Symbolic Simulation of Real-Time Concurrent Systems [*]

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

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 encompassing many, even densely many, traces in traditional simulation into one symbolic trace. We discuss how we generate traces with various policies, how we handle the issue of code coverage and functional coverage, how we manipulate the state-predicate, and how we manage the trace trees. Finally, we report experiment with our simulator in the verification of the Bluetooth baseband protocol.

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

1 Introduction

Traditional simulation[8, 14, 18] 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 the new system designs in the new century, e.g. System-on-a-Chip (SOC) with tens of millions of gates, there will not be enough time and manpower to run enough number of simulation traces to gain enough functional coverage and confidence of the system designs. The complexity incurred by the system designs in the next few years simply overwhelms the capability of traditional simulation technology.

On the other hand, model-checking technology[12, 2] has promised to mathematically prove the correctness of system design. The development of model-checking with symbolic manipulation techniques[11, 5] has made the full 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. For example, we have procedure to compute the state-predicate at the next-step from the current state-predicate. Such succinctness not only saves the memory space in representation but also allows us to construct a huge (or even dense) set of states in a few symbolic manipulation steps.

But even with such powerful techniques of symbolic manipulation, the verification task of real-time concurrent system still demands tremendous resources beyond the reach of current technology. The 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 (you may 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 huge amount of system resources. Thus it will be helpful and attractive if a technique that makes a balance between the two extremes can be developed.

The technique of *symbolic simulation* represents such a balance[28]. The technique was originally introduced and proved 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 a (convex or concave) space of "current states." In metaphor, traditional simulation is like a probe while the new symbolic simulation technique is like a searchlight into the space and can monitor a set of state-traces at the same time. With proper choice of the caliber of the searchlight, we have much better chance to discover the imminent risk and potential threats in the immense sky.

We have implemented a symbolic simulator, for dense-time concurrent systems, with GUI (Graphical User-Interface), convenient facilities to generate and manage the traces, options for coverage estimations. The simulator is now part of RED 4.0 (http://val.iis.sinica.edu.tw), a model-checker/simulator for real-time systems. In the development of the symbolic simulation function, we encounter the following many challenges and opportunities.

[*]The work is partially supported by NSC, Taiwan, ROC under grants NSC 90-2213-E-001-006, NSC 90-2213-E-001-035, and the by the Broadband network protocol verification project of Institute of Applied Science & Engineering Research, Academia Sinica, 2001.

What is the model we adopt for real-time concurrent systems ?

In simulation, we construct a mathematical model for a system design (and the environment) with computer programs and observe how the model behaves in computer's virtual world. The semantics of the model will determine how precise we can approximate the system/environment interaction and how efficient we can compute the traces.

Symbolic simulation has gained much success in the verification of VLSI circuits, which are usually synchronous. We plan to extend the success in the area of real-time concurrent systems, like communication protocols, embedded softwares, ..., etc. For such systems, the assumption of the existence of a global clock is inappropriate and 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. Even the new SOC can have multi-clocks in the same chip. Based on all these consideration, we adopt the well-accepted timed automata[3], with multiple dense-time clocks, as our system model.

The input language of RED 4.0 allows the description of a timed automaton as a set of process automata communicating with each other through synchronizers (namely, input/output events through channels in [21]) 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 "caliber of the searchlight" to better monitor a user-given goal (or risk) condition along traces.

How do we know when to stop ?

In order to gain sufficient confidence in the system design, we have to know how well the simulation traces have helped us in observing the behaviors we need to observe. Traditional simulation has the concept of *coverage*, like code coverage, path coverage, functional coverage, ..., developed for the verification of VLSI circuits[7]. The concept of coverage is to estimate how much percentage of the behaviors, which users are interested at, has been observed. Such a concept is important because engineers and their companies need a metric to tell them when they can be confident enough in their products.

But it can be difficult in general to design such metrics and make precise estimation for dense-time systems. The challenge comes from the known high complexity of the state-space representations of dense-time systems. According to [2], we can partition the state-space of dense-time systems into small behaviorally equivalent spaces of states, called *regions*. The number of such regions is exponential to the size of input description sizes. It is very difficult to guess in advance which regions will be reachable from the initial states. An imprecise estimation of the whole reachable state-space may lead to very low percentage in coverage while actually enough traces have been generated to cover the whole reachable state-space.

To cope with the challenge, we propose two approaches. First, we developed a metric corresponding to the traditional metric of *code coverage*, which means how much percentage of program codes the simulation traces have used. We propose to use the total number of legitimate global transitions to measure the code coverage. In this approach, the exact percentage of code coverage can be derived.

Second, we develop a technique, based on abstract image functions of state-spaces, to estimate the functional coverage of simulation traces. Functional coverage is specific to the functions, which users want to verify. Our simulator will keep on computing how much percentage of the interesting state-space has been covered for the verification of the relevant functionalities of the systems.

How do we construct and manage traces ?

The traces can be constructed randomly or with a policy. *Random traces* are computed with random number generators without the bias of the designers and verification engineers. Many people do not feel confident with a design until it has been verified with random traces. On the other hand, directed traces are constructed with built-in or user-given policies. *Directed traces* can help in guiding the simulators to program lines which are suspicious of bugs or whose effects need to be closely monitored. With directed traces, the simulators can better efficiently construct the traces that are of interest to the verification engineers.

Symbolic simulation actually adds one more dimension to the issue of random vs. directed traces. Since we can use complex logic constraints to represent a space of states, from steps to steps, we are actually building traces of state-spaces, instead of a single precise state. So it is more like (even densely) many traces are constructed simultaneously. Symbolic simulation thus add the dimension of *"width"* to a trace of state-spaces. In section 5, we shall discuss how to control the width of traces with the many options supported by our simulator.

Organizations of the 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, and 7). Finally, 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)[28], or *called symbolic simulation*, is the main alternative to symbolic model checking[5], 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[19] 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[8] provides simulation platform for the Modechart toolset(MT)[14], 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.

UPPAAL[26] 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)[31, 32, 33, 34], for the representation of dense-time state-space while UPPAAL uses the traditional DBM (Difference-Bounded Matrix)[15]. A CRD can represent disjunction and conjunction while a DBM can only represent a conjunction. With this advantage, CRD is more convenient and flexible in manipulating the *"width"* of simulation traces. Also in previous experiments[31, 32, 34], CRD has shown better performance than DBM w.r.t. several benchmarks of dense-time concurrent systems.

In[18], 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.

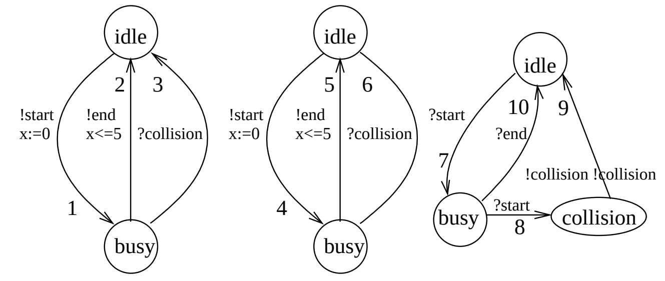


Figure 1: the model of bus-contending systems

3 Synchronized concurrent timed automata

A *timed automaton*[3] is a finite-state automaton equipped with a finite set of clocks that can hold non-negative real-values. At any moment, the timed automaton can stay in only one *mode* (or *control location*). In its operation, one of the global 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 global transitions, all clocks increase their readings at a uniform rate.

In our input language, users can describe the timed automaton as a *synchronized concurrent timed automaton (SCTA)*[31, 32, 33]. Such an automaton is in turn described as a set of *process automata (PA)*. Users can declare local (to each process) and global variables of type clock, integer, and pointer (to identifier of processes). Boolean conditions on variables can be tested and variable values can be assigned. 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[21]. Similar synchronization devices have been used in the input languages to HyTech[4], IO Automata[25], UPPAAL[9], Kronos[16], VERIFAST[37], SGM[22, 35, 36], and RED[29, 30, 31, 32, 33].

In figure 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., !`begin`, ?`end`, !`collision`, ...), triggering conditions (e.g., $x \leq 5$), 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 `begin` 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 `begin` 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 transi-*

tion (LG-transition) can happen in the SCTA. For formal semantics of the systems, please check out appendix A. A process transition may not represent an LG-transition and may not be executed by itself. Only LG-transition can be executed. Symbols `begin`, `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 by a channel name means an *output (input)* event through the channel. For example, !`begin` means a sending event through channel `begin` while ?`begin` 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 figure 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

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

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 let time progress. This affects the constraint on clock x and yields

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

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 [20].

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[29, 30, 31, 32, 33]. 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[38]
- `-Sg`: Symmetry reduction for zones[17, 33],
- `-c`: Counter-example generation
- `-s`: Simulator mode with GUI

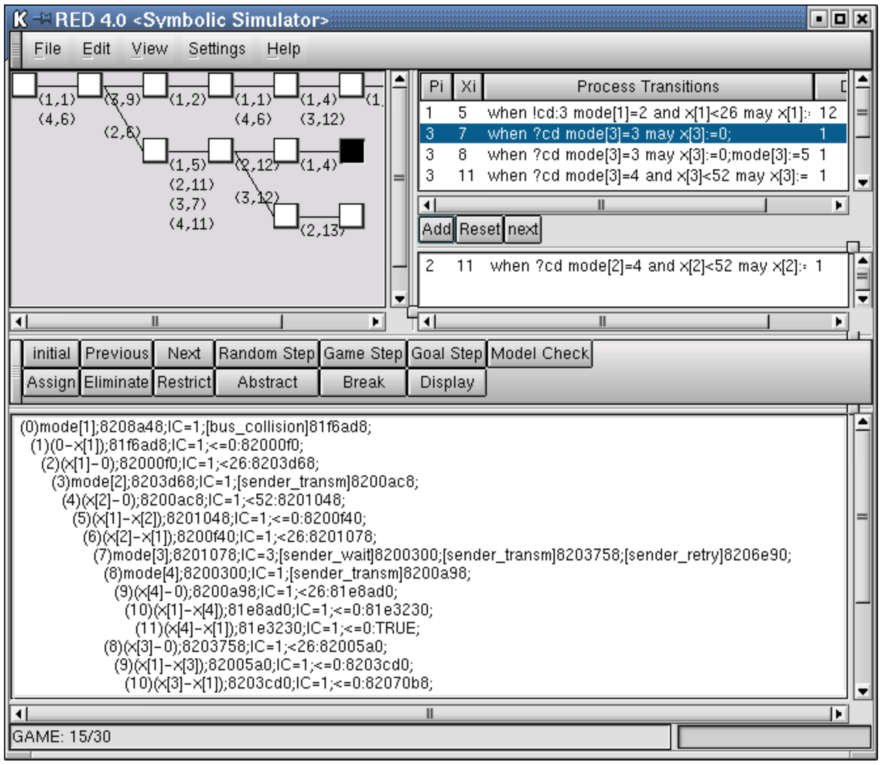
Without option `-s`, the tool serves as a high-performance TCTL model-checker in backward analysis. When the simulation mode GUI is activated, we will see the window like figure 2 popping up. The GUI window is partitioned into four frames respectively of trace trees (on the upper-left corner), current 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.

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 (by clicking the process transition and then clicking on button $\boxed{\texttt{add}}$), 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, for the construction of the next-step state-predicate along the trace, users can click button $\boxed{\texttt{next}}$ to command the simulator to step forward and compute 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 button $\boxed{\texttt{next}}$ is depressed, 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 figure 3. We explain briefly its components in the following:

- *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 [20].
- *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,



Pi	Xi	Process Transitions	
1	5	when !cd:3 mode[1]=2 and x[1]<26 may x[1]:	12
3	7	when ?cd mode[3]=3 may x[3]:=0;	1
3	8	when ?cd mode[3]=3 may x[3]:=0;mode[3]:=5	1
3	11	when ?cd mode[3]=4 and x[3]<52 may x[3]:=	1

```
2    11   when ?cd mode[2]=4 and x[2]<52 may x[2]:  1
```

```
(0)mode[1];8208a48;IC=1;[bus_collision]81f6ad8;
 (1)(0-x[1]);81f6ad8;IC=1;<=0:82000f0;
  (2)(x[1]-0);82000f0;IC=1;<26:8203d68;
   (3)mode[2];8203d68;IC=1;[sender_transm]8200ac8;
    (4)(x[2]-0);8200ac8;IC=1;<52:8201048;
     (5)(x[1]-x[2]);8201048;IC=1;<=0:8200f40;
      (6)(x[2]-x[1]);8200f40;IC=1;<26:8201078;
       (7)mode[3];8201078;IC=3;[sender_wait]8200300;[sender_transm]8203758;[sender_retry]8206e90;
        (8)mode[4];8200300;IC=1;[sender_transm]8200a98;
         (9)(x[4]-0);8200a98;IC=1;<26:81e8ad0;
          (10)(x[1]-x[4]);81e8ad0;IC=1;<=0:81e3230;
           (11)(x[4]-x[1]);81e3230;IC=1;<=0:TRUE;
        (8)(x[3]-0);8203758;IC=1;<26:82005a0;
         (9)(x[1]-x[3]);82005a0;IC=1;<=0:8203cd0;
          (10)(x[3]-x[1]);8203cd0;IC=1;<=0:82070b8;
```

```
GAME: 15/30
```

Figure 2: the GUI of RED 4.0

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 intersection 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, according to some metrics of coverage if any, in the generation of traces by repetitive invoking the RED symbolic simulation engine. Functional and code coverage can not only be computed for a trace and but also used to automatically direct the generation of traces. 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. Users can click button $\boxed{\texttt{next}}$ to step forward and button $\boxed{\texttt{previous}}$ to backtrack. 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.
- *GUI (graphical user-interface*: A user-friendly window for easy access to the power of formal verification.
- *RED symbolic TCTL model-checker*: With a single click on button $\boxed{\texttt{model-check}}$, the high performance backward analysis power of RED can be 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" 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 button $\boxed{\texttt{next}}$, $\boxed{\texttt{previous}}$, and selection of process transitions from the PT-frame, users have total control on how to select process transitions to make LG-transitions in the

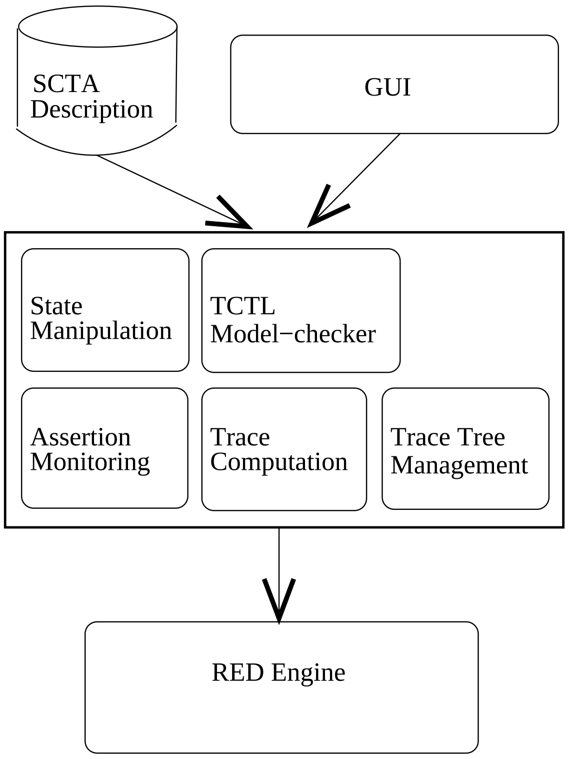


Figure 3: the architecture of RED model-checker/simulator

construction of the next-step state predicates along the current trace.

- *random steps:* By clicking button `random`, the simulator will randomly choose an LG-transition in each step. Users can command the autonomous execution of a given number of random steps.
- *game-based policy:* This option can be activated by clicking button `game-step`. 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[27], 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 option can be activated by clicking button `goal-step`. It 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 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 When to stop simulation ?

In RED 4.0, users can choose two options to estimate whether enough number of traces have been generated. Users can pull down the top menu item `Coverage` and choose either *code coverage* or *full functional coverage*. The current chosen coverage option is also displayed in the status line at the bottom of the window. In subsections 6.1 and 6.2, we first discuss how to estimate the code and full functional coverages. Then in subsection 6.3, we discuss how to use the coverage estimation to automatically generate traces.

6.1 Code coverage

We correspond the traditional *code coverage*[7] to the coverage of LG-transitions. That is, we think LG-transitions are equivalent to code lines as far as coverage is concerned. We argue for this correspondence because in state-transition systems, the LG-transitions act like program statements. Thus code coverage in our framework means how many LG-transitions have been tried in previous trace computation.

In our implementation, RED 4.0 will construct a BDD (Binary Decision Diagram [11]) for the legitimate combinations of process transitions for LG-transitions. This BDD is called `XSync` in our implementation. The maximum of LG-transitions triggerable in all traces is equal to the number of root-terminal paths in `XSync`. After LG-transitions have been selected (with plain interaction, random steps, ..., etc.), we will save these combinations in another BDD called `CCoverage`. The percentage of code coverage can be computed as the ratio of the number of LG-transitions in `CCoverage` over that of `XSync`. When option code coverage is selected, the ratio will be displayed in the status bar at the bottom of RED 4.0 window.

6.2 Detection of full functional coverage w.r.t. assertions

We resort to the development of various abstraction techniques to determine when enough number of traces have been generated to cover the functions users want to monitor. We need two BDD-like data structure `FFC` (for full functional coverage) and `FCoverage`. After the generation of a state-predicate, say η, we will first compute its abstract image, say $g(\eta)$ and calculate

$$\texttt{FFC} := \texttt{FFC} \vee g(\eta) \wedge \texttt{XSync}$$

In this way, `FFC` will always record the set of firable LG-transitions with their corresponding state-space reachable so far (at the current step).

Then from $g(\eta) \wedge \texttt{XSync}$, we will only choose some set, say representable with BDD η', of LG-transitions from it, according to various optional policies or user-guidance, to construct the state-predicate for the next-step. Be fore the computation of the next-step state-predicate, we shall calculate

$$\texttt{FCoverage} := \texttt{FCoverage} \vee \eta'.$$

In this way, `FCoverage` will always record the set of fired LG-transitions with their corresponding state-space reachable so far (at the current step). The percentage of code coverage can be computed as the ratio of the number of LG-transitions in `FCoverage` over that of `FFC`.

The options for the abstract image functions are:

- *Zone-coverage*: This is the case where no abstraction is performed. The next-step state-predicate is directly unioned with `FCoverage`. This provides the ultimate and costly precision but makes the simulation session somewhat equivalent to forward reachability analysis.
- *Game-coverage*: As in the paragraph of game-based policy in page 6, we view the behavior of the target system as a game process and players, opponents can be identified. The game-coverage abstract image function will eliminate the state information of the opponents from its argument.
- *Game-discrete-coverage*: This abstract image function will eliminate all clock constraints for the opponents in the state-predicate.
- *Game-magnitude-coverage*: 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.

Then with different implementation of the abstract image functions, we have an array of functional coverage measures with different precisions. Obviously, the zone-coverage image function is the most precise, while the game-coverage is the least precise. To achieve full zone-coverage can be very costly, in fact, of the same complexity as forward reachability analysis. On the other hand, the coverage-scheme with abstract image functions are less precise but can be less expensive to achieve their respective full coverage than zone-coverage.

6.3 Directed trace generation with coverage-based policy

If a trace will not increment the percentage of coverage, whether code or functional, it will be better if we can skip the trace since it does not increase users' confidence of their design. In addition to the trace-generation policies in section 5, we also have the option based on coverage. For example, in the case of functional coverage, after the generation of next-step state-predicates, say D, only the part of D which are not already included in `FCoverage` will be used as the state-predicate for new current state. This is technically achieved by the following three statements.

```
1) compute the abstract image $D'$ of $D$;
2) assign $\texttt{FCoverage} := \texttt{FCoverage} \cup D$;
3) compute the new current state-predicate
   as $D \cap \neg\texttt{FCoverage}$.
```

In the choice of random traces, we can also opt for the choice of traces which leads to the greatest increment in functional coverage based on greedy method.

7 Manipulation of current state-predicate

Our simulator allows for the modification of the current state-predicate before proceeding to the next-step. The following four buttons in figure 2 can be used to manipulate the current state-predicate and control the "width" of traces.

- `assign`: clicked to assign a new value to a state-variable. This can be used to change the behavior of the systems and insert faults.
- `eliminate`: clicked to 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`: By clicking this button, 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`: By clicking this button, users can choose to apply one of the three abstract image functions in subsection 6.2 to the current state-predicate. This is also equivalent to broadening the width of the trace.

Note that these four buttons 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-coverage abstract image function to $x_1 \geq 4 \wedge x_2 \geq 3 \wedge (x_1 - x_2 \leq -2 \vee x_2 - x_1 \leq -1)$, we get $x_1 \geq 4 \wedge x_2 \geq 3$ 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 the manipulations.

In addition, by clicking button `break`, the tool will prompt for the process transitions to set breakpoints. Then whenever at a current state-predicate the corresponding process transitions can be triggered, the simulator will notify the users. This is also an indispensible function in debugging.

8 Experiments on Bluetooth baseband protocol

In the following, we first give a brief introduction to the Bluetooth baseband protocol[23]. Then we present our model of baseband protocol in SCTA 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 8.5 respectively, and show how to quickly find the bugs with symbolic traces of Red 4.0. In subsections 8.3 and 8.4, we also demonstrate how to generate traces to observe system behaviors step by step and to gain enough confidence with options of coverage. Finally, in subsection 8.6, we use the bug-free model to report the performance in full verification of the Baseband protocol with a single click on button `model-check`.

8.1 Bluetooth baseband protocol

Bluetooth is a specification for wireless communication protocols[23]. 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

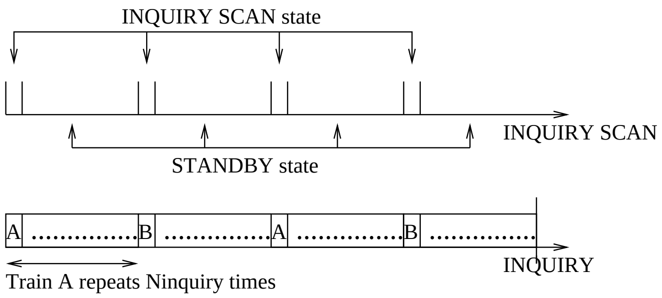


Figure 4: mode sequences of processes INQUIRY and INQUIRY SCAN in baseband protocol

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 [23];

8.2 The system model

In this subsection, we will introduce our system model briefly. For more details, the timed automata are shown in Appendix B. For convenience, we have labeled the process transitions with numbers.

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 and INQUIRY SCAN processes, respectively. 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. After three train switches, the process goes to TIMEOUT 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 click button $\boxed{\texttt{game-step}}$ to automatically trace to our goal states.

We use the step sequence shown in the second row of table 1 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 figure 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 `inqscanTimer_` < `TwInqScan_c` (`TwInqScan_c` is a macro constant defining the scan window size), the printer stays in mode INQUIRY_SCAN. At the time when `inqscanTimer_` = `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

$$\begin{array}{rl} & \texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \\ \wedge & \texttt{mode\_scan} = \text{INQUIRY\_SCAN} \end{array}$$

and

$$\begin{array}{rl} & \texttt{inqscanTimer\_} \geq \texttt{TwInqScan\_c} \\ \wedge & \texttt{mode\_scan} = \text{STANDBY} \end{array}$$

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

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

We want to use our model-checker/simulator to gain confidence that this risk will never happen.

When the notebook starts to inquiry, the printer may be in mode INQUIRY_SCAN or mode STANDBY. With traditional simulation[8, 14, 18], 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.

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

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 clicking $\boxed{\texttt{next}}$ to execute the first five steps in the sequence of table 1, 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 figures in figure 5 in appendix B) of process INQUIRY SCAN. With RED 4.0, we can simulate all these possibilities in a single step.

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 figure 5). According to Bluetooth specification[23], 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 figure 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 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 first click on button $\boxed{\texttt{restrict}}$ and keying state-predicate `inqscanTimer_ = equals TwInqScan_c` to restrict the current state-predicate.

Now the new current state-predicate satisfies

$$\begin{array}{rl} & \texttt{inqscanTimer\_} < \texttt{TwInqScan\_c} \\ \wedge & \texttt{mode\_scan} = \text{INQUIRY\_SCAN} \end{array}$$

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 do

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-step`
code coverage	1/23	2/23	3/23	6/23	10/23	10/23	10/23	21/23
fun. coverage estimation	56/112	728/1568	1196/1736	1352/1892	2276/3086	2276/3086	2276/3086	2643/4424

I: process INQUIRY; IS: process INQUIRY_SCAN; (p, x): process p executing process transition x.
c/d: c is the current coverage, d is the whole coverage.

Table 1: the step-by-step coverages during simulation

this by clicking button `assign` and changes the value of `mode_scan` from INQUIRY_SCAN to STANDBY. Then we click button `game-step` to generate trace 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 give users confidence that the both parties indeed can hop to the same frequency.

8.4 Coverage for confidence

In the 3rd and 4th rows of table 1, we respectively show the code and functional coverage percentages of the simulation traces used in subsection 8.3. These coverage percentages will be displayed at the bottom of our window when appropriate options for coverage are selected. The percentage of code coverage grows monnotonously. There are 23 LG-transitions in total. At the end of simulation at step 8, only two LG-transitions have not been covered by the symbolic trace. So in the end of the simulation session of last subsection, we have achieved 21/23 in code coverage.

On the other hand, the percentage of functional coverage estimation may increase and decrease during the simulation. This is due to that the number of paths in `FCoverage` and `FFC` may both increase in a step. Nonetheless it still gives us a strong hint about the progress of our simulation. In the end of the simulation session of last subsection, we have achieved 2643/4424 in functional coverage. The number is lower than the number for code coverage simply because functional coverage is more precise.

8.5 Fast debugging with `goal-step`

Here we show how to find bugs in our Baseband model with our `goal-step`. 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 `goal-step`. 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 full 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-step` is more efficient in debugging the model as compared with `random` and full verification.

8.6 Full verification with `model-check`

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 with button `model-check` 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 symbolic simulator based on BDD-like data-structure with GUI for dense-time concurrent systems. RED 4.0 can generate symbolic traces with various policy, compute the coverage, and manipulate the state-predicate. By properly control 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. First, we hope to derive new HD-estimation functions used in directed trace generation, and support customized automatic trace generation policy. These would help users finding bugs with fewer simulation traces. Second, the improvement on the precision of coverage estimation is also an important issue in our future work. Finally, we plan to make our GUI more friendly so that users can have easy access to the power of formal verification.

References

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

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

[3] R. Alur, D.L. Dill. Automata for modelling real-time systems. ICALP' 1990, LNCS 443, Springer-Verlag, pp.322-335.

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

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

[6] M. Bozga, C. Daws. O. Maler. Kronos: A model-checking tool for real-time systems. 10th CAV, June/July 1998, LNCS 1427, Springer-Verlag.

[7] Bening, L. and Foster, H., i. Principles of Verifiable RTL Design, a Functional Coding Style Supporting Verification Processes in Verilog,li 2nd ed., Kluwer Academic Publishers, 2001.

[8] M. Brockmeyer, C. Heitmeyer, F. Jahanian, B. Labaw. A Flexible, Extensible Simulation Environment for Testing Real-Time, IEEE, 1997.

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

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

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

[12] E. Clarke, 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.

[13] E. Clarke, O. Grumberg, M. Minea, D. Peled. State-Space Reduction using Partial-Ordering Techniques, STTT 2(3), 1999, pp.279-287.

[14] 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.

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

[16] C. Daws, A. Olivero, S. Tripakis, S. Yovine. The tool KRONOS. The 3rd Hybrid Systems, 1996, LNCS 1066, Springer-Verlag.

[17] E.A. Emerson, A.P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM TOPLAS, Vol. **19**, Nr. 4, July 1997, pp. 617-638.

[18] S.J. Garland, N.A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Technical Report MIT/LCS/TR.

[19] 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.

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

[21] C.A.R. Hoare. Communicating Sequential Processes, Prentice Hall, 1985.

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

[23] J. Haartsen. Bluetooth Baseband Specification, version 1.0. http://www.bluetooth.com/

[24] K.G. Larsen, F. Larsson, P. Pettersson, Y. Wang. Efficient Verification of Real-Time Systems: Compact Data-Structure and State-Space Reduction. IEEE RTSS, 1998.

[25] N. Lynch, M.R. Tuttle. An introduction to Input/Output automata. CWI-Quarterly, 2(3):219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[26] P. Pettersson, K.G. Larsen, UPPAAL2k. in Bulletin of the European Association for Theoretical Computer Science, volume 70, pages 40-44, 2000.

[27] R.S. Pressman. Software Engineering, A Practitioner's Approach. McGraw-Hill, 1982.

[28] C.-J.H. Seger, R.E. Brant Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. Formal Methods in System Designs, Vol. 6, No. 2, pp. 147-189, Mar. 1995.

[29] F. Wang. Efficient Data-Structure for Fully Symbolic Verification of Real-Time Software Systems. TACAS'2000, March, Berlin, Germany. in LNCS 1785, Springer-Verlag.

[30] F. Wang. Region Encoding Diagram for Fully Symbolic Verification of Real-Time Systems. the 24th COMPSAC, Oct. 2000, Taipei, Taiwan, ROC, IEEE press.

[31] 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.

[32] F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram, to appear in Proceedings of FORTE, August 2001, Cheju Island, Korea.

[33] F. Wang. Symmetric Model-Checking of Concurrent Timed Automata with Clock-Restriction Diagram. RTCSA'2002.

[34] F. Wang. Efficient Verification of Timed Automata with BDD-like Data-Structures. Technical Report, IIS, Academia Sinica, 2002.

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

[36] F. Wang, P.-A. Hsiung. Efficient and User-Friendly Verification. IEEE Transactions on Computers, Jan. 2002.

[37] F. Wang, C.-T. Lo. Procedure-Level Verification of Real-Time Concurrent Systems. International Journal of Time-Critical Computing Systems **16**, 81-114 (1999).

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

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

APPENDICES

A Definition of SCTA

A *SCTA (Synchronized Concurrent Timed Automaton* is a set of finite-state automata, called *process automata*, 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.

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 $\texttt{mode}$ is a special auxiliary variable, $q \in Q$, $x, x' \in X \cup \{0\}$, "$\sim$" is one of $\leq, <, =, >, \geq$, and c is an integer constant.

Definition 1 <u>**process automata**</u> 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
<u>*SCTA (Synchronized Concurrent Timed Automata)*</u> An SCTA 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 $1 \leq p \leq m$, $A_p = \langle X_p, E, Q_p, I_p, \mu_p, T_p, \lambda_p, \tau_p, \pi_p\rangle$ 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 <u>**states**</u> Suppose we are given an SCTA $S = \langle E, A_1, A_2, \ldots, A_m\rangle$ such that for each $1 \leq p \leq m$, $A_p = \langle X_p, E, Q_p, I_p, \mu_p, T_p, \lambda_p, \tau_p, \pi_p\rangle$. A state ν of S is a valuation of $\bigcup_{1\leq p\leq m}(X_p \cup \{\texttt{mode}_p\})$ such that

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

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, $1 \leq p \leq m$, 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, $\sum_{1\leq p\leq m;\Phi(p)\neq\perp} \lambda(e, \Phi(p)) = 0$.

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 $1 \leq p \leq m$ and $\Phi(p) \neq\perp$, the following two conditions must be true.

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

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 $1 \leq i < k$, 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 an SCTA $S = \langle E, A_1, A_2, \ldots, A_m\rangle$ such that for each $1 \leq p \leq m$, $A_p = \langle X_p, E, Q_p, I_p, \mu_p, T_p, \lambda_p, \tau_p, \pi_p\rangle$. 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 $\nu_0 \models I$ and $t_0 t_1 \ldots t_k \ldots\ldots$ is a monotonically increasing real-number (time) divergent sequence, and for all $k \geq 0$,

- for all $t \in [0, t_{k+1} - t_k]$, $\nu_k + t \models \bigwedge_{1\leq p\leq m} \mu(\nu_k(\texttt{mode}_p))$; and
- either
  - $\nu_k(\texttt{mode}_p) = \nu_{k+1}(\texttt{mode}_p)$ and $\nu_k + (t_{k+1} - t_k) = \nu_{k+1}$; or
  - there exists a race-free IST-plan Φ such that for all $1 \leq p \leq m$,
    * either $\nu_k(\texttt{mode}_p) = \nu_{k+1}(\texttt{mode}_p)$ or $(\nu_k(\texttt{mode}_p), \nu_{k+1}(\texttt{mode}_p)) \in T_p$ and
    * $\nu_k + (t_{k+1} - t_k) \models \bigwedge_{1\leq p\leq m;\Phi(p)\neq\perp} \tau_p(\nu_k(\texttt{mode}_p), \nu_{k+1}(\texttt{mode}_p))$ and
    * $(\nu_k + (t_{k+1} - t_k))\texttt{concat}_{1\leq p\leq m;\Phi(p)\neq\perp}\pi_p(\nu_k(\texttt{mode}_p), \nu_{k+1}(\texttt{mode}_p)) = \nu_{k+1}$. Here $\texttt{concat}(\gamma_1, \ldots, \gamma_h)$ 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. Due to page-limit, we here 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 automata 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 $k \geq 0$, and for all $t \in [0, t_{k+1} - t_k]$, $\nu_k + t \models \eta$, i.e., the safety requirement is guaranteed.

B Model of Bluetooth baseband protocol

Figures in the next two pages.

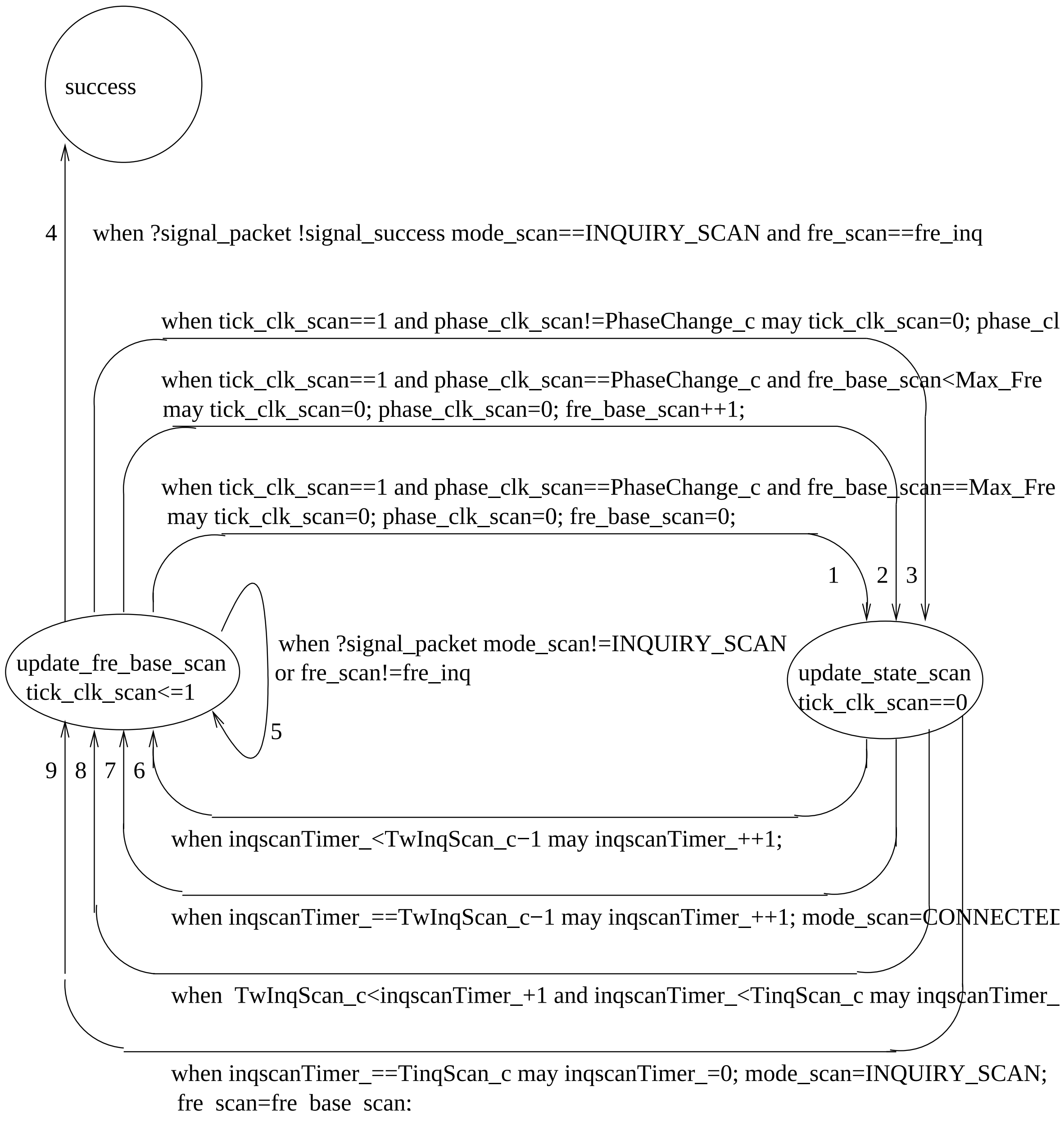

success
4
when ?signal_packet !signal_success mode_scan==INQUIRY_SCAN and fre_scan==fre_inq
when tick_clk_scan==1 and phase_clk_scan!=PhaseChange_c may tick_clk_scan=0; phase_cl
when tick_clk_scan==1 and phase_clk_scan==PhaseChange_c and fre_base_scan<Max_Fre
may tick_clk_scan=0; phase_clk_scan=0; fre_base_scan++1;
when tick_clk_scan==1 and phase_clk_scan==PhaseChange_c and fre_base_scan==Max_Fre
may tick_clk_scan=0; phase_clk_scan=0; fre_base_scan=0;
1
2
3
update_fre_base_scan
tick_clk_scan<=1
when ?signal_packet mode_scan!=INQUIRY_SCAN
or fre_scan!=fre_inq
5
update_state_scan
tick_clk_scan==0
9
8
7
6
when inqscanTimer_<TwInqScan_c−1 may inqscanTimer_++1;
when inqscanTimer_==TwInqScan_c−1 may inqscanTimer_++1; mode_scan=CONNECTEI
when TwInqScan_c<inqscanTimer_+1 and inqscanTimer_<TinqScan_c may inqscanTimer_
when inqscanTimer_==TinqScan_c may inqscanTimer_=0; mode_scan=INQUIRY_SCAN;
fre scan=fre base scan:

Figure 5: INQUIRY SCAN

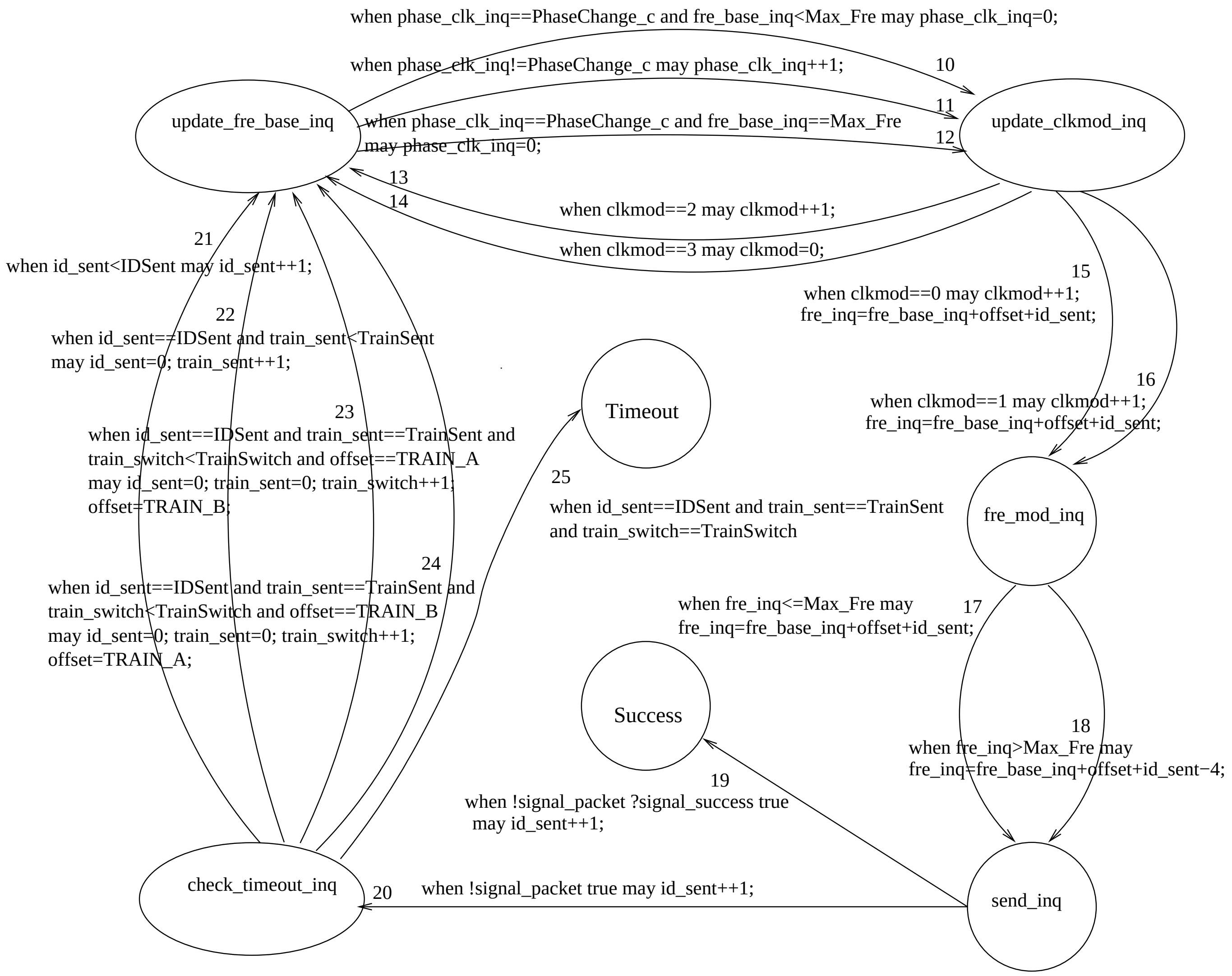

when phase_clk_inq==PhaseChange_c and fre_base_inq<Max_Fre may phase_clk_inq=0;
10
when phase_clk_inq!=PhaseChange_c may phase_clk_inq++1;
11
update_fre_base_inq
when phase_clk_inq==PhaseChange_c and fre_base_inq==Max_Fre may phase_clk_inq=0;
12
update_clkmod_inq
13
14
when clkmod==2 may clkmod++1;
when clkmod==3 may clkmod=0;
21
when id_sent<IDSent may id_sent++1;
15
when clkmod==0 may clkmod++1; fre_inq=fre_base_inq+offset+id_sent;
22
when id_sent==IDSent and train_sent<TrainSent may id_sent=0; train_sent++1;
Timeout
16
when clkmod==1 may clkmod++1; fre_inq=fre_base_inq+offset+id_sent;
23
when id_sent==IDSent and train_sent==TrainSent and train_switch<TrainSwitch and offset==TRAIN_A may id_sent=0; train_sent=0; train_switch++1; offset=TRAIN_B;
25
when id_sent==IDSent and train_sent==TrainSent and train_switch==TrainSwitch
fre_mod_inq
24
when id_sent==IDSent and train_sent==TrainSent and train_switch<TrainSwitch and offset==TRAIN_B may id_sent=0; train_sent=0; train_switch++1; offset=TRAIN_A;
when fre_inq<=Max_Fre may fre_inq=fre_base_inq+offset+id_sent;
17
Success
18
when fre_inq>Max_Fre may fre_inq=fre_base_inq+offset+id_sent−4;
19
when !signal_packet ?signal_success true may id_sent++1;
check_timeout_inq
20
when !signal_packet true may id_sent++1;
send_inq

Figure 6: INQUIRY