig. 5). According to Bluetooth specification [26], when counter `inqscanTimer_` increments from `TwInqScan_c`-1 to `TwInqScan_c`, process INQUIRY SCAN should change from mode INQUIRY SCAN to mode STANDBY. And transition 7 in Fig. 5 is supposed to model this mode change. The bug is inserted by changing the triggering condition of process transition 7 from $\texttt{inqscanTimer\_} = \texttt{TwInqScan\_c} - 1$ to $\texttt{inqscanTimer\_} = \texttt{TwInqScan\_c}$. It means that the printer enters mode STANDBY one tick too late and the system reaches the risk state of

$$\texttt{inqscanTimer\_} = \texttt{TwInqScan\_c} \\ \wedge \texttt{mode\_scan} = \text{INQUIRY\_SCAN}$$

In order to pseudo-correct the bug, we want to test what will happen if the mode change does not

happen in time. To do this what-if analysis, we first restrict our attention to the state-predicate with `inqscanTimer_equals TwInqScan_c`. We do this by keying state-predicate $\texttt{inqscanTimer\_} = \texttt{TwInqScan\_c}$ to restrict the current state-predicate.

Now the new current state-predicate satisfies

$$\texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \wedge \texttt{mode\_scan} = \text{INQUIRY\_SCAN}$$

We want to see whether by correcting the bug of the late mode-change, we can indeed get the correct behavior (i.e. both parties hop to the same frequency). We change the value of `mode_scan` from INQUIRY_SCAN to STANDBY. Then we generate traces automatically and see if we can see any faulty behaviors in the traces constructed with the game-based policy (i.e., all process transitions for players (process INQUIRY SCAN) and random transitions for opponents (process INQUIRY). In our experiment, RED 4.0 constructed a symbolic trace leading to SUCCESS mode. This gives users confidence that the both parties indeed can hop to the same frequency.

8.4. Fast debugging with goal-oriented policy

Here we show how to find bugs in our Baseband model with our goal-oriented policy. The bug is inserted as follows. In transitions 19 and 20, variable `id-sent` is now incremented when a packet is sent. However, this increment is redundant because variable `id_sent` has already been incremented with variables `train_sent` and `train_switch` together in transitions 21 to 24. This bug would make `id_sent` to be incremented by 2 for each packet sent, and causes the INQUIRY process timeout quickly.

We generate directed traces with our goal-oriented policy. The simulator selects transitions that minimize the HD-estimation to the goal state. For example, transition 20 which leads to TIMEOUT mode would be taken rather than transition 19 that leads to SUCCESS mode, since our goal state is TIMEOUT mode which means the existence of a bug. In our first trial, we generate a trace that reaches the TIMEOUT mode, and fix the bug by observing the trace. It costs RED 4.0 8.21 seconds on an Pentium 1.7G MHz desktop with 256 MB memory to generate the directed trace. However, if we do complete verification to generate a counter-example trace, it costs RED 4.0 137.78 seconds.

With random traces, the time needed to find a bug depends on how fast the random traces hit the bug. In our experiment, we generate a random traces, but it does not reach the TIMEOUT mode. Then we have to generate a new trace from the step that may lead to the TIMEOUT mode. Repeating this trial-and-error iterations for six times, we finally reaches the TIMEOUT mode. Our experiment shows that the goal-oriented policy is more efficient in debugging the model as compared with random steps and complete verification.

8.5. Complete verification

Finally, we have finished simulating and debugging our model, and gained confidence in the correctness of our system. We can now proceed to the more expensive step of formal model-checking to see whether two Bluetooth units in complementary modes will hop to the same frequency before timeout. RED 4.0 uses 197 seconds on an Pentium 1.7G MHz desktop with 256 MB memory to check this model.

9. Conclusion

This paper has described RED 4.0, a model-checker/simulator based on BDD-like data-structure with GUI for dense-time concurrent systems. Engineers can describe their systems with a C-like language, TC, and insert OVL assertions as comment lines. RED 4.0 can generate symbolic traces with various policies and manipulate the state-predicate. By properly controlling the width of symbolic traces, we have much better chances in observing what we are interested. The usefulness of our techniques can be justified by our report on experiment with the Bluetooth baseband protocol.

Future work may proceed in several directions. Firstly, we hope to derive new HD-estimation functions used in the directed trace generation, and support customized automatic trace generation policy. These would help users finding bugs with fewer simulation traces. Secondly, the improvement of TC is also important, since TC bridges our tool and industrial systems. Finally, we plan to make our GUI more friendly so that users can have easy access to the power of formal verification.

References

[1] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli and A. Rasse, Data-Structures for the Verification of Timed Automata, Proceedings, HART'97, LNCS 1201.

[2] R. Alur, C. Courcoubetis and D.L. Dill, *Model Checking for Real-Time Systems*, IEEE LICS, 1990.
[3] R. Alur and D.L. Dill, *Automata for modelling real-time systems*, ICALP' 1990, LNCS 443, Springer-Verlag, pp. 322–335.
[4] R. Alur, T.A. Henzinger and P.-H. Ho, *Automatic Symbolic Verification of Embedded Systems*, in Proceedings of 1993 IEEE Real-Time System Symposium.
[5] A.V. Aho, R. Sethi and J.D. Ullman, *Compliers – Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986, pp. 393–396.
[6] J. Bhasker, *A VHDL Primer*, (third edition), ISBN 0-13-096575-8, Prentice Hall, 1999.
[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, IEEE LICS, 1990.
[8] M. Bozga, C. Daws and O. Maler, *Kronos: A model-checking tool for real-time systems*, 10th CAV, June/July 1998, LNCS 1427, Springer-Verlag.
[9] L. Bening and H. Foster, *Principles of Verifiable RTL Design, a Functional Coding Style Supporting Verification Processes in Verilog*, (li 2nd ed.), Kluwer Academic Publishers, 2001.
[10] M. Brockmeyer, C. Heitmeyer, F. Jahanian and B. Labaw, *A Flexible, Extensible Simulation Environment for Testing Real-Time*, IEEE, 1997.
[11] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson and Y. Wang, *UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems*, Hybrid Control System Symposium, 1996, LNCS, Springer-Verlag.
[12] G. Behrmann, K.G. Larsen, J. Pearson, C. Weise and Y. Wang, *Efficient Timed Reachability Analysis Using Clock Difference Diagrams*, CAV'99, July, Trento, Italy, LNCS 1633, Springer-Verlag.
[13] R.E. Bryant, Graph-based Algorithms for Boolean Function Manipulation, *IEEE Trans. Comput.* **C-35**(8), 1986.
[14] E. Clarke and E.A. Emerson, *Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic*, in "Proceedings, Workshop on Logic of Programs", LNCS 131, Springer-Verlag.
[15] E. Clarke, O. Grumberg, M. Minea and D. Peled, State-Space Reduction using Partial-Ordering Techniques, *STTT* **2**(3) (1999), 279–287.
[16] P. Clements, C. Heitmeyer, G. Labaw and A. Rose, *MT: a toolset for specifying and analyzing real-time systems*, in IEEE Real-Time Systems Symposium, 1993.
[17] D.L. Dill, *Timing Assumptions and Verification of Finite-state Concurrent Systems*, CAV'89, LNCS 407, Springer-Verlag.
[18] C. Daws, A. Olivero, S. Tripakis and S. Yovin, *The tool KRONOS*, The 3rd Hybrid Systems, 1996, LNCS 1066, Springer-Verlag.
[19] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gubstafsson and H. Hansson, Worst-case execution-time analysis for embedded real-time systems, *Journal of Software Tools for Technology Transfer* **14** (2001).
[20] E.A. Emerson and A.P. Sistla, Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach, *ACM TOPLAS* **19** (July, 1997), 617–638, Nr. 4.
[21] S.J. Garland and N.A. Lynch, *The IOA Language and Toolset: Support for Designing*, Analyzing, and Building Distributed Systems, Technical Report MIT/LCS/TR.
[22] D. Harel et al., STATEMATE: A working environment for the development of complex reactive systems, *IEEE Trans. on Software Engineering* **16**(4) (1990), 403–414.
[23] T.A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine, *Symbolic Model Checking for Real-Time Systems*, IEEE LICS, 1992.
[24] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
[25] P.-A. Hsiung and F. Wang, User-Friendly Verification. Proceedings of 1999 FORTE/PSTV, Beijing, in: *Formal Methods for Protocol Engineering and Distributed Systems*, J. Wu, S.T. Chanson and Q. Gao, eds, Kluwer Academic Publishers, October, 1999.
[26] J. Haartsen, *Bluetooth Baseband Specification*, version 1.0. http://www.bluetooth.com/.
[27] K.G. Larsen, F. Larsson, P. Pettersson and Y. Wang, *Efficient Verification of Real-Time Systems: Compact Data-Structure and State-Space Reduction*, IEEE RTSS, 1998.
[28] N. Lynch and M.R. Tuttle, An introduction to Input/Output automata, *CWI-Quarterly* **2**(3) (September, 1989), 219–246. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
[29] http://www.verificationlib.com/.
[30] P. Pettersson and K.G. Larsen, *UPPAAL2k*, (Vol. 70), Bulletin of the European Association for Theoretical Computer Science, 2000, pp. 40–44.
[31] R.S. Pressman, *Software Engineering, A Practitioner's Approach*, McGraw-Hill, 1982.
[32] V. Sagdeo, *The Complete VERILOG Book Kluwer Academic Publishers*, 1998, ISBN: 0792381882.
[33] A. Shaw, Communicating Real-Time State Machines, *IEEE Transactions on Software Engineering* **18**(9) (Sept., 1992).
[34] C.-J.H. Seger and R.E. Brant, Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories, *Formal Methods in System Designs* **6**(2) (Mar., 1995), 147–189.
[35] F. Wang, *Efficient Data-Structure for Fully Symbolic Verification of Real-Time Software Systems*, TACAS'2000, March, Berlin, Germany. in LNCS 1785, Springer-Verlag.
[36] F. Wang, *Region Encoding Diagram for Fully Symbolic Verification of Real-Time Systems*, the 24th COMPSAC, Oct. 2000, Taipei, Taiwan, ROC, IEEE press.
[37] F. Wang, RED: Model-checker for Timed Automata with Clock-Restriction Diagram, Workshop on Real-Time Tools, Aug. 2001, Technical Report 2001-014, ISSN 1404-3203, Dept. of Information Technology, Uppsala University.
[38] F. Wang, *Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram, to appear in Proceedings of FORTE*, August, 2001, Cheju Island, Korea.
[39] F. Wang, *Symmetric Model-Checking of Concurrent Timed Automata with Clock-Restriction Diagram*, RTCSA'2002.
[40] F. Wang, Efficient Verification of Timed Automata with BDD-like Data-Structures, Technical Report, IIS, Academia Sinica, 2002.
[41] F. Wang and P.-A. Hsiung, *Automatic Verification on the Large*, Proceedings of the 3rd IEEE HASE, November, 1998.
[42] F. Wang and P.-A. Hsiung, Efficient and User-Friendly Verification, *IEEE Transactions on Computers* (Jan., 2002).
[43] F. Wang and C.-T. Lo, Procedure-Level Verification of Real-Time Concurrent Systems, *International Journal of Time-Critical Computing Systems* **16** (1999), 81–114.
[44] H. Wong-Toi, *Symbolic Approximations for Verifying Real-Time Systems*, Ph.D. thesis, Stanford University, 1995.
[45] F. Wang, G-D. Hwang and F. Yu, *Symbolic Simulation of Real-Time Concurrent Systems to appear in proceedings of RTCSA 2003*, Tainan, Taiwan. Feb., 2003. LNCS, Springer-Verlag.
[46] F. Wang, G-D. Hwang and F. Yu, *Numerical Coverage Estimation for the Symbolic Simulation of Real-Time Systems*

to appear in proceedings of FORTE 2003, Berlin, Germany. Sep., 2003. LNCS, Springer-Verlag.

[47] F. Wang and F. Yu, *OVL Assertion-Checking of Embedded Software with Dense-Time Semantics to appear in proceedings of RTCSA 2003*, Tainan, Taiwan, Feb., 2003. LNCS, Springer-Verlag.

[48] F. Wang and K. Schmidt, Symmetric Symbolic Safety-Analysis of Concurrent Software with Pointer Data Structures, IIS Technical Report, 2002, IIS, Academia Sinica, Taipei, Taiwan, ROC.

[49] S. Yovine, Kronos: A Verification Tool for Real-Time Systems, *International Journal of Software Tools for Technology Transfer* **1**(Nr. 1/2) (October, 1997).