OVL Assertion-Checking of Embedded Softwares with Dense-Time Semantics *

Farn Wang and Fang Yu
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://val.iis.sinica.edu.tw

Abstract

OVL (Open Verification Library) is designed to become a standard assertion language of the EDA (Electronic Design Automation) industry and has been adopted by many companies. With OVL, verification process can blended seamlessly into the development cycles of complex systems. We investigate how to use OVL assertions for the verification of dense-time concurrent systems. We have designed a C-like language, called TC (timed C), for the description of real-time system with OVL assertions between code lines. We explain how to translate TC programs into optimized timed automata, how to translate OVL assertions into TCTL (Timed Computation-Tree Logic) formulae, and how to analyze assertions when not satisfied. The idea is realized in our translator RG (RED Generator).

In addition, we have developed several new verification techniques to take advantage of the information coming with OVL assertions for better verification performance. The new techniques have been incorporated in our high-performance TCTL model-checker RED 4.0. To demonstrate how our techniques can be used in industry projects, we report our experiments with the L2CAP (Logic Link Control and Adaptation Layer Protocol) of BlueTooth specification.

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

1 Introduction

In the last decade, many formal verification tools with proprietary (i.e., commercial or tool-specific) assertion languages have emerged in the industry[3, 14, 20, 24, 25, 31]. But, as Forster discussed, the lack of standards in assertion languages not only can frustrate engineers but also can create significant chaos and damage to the healthy progress of verification technology[7]. But what should a standard assertion language look like ? A good assertion language must blend seamlessly into the development cycles of system designs. In real-world projects, engineers naturally describe their systems in programming languages and insert comment lines to assert some intuitive properties between codes, such as preconditions or post conditions. If a verification tool asks engineers to rewrite their C-codes in automata descriptions or Petri net descriptions and to make up some assertions offline of the programming cycle, then the engineers will more likely be reluctant to accept the tool in fear of extra workload and deadline misses. Thus, providing a natural method to bridge this gap in the verification of real-time concurrent systems is one main goal in this paper.

OVL (Open Verification Library) [7, 27] is a new initiative in VLSI industry for unifying the many commercial EDA (Electronic Design Automation) tools, by providing a set of predefined specification modules instantiated as assertion monitors. It is supported by EDA industry companies and donated to Accellera (an electronic industry standards organization) in anticipation to make OVL an industry standard. With OVL, engineers can write assertions as comment lines in their HDL (Hardware Description Language [5, 30]) programs.

OVL was originally designed for the assertions of VLSI circuits, which are highly synchronous discrete-time systems. With the coming of multi-multimillion-gate SOC (System-on-a-Chip)[29] in the new century, clock skews may eventually invalidate the synchrony assumptions. Also, multiclock chips can also happen frequently in systems developed with *IPs (Intellectual Properties)* to simplify design patterns. But today's industry projects usually only use static timing analysis[26, 28] to guarantee real-time properties. Thus it will be of great interest if we can extend OVL assertions to dense-time model in formal verification. Such an extension will also allow embedded system engineers to take advantage of verification technology with minimum effort in their development cycles. And that is the first motivation of this research.

To blend seamlessly into the development cycles,

*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.

it is also important that system designs can be described in a format close to programming languages. In section 5, we define a new language, called *Timed C (TC)*, with C-like syntax and OVL assertions as comment lines. TC is designed for efficient mechanical translation from C-programs into input languages of our TCTL model-checker RED 4.0 for formal verification. The input to RED 4.0 consists of a *timed automata*[2] (with synchronization channels[22]) and a *TCTL (Timed Computation-Tree Logic)*[1] specification. In section 5, we discuss how to mechanically translate TC programs to optimized (for verification performance) timed automata with synchronizers.

In section 7, we present four types of OVL assertions and demonstrate how to translate these OVL assertions, with dense-time semantics, to TCTL formulae. In some cases, we have to create auxiliary processes and state-variables to monitor the satisfaction of OVL assertions. We have realized all these ideas in a translator, *RG (RED Generator)*, which translates TC programs into input format to RED[32, 33, 34, 35, 36, 37], a high-performance TCTL model-checker for timed automata.

The positions of OVL assertions in a program may also shed light on the possibility of verification performance enhancement. If an assertion is declared specifically in a process' program, usually it means that the assurance of the assertion is strongly linked to the behavior of this process. Then by carefully abstracting out state information of other processes, state-space representation can be significantly simplified and performance improvement in verification can be obtained. This intuition has led us to the design of several *localized abstraction* functions, which are explained in section 8. Unlike the previous work on approximate model-checking[40], our new abstraction technique is specially tailored to take advantage of the information hidden in OVL assertions. And our experiment with this new technique of localized abstract reduction indeed shows that performance improvement can be gained in verification with the information hidden in OVL assertions.

To demonstrate the usefulness of our techniques for real-world projects, in section 9, we have experimented to model and verify the *L2CAP (Logic Link Control and Adaptation Layer Protocol)* of BlueTooth specification[11]. BlueTooth, a wireless communication standard, has been widely adopted in industry. We model two devices, communicating with the L2CAP of BlueTooth, in TC and carry out experiments to verify various properties between the two devices. The experiments are by themselves important because of the wide acceptance and application of the protocol.

We have also extended the functionality of RED, including the capability of *deadlock detection* (reachability analysis of states from which no more transitions are possible). Moreover, since OVL assertions are written in between code lines, their dissatisfaction may provide valuable feedback for code debugging and direction to system refinement. When there are more than one assertions in a TC program and some of them are not satisfied, RED is also capable of identifying which assertions are not satisfied. It is also possible to use the counter-example generation capability of RED to better understand the system behavior and diagnose the design bugs.

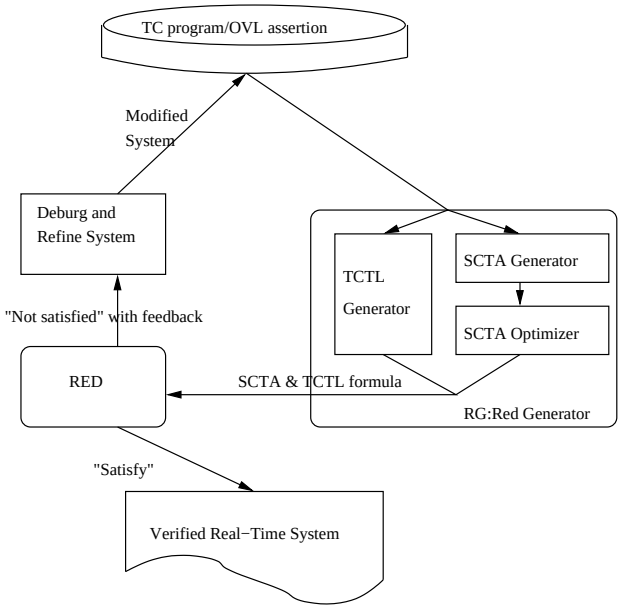

Figure 1: Software architecture

The remainder of this paper is organized as follows. Section 2 discusses the verification tool framework. Section 3 and 4 introduce the input language to RED 4.0, i.e., *synchronized concurrent timed automata (SCTA)* and TCTL. Section 5 discusses the language of TC(Timed C) and algorithms for translating TC constructs into optimized SCTA subgraphs. Section 6 describes OVL assertions. Section 7 discusses how to translate OVL assertions into TCTL formulae. Section 8 introduces our localized abstraction technique specially tailored for performance verification of OVL assertions. Section 9 reports our verification experiments with L2CAP. Section 10 discusses how to implement deadlock detection in RED and how the technique can be useful in practice. Section 11 concludes the paper with remarks on future plan of the work.

Formal semantics of SCTA and TCTL can be found in appendices A and B respectively. An example of TC program with OVL assertion and its corresponding optimized SCTA can be found in appendices C and D respectively.

2 Verification tool framework

The software architecture of our verification framework is shown in figure 1. On the top, users describe the

system designs in our C-like language, TC, with OVL assertions as comments between code lines. After parsing and analyzing a TC program, our translator RG generates a file, in the format of input language to our TCTL model-checker RED, with an SCTA and a TCTL formulus. An SCTA includes a set of *process automata* communicating with each other with binary *synchronizers*[22] and global variables. The global automaton for the whole system is the Cartesian product of the process automata. Some process automata describe the system behaviors while others monitor the satisfaction of the OVL assertions.

The TCTL formulus is derived from the OVL assertions. If there are more than one assertions, then their corresponding TCTL formulae conjunct together to construct the final TCTL formulus.

We use two phases in the generation of SCTAs. The first phase generates an SCTA, which is further optimized in the second phase. The optimization program used in the second phase can also be used independently to help users of RED in optimizing their system descriptions.

After the SCTA and TCTL-formulus are generated, users may feed them to RED[32, 33, 34, 35, 36], our TCTL model-checker. Our RED is implemented with the new BDD-like data-structure of CRD (Clock-Restriction Diagram)[34, 35, 36, 37]. If RED says that the SCTA does not satisfy the TCTL formulus, RED can identify among the many OVL assertions which ones are not satisfied and may generate counter-example traces in some situations. Users can use this information as feedback to fix bugs and re-execute this verification cycle. On the other hand, if RED says the SCTA satisfies the TCTL formulus, the correctness of the system design is formally confirmed.

3 Synchronized concurrent timed automata (SCTA)

We use the widely accepted model of *timed automata*[2] with synchronizers[22]. A *timed automaton* is a finite-state automaton equipped with a finite set of clocks which can hold nonnegative real-values. At any moment, the timed automaton can stay in only one *mode* (or *control location*). In its operation, one of the transitions can fire when the corresponding triggering condition is satisfied. Upon firing, 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.

In our input language, users can describe the timed automata as a *synchronized concurrent timed automata (SCTA)*. 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. Each transition (arc) in the process automata is called a *process transition*.

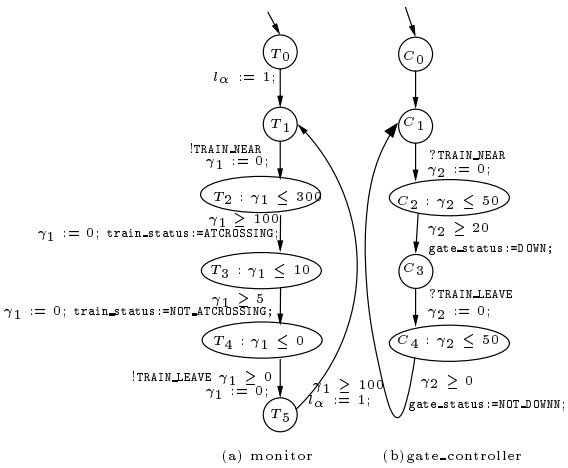

Figure 2: process automata of the railroad crossing system

In figure 2, we have drawn two process automata, in a railroad crossing system. One process is for train-monitor and one for the gate-controller. The monitor uses a local clock γ_1 while the controller uses γ_2. In each mode, we may label an invariance condition (e.g., $\gamma_1 \leq 300$). Along each process transition, we may label synchronization symbols (e.g. `!TRAIN_NEAR`), a triggering condition (e.g., $\gamma_1 \geq 100$), and assignment statements (e.g., $\gamma_1 := 0$;). When the monitor detects that a train is approaching the crossing, it sends out a `!TRAIN_NEAR` signal to the controller. On receiving the signal, the train will reach the crossing in 100 to 300 time units while the gate will be lowered down in 20 to 50 time units.

A process transition may not represent a *legitimate global transition (LG-transition)*. Only LG-transitions can be executed. Symbols `TRAIN_NEAR` and `TRAIN_LEAVE`, on the arcs, represent channels for synchronizations. Synchronization channels 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, `!TRAIN_NEAR` means a sending event through channel `TRAIN_NEAR` while `?TRAIN_NEAR` 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. For example, in figure 2, process transitions $T_1 \longrightarrow T_2$ and $C_1 \longrightarrow C_2$ can combine to be an LG-transition while $T_1 \longrightarrow T_2$ and $C_2 \longrightarrow C_3$ cannot. Also process transition $T_2 \longrightarrow T_3$ by itself can

constitute an LG-transition since no synchronization is involved. The formal semantics of SCTA is left in appendix A.

4 TCTL (Timed CTL)

TCTL (Timed Computation-Tree Logic)[1] is a branching-time temporal logic for the specification of dense-time systems. An interval $\mathcal{I}$ specifies a continuous time segment and is denoted as the pair of (open) starting time and (open) stopping time like $(c,d), [c,d), [c,d], (c,d]$ such that $c \in \mathcal{N}$, $d \in \mathcal{N} \cup \{\infty\}$, and $c \leq d$. Open and closed intervals are denoted respectively with parentheses and square brackets.

Suppose we are given a set P of atomic propositions and a set X of clocks, a TCTL formula ϕ for S has the following syntax rules.

$$\phi \quad ::= \quad p \,|\, x_1 - x_2 \sim c \,|\, \phi_1 \vee \phi_2 \,|\, \neg\phi_1 \,|\, \exists\phi_1 \mathcal{U}_{\mathcal{I}} \phi_2 \,|\, \forall\phi_1 \mathcal{U}_{\mathcal{I}} \phi_2$$

Here $p \in P$, $x_1, x_2 \in X$, $c \in \mathcal{N}$, $\mathcal{I}$ is an interval, ϕ_1 and ϕ_2 are TCTL formulae, and $\mathcal{I}$ is an interval.

$\exists$ means "there exists a computation." $\forall$ means "for all computations." $\phi_1 \mathcal{U}_{\mathcal{I}} \phi_2$ means that along a computation, ϕ_1 is true until ϕ_2 becomes true and ϕ_2 happens at time in $\mathcal{I}$. For example, with a specification like

$$\forall \texttt{train\_status = ATCROSSING} \\ \mathcal{U}_{[0,10)} \texttt{train\_status = NOT\_ATCROSSING}$$

we require that for all computations, `train_status` becomes `NOT_ATCROSSING` in 10 time units.

Also we adopt the following standard shorthand : *true* for $\neg$*false*, $\phi_1 \wedge \phi_2$ for $\neg((\neg\phi_1) \vee (\neg\phi_2))$, $\phi_1 \rightarrow \phi_2$ for $(\neg\phi_1) \vee \phi_2$, $\exists\Diamond_{\mathcal{I}}\phi_1$ for $\exists true\, \mathcal{U}_{\mathcal{I}} \phi_1$, $\forall\Box_{\mathcal{I}}\phi_1$ for $\neg\exists\Diamond_{\mathcal{I}}\neg\phi_1$, $\forall\Diamond_{\mathcal{I}}\phi_1$ for $\forall true\, \mathcal{U}_{\mathcal{I}} \phi_1$, $\exists\Box_{\mathcal{I}}\phi_1$ for $\neg\forall\Diamond_{\mathcal{I}}\neg\phi_1$.

The formal semantics of TCTL formulae is left in appendix B.

5 Timed C

Engineers are trained to write programs in traditional programming languages, like C, C++, Verilog, ..., etc. Timed C (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, like sequences, while-loops, and switch-statements. It also provides syntax constructs to abstract unimportant details for mechanical translation to SCTA. Moreover, we have added new constructs to make it easy to describe event-driven behaviors, like timeouts.

5.1 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 the program, there are also two OVL assertions, which is explained in section 6.

5.2 Mechanical translation to SCTA

The real-time system model-checkers nowadays are based on mathematical models, like SCTA, Petri net, hybrid automata, ... [9, 10, 17, 23, 39, 34, 35, 40, 41]. To make the model-checking technology more attractive, it will be nice if we can mechanically translate C-programs to SCTAs. The language of TC (Timed C) serves as a middle language from C-programs to SCTAs.

The SCTA (generated from RG) for the TC-program in table 1 is exactly the one in figure 2.

For convenience, given a TC program construct $\mathbf{B}$, let $RG(\mathbf{B})$ be the subgraph in an SCTA representing the behavior of $\mathbf{B}$. The SCTA subgraphs of $RG(y = 3;)$ (an atomic assignment), $RG(\mathbf{B}_1\mathbf{B}_2)$ (a sequence), $RG(\texttt{while}\ (x < 3)\ \mathbf{B})$, and $RG(\texttt{switch}\ (y)\ \{\ldots\})$, are shown in figures 3(a), (b), (c), and (f) respectively. In construct `switch`$(y)\{\ldots\}$, y must be of type int. Constructs of if-else can be treated similarly as construct `switch`. Since we require the specification of the range of integer variable in their declaration in TC programs, constructs like if-else can be treated as special cases of constructs `switch`$(\ldots)\{\ldots\}$.

Note that in the subgraphs figure 3(c) and (f) for constructs `while` and `switch`, the test conditions for the cases are directly labeled on the incoming transitions as additional constraints. This means that the conditional statements in TC do not take time in our model. This assumption is important for efficient translation to SCTA, in which a transition with triggering condition testing and assignments is executed instantaneously. This assumption is suitable for embedded systems in which dedicated hardware is used for each process.

But 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. To put it 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

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

process monitor() {
  while (1) {
//assert_change #([0,20], 0) 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;
  }
}
```

Table 1: TC program for the modeling of railroad crossing system

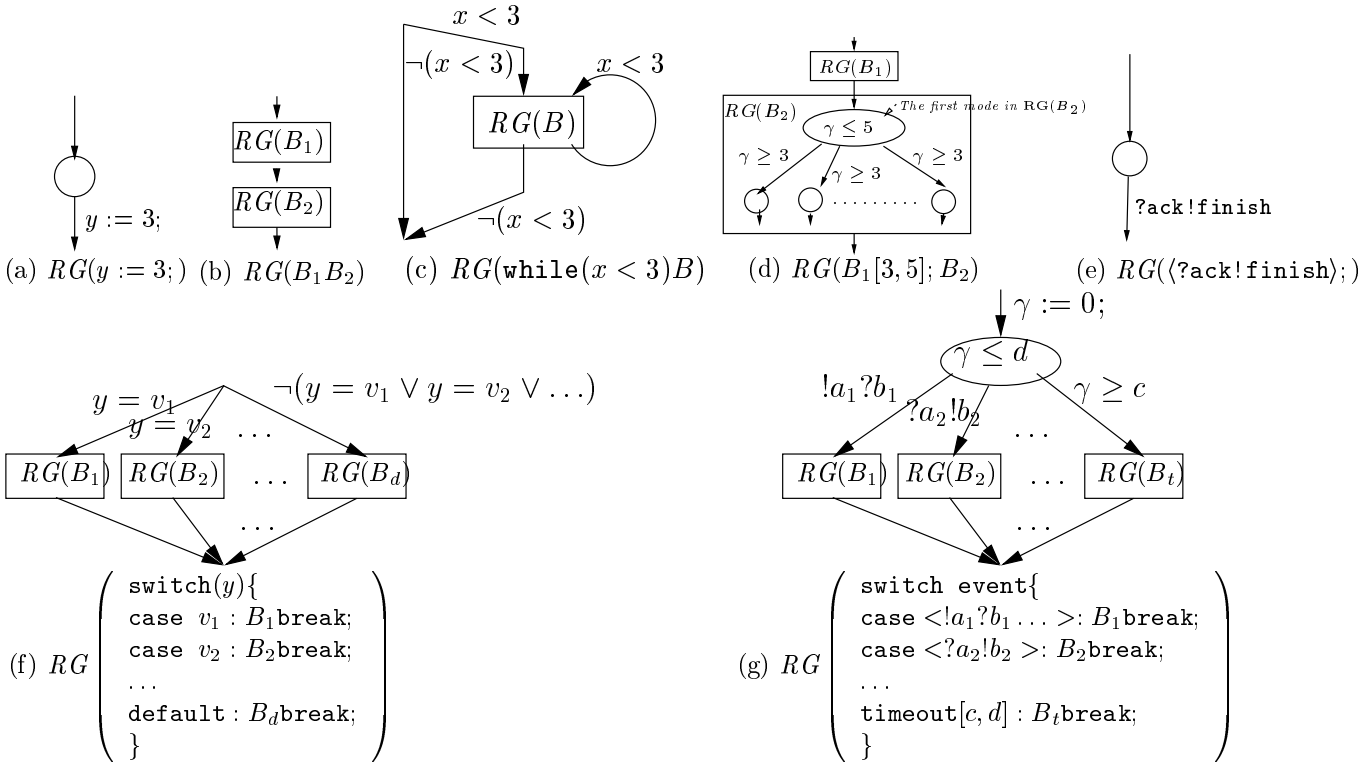

Figure 3: SCTA subgraphs for TC-program constructs

that $c \leq d$ 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 $[3, 5]$. The SCTA subgraph of $RG(\mathbf{B}_1[3, 5]; \mathbf{B}_2)$ is shown in figure 3(d). Note how we use an auxiliary system clock γ here to control the earliest starting time and deadline of the successor transition.

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[18] 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⟨ss1⟩ : B1break;
case⟨ss2⟩ : B2break;
...
timeout[c, d] : Btbreak;
}
```

to capture this kind of system behaviors. $ss_1, ss_2, \ldots$ are sequences of synchronization labels, like ?`receive`, !`send`, ... 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. The corresponding SCTA subgraph is drawn in figure 3(h). Note that the SCTA subgraph does have an auxiliary entry mode to enforce the timeout.

Finally e also allow programmers to use synchronizers in SCTA 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` >;" and $RG(<$ `?ack !finish` $>;)$ is shown in figure 3(e).

5.3 Optimization of SCTA

The first phase of RG generates an SCTA, which is clumsy to verify. The SCTA will have a lot of null states connecting together the SCTA subgraphs generated for various TC program constructs. Also, many operations on local variables may create unnecessary partial-ordering and irrelevant intermediate states, which can only waste resources in the verification tasks for the given OVL assertions. We borrowed the code optimization techniques from compiler research[4] for the optimization of SCTAs. After the optimization, the reachable state-space representation of the SCTA can be reduced and verification performance can be enhanced.

A simple but effective technique for locally improving the target code is *peephole optimization*, a method to improve the performance of the target program by examining a short sequence of target instructions and replacing these instructions (called the peephole) by a shorter or faster sequence [4]. We followed this idea and developed our SCTA Optimizer. The optimization techniques, which we employed, include

- *bypass of null transitions*: For easy mechanical translation, sometimes we generate null modes and transitions. These modes and transitions can be eliminated without changing the system behaviors.
- *compaction of intermediate local transitions*: In SCTA, we can declare local variables of type integer and pointers. The exact execution time (within an interval) of assignments to such local variables may not affect the behavior of peer processes. This kind of situation can be analyzed and we can compact these local actions into one process transition.
- *elimination of unreachable modes*: After the bypassing of many transitions, some modes in the original SCTA may no longer be connected to the initial mode in the SCTA graph. We can simply ignore such modes.
- *elimination of intermediate temporary variables*: In the evaluation of complex expressions, sometimes we have to declare intermediate temporary state-variables to store the intermediate results, like the sum of an addition inside a multiplication. By properly analyzing the structure of the arithmetic expressions, we can avoid the usage of some intermediate temporary variables.

Because of the page-limit, we omit the details of our implementation here. But we have carried out experiment on the L2CAP used in section 9. The TC program with some OVL assertions can be found in appendix C. The experiment reported in section 9 shows dramatic improvement in verification performance after the optimization.

6 OVL assertions

We here demonstrate how to translate the following four types of OVL assertions to TCTL formulae for model-checking with RED.

$$//\texttt{assert\_always}(\phi)$$
$$//\texttt{assert\_never}(\phi)$$
$$//\texttt{assert\_change}\#(\mathcal{I}, f)ID(\phi_1, \phi_2)$$
$$//\texttt{assert\_time}\#(\mathcal{I}, f)ID(\phi_1, \phi_2)$$

Here ϕ, ϕ_1, ϕ_2 are Boolean predicates on variable values. $\mathcal{I}$ is an interval (as in section 4). 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, $//\texttt{assert\_always}(\phi)$ and $//\texttt{assert\_never}(\phi)$ specify some properties at the current state. The first type

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

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 $f = 0$,

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

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.

When $f = 1$, assertion (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.

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[7]. 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 (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, WHENEVER ϕ_1 is true, ϕ_2 must not change value at any time in $\mathcal{I}$. Option $f = 0$ also means that this assertion will be claimed once and for all.

Similarly, when $f = 1$, assertion (2) 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 $\mathcal{I}$. Option $f = 1$ means that whenever this assertion is encountered, it will only be used once (when ϕ_1 is true) and then discarded.

7 From assertions to TCTL

Suppose we have n assertions $\alpha_1, \ldots, \alpha_n$. For each assertion α, we need a binary flag b_α. Then we label the modes of the automata with $b_{\alpha_1}, \ldots, b_{\alpha_n}$ to denote the scope within which the respective assertions are honored. For example, in the TC-program in table 1, there are two assertions. Suppose the `assert_change` assertion on the top is α_1 and the `assert_always` assertion in the middle is α_2. The SCTA of this TC-program is shown in figure 2. Then b_{α_1} is only labeled at mode T_1 while b_{α_2} is only labeled at mode T_3.

An assertion like α : $//\texttt{assert\_always}(\phi)$ is translated to the TCTL formulus, denoted as $TCTL(\alpha)$,

$$\forall\Box((\bigvee_{(q \text{ labeled with } b_\alpha)} q) \rightarrow \phi).$$

Here "$\bigvee_{(q \text{ labeled with } b_\alpha)} q$" is a predicate, which we generate to signal when assertion α must be satisfied.

For α : $//\texttt{assert\_never}(\phi)$, $TCTL(\alpha)$ is

$$\forall\Box((\bigvee_{(q \text{ labeled with } b_\alpha)} q) \rightarrow \neg\phi).$$

For each `assert_time` or `assert_change` α with unique name ID, we need to use auxiliary variables, auxiliary actions, and sometimes auxiliary processes to monitor their satisfaction. We need an auxiliary Boolean state variable l_α to monitor either

- when ϕ_1 has become true with option $f = 0$; or
- when ϕ_1 has become true for the first time with option $f = 1$.

For example, in figure 2, l_{α_1} is initially false and set to true at every process transition to mode T_1. l_{α_1} is never reset to false with option $f = 0$. (Details are discussed in the following.)

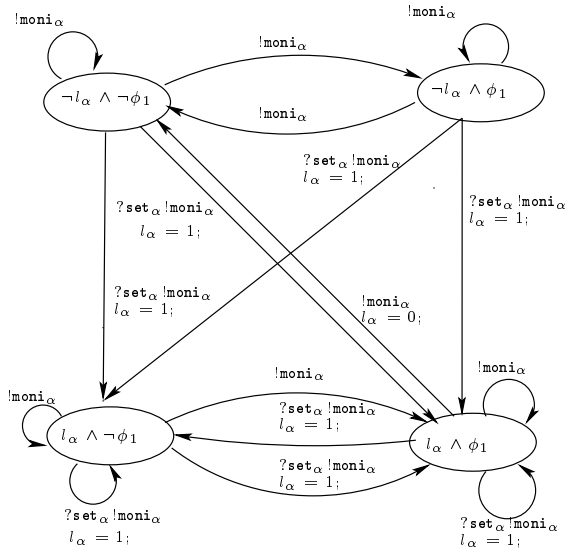

Figure 4: Auxiliary monitor process with option $f = 1$

For α : $//\texttt{assert\_change}\#(\mathcal{I}, f)ID(\phi_1, \phi_2)$, no matter whether $f = 0$ or $f = 1$, $TCTL(\alpha)$ is

$$\forall\Box\left(\begin{array}{l}(\bigvee_{(q \text{ labeled with } b_\alpha)} q)\\ \rightarrow \forall\Box\left(\begin{array}{l}(l_\alpha \wedge \phi_1)\\ \rightarrow ((\forall\neg\phi_2\mathcal{U}_\mathcal{I}\phi_2) \vee (\forall\phi_2\mathcal{U}_\mathcal{I}\neg\phi_2))\end{array}\right)\end{array}\right)$$

Formulus $\forall\neg\phi_2\mathcal{U}_\mathcal{I}\phi_2$ captures the trace along which ϕ_2 changes from false to true at time in $\mathcal{I}$ while $\forall\phi_2\mathcal{U}_\mathcal{I}\neg\phi_2$ captures the trace along which ϕ_2 changes from true to false at time in $\mathcal{I}$.

For α : $//\texttt{assert\_time}\#(\mathcal{I}, f)ID(\phi_1, \phi_2)$, no matter whether $f = 0$ or $f = 1$, $TCTL(\alpha)$ is the same

$$\forall\Box\left(\begin{array}{l}(\bigvee_{(q \text{ labeled with } b_\alpha)} q)\\ \quad \rightarrow \forall\Box((l_\alpha \wedge \phi_1) \rightarrow ((\forall\Box_\mathcal{I}\neg\phi_2) \vee (\forall\Box_\mathcal{I}\phi_2)))\end{array}\right)$$

Formulus $\forall\Box_\mathcal{I}\neg\phi_2$ captures the trace along which ϕ_2 is maintained false within $\mathcal{I}$ while $\forall\Box_\mathcal{I}\phi_2$ is maintained true within $\mathcal{I}$.

When the assertions of type either `assert_change` or `assert_time` is written with option $f = 0$, we need the following minor modification to the process automata input to RED: *for every incoming transition to modes labeled with b_α, we need to label it with the auxiliary assignment $l_\alpha := 1$; to indicate that the scope of assertion α is entered.* This can be seen from label $l_{\alpha_1} := 1$; on the incoming transitions to mode T_1 in figure 2.

When the assertions of type either `assert_change` or `assert_time` is written with option $f = 1$, we need one *auxiliary monitor process (AMP)* to report, with the auxiliary state-variable l_α, when ϕ_1 is true for the first interval. The AMP's behavior for α is shown in figure 4. There are four modes in AMP to reflect all combinations of truth values of l_α and ϕ_1. Every LG-transition in

the original system will now have to synchronize with a transition in the AMP. This is done with synchronizer $\texttt{moni}_\alpha$. We label the first process transition in each LG-transition with synchronization $?\texttt{moni}_\alpha$. In this way, the AMP is tightly synchronized with the original system and the beginning and ending of the assertion scope are precisely monitored.

When the system transits into the scope of assertion α, the AMP will also receive a synchronizer $?\texttt{set}_\alpha$, in addition to the sending out of synchronizer $!\texttt{moni}_\alpha$. On receiving $?\texttt{set}_\alpha$, the AMP will set the value l_α to report that the scope is entered. Then on every change value of ϕ_1 from true to false in a state with $l_\alpha = \mathit{true}$, l_α will be reset to false. When l_α changes from true to false, it means that the the system has left the first interval in which ϕ_1 is true in the scope of α.

8 Localized abstract assertion-checking

Verification problem is highly complex to solve with the state-space explosion problem. Thus it is very important to take advantage of whatever ideas, used in the designs, communicable from the design engineers to the verification engineers. The framework of OVL assertion-checking has advantage in this aspect because the assertions are given in between lines of process programs. Thus it is reasonable to assume that an assertion is either assured by the corresponding process or essential for the correctness of the process. Along this line of reasoning, we have developed three state-space abstraction technique, which we call *localized abstraction*. Unlike traditional abstraction techniques[40], our new technique adjust to the information coming with assertions.

Suppose we have an assertion α given in the program of process p. For α, a process p' is called *significant* if either $p = p'$ or some local variables of p' appear in α. All other processes are called *insignificant*. For an assertion, the three localized abstractions reduce the state-space representations by making abstractions on the state-variables of the insignificant processes. The three localized abstractions are described in the following. Suppose we have a state-space description η.

- $L^\alpha()$: *strictly local abstraction*
  $L^\alpha(\eta)$ is identical to η except all information about state-variables, except the operation modes, of insignificant processes are eliminated. The option can be activated with option `-Ad` of RED 4.0.
- $L^\alpha_d()$: *local and discrete abstraction*
  $L^\alpha_d(\eta)$ is identical to η except all information about local clocks of insignificant processes are eliminated. The option can be activated with option `-At` of RED 4.0.
- $L^\alpha_m()$: *local and magnitude abstraction*
  A clock inequality $x - x' \sim c$ is called a *magnitude constraint* iff either $x = 0$ or $x' = 0$. $L^\alpha_m(\eta)$ is identical to η except all non-magnitude clock difference constraints of the insignificant processes are eliminated. The option can be activated with option `-Am` of RED 4.0.

We report the performance of our three abstraction abstractions in section 9.

9 Verification experiments

The wireless communication standard of BlueTooth has been widely discussed and adopted in many appliances since the specification[11] was published. To show the usefulness of our techniques for industry projects, in the following, we report our verification experiments with the *L2CAP (Logical Link Control and Adaptation Layer Protocol)* of BlueTooth specification[11].

9.1 Modelling L2CAP

L2CAP is layered over the Baseband Protocol and resides in the data link layer of BlueTooth. This protocol supports higher level message multiplexing, packet segmentation and reassembly, and the conveying of quality of service information. We model the behavior of L2CAP in TC and write specification in OVL assertions. The protocol regulates the behaviors between a master device and a slave device. We use eight processes: the *master upper* (user on the master side), the *master* (L2CAP layer), master L2CAP time-out process, master L2CAP extended time-out process, the *slave upper* (user on the slave side), the *slave* (L2CAP layer), slave L2CAP time-out process, and slave L2CAP extended time-out process to model the whole system.

The SCTA in figure 5 describes the behavior of a L2CAP device described in the BlueTooth specification[11]. A device may play the role of either master or slave depending on whether the device starts the connection. Both the master and the slave use the SCTA in figure 5. A master is a device issuing a request while a slave is the one responding to the master's request.

The TC program of L2CAP with an OVL assertion is shown in appendix C. The corresponding optimized SCTA generated from RG is shown in appendix D. The original TC program has 303 lines of code. The optimized SCTA has 25 modes, 151 process transitions, 6 state variables, and 8 dense-time clocks in total.

The message sequence chart (MSC) in figure 6 may better illustrate a typical scenario of event sequence in L2CAP. The two outside vertical lines represent the L2CA interface from (slave's and master's) upper layers to the L2CAP layers (slave and master respectively). The scenario starts when the master's upper layer issues an `L2CA_ConnectReq` (Connection Request) through the L2CA interface. Upon receiving the request, the master communicates the request to the slave (with an `L2CAP_ConnectReq`), who will then convey the request to the slave's upper layer (with an

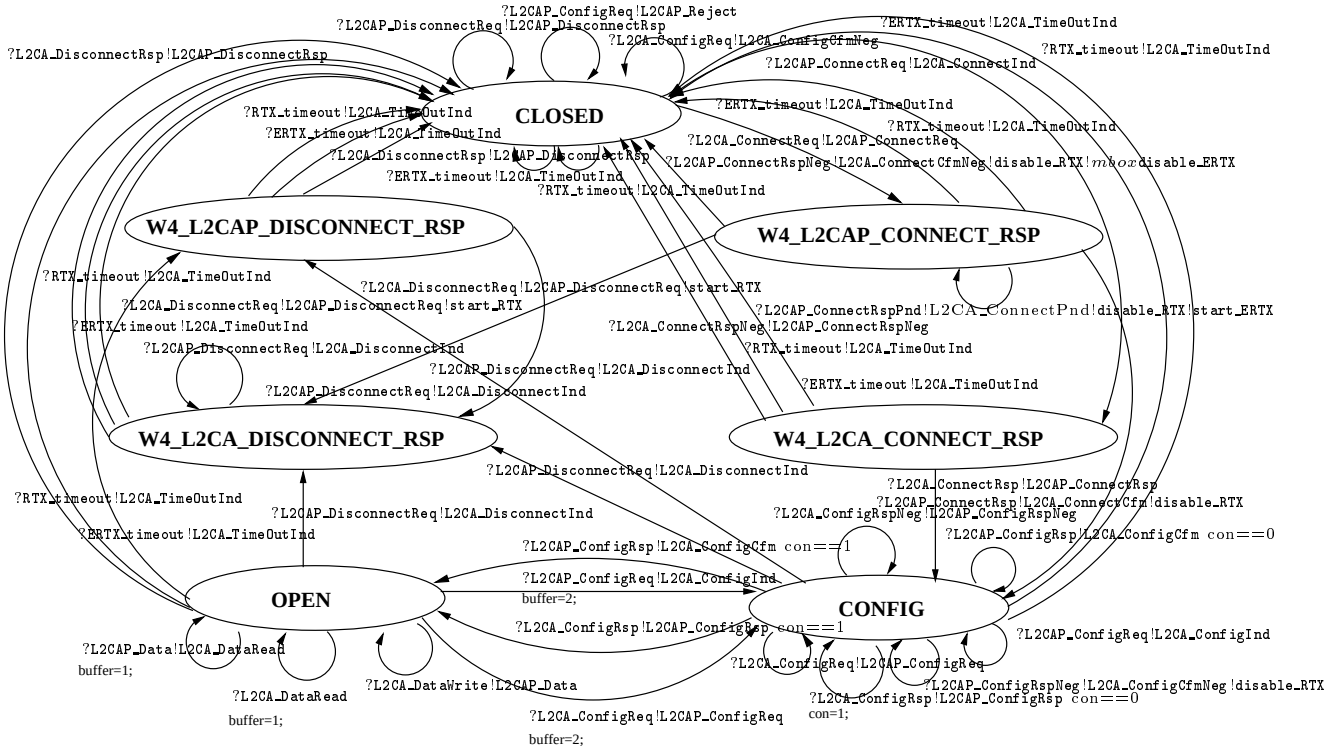

Figure 5: SCTA of a Bluetooth device

L2CA_ConnectInd). The protocol goes on with messages bouncing back and forth until the master sends an L2CAP_ConfigRsp message to the slave. Then both sides exchange data. Finally the master upper layer issues message L2CA_DisconnectReq to close the connection and the slave confirms the disconnection.

We have made the following assumption in the model. When an upper layer process needs to send out an event in response to the receiving of an event, the time between the receiving and sending is in $[0, 5]$. Also, we assume that the timeout value of RTX timers and ERTX timers are all 60 time units. With one timeout, the L2CAP process aborts the session and changes to state CLOSED.

9.2 Performance data

We have experimented with four OVL assertions. The first is

```
//assert_always(M_Con == 0)                    (a)
```

inserted at the beginning of the switch-case W4_L2CAP_CONNECT_RSP of the master TC process program. M_Con is a binary flag used to check if connection requests have been received from both master upper and slave. The TC program with assertion (a) are presented in appendices C. The assertion is satisfied because at the time process master enters state W4_L2CAP_CONNECT_RSP, the master reset M_Con to zero as initial value.

The second OVL assertion is

```
//assert_never(S_Con==0)                       (b)
```

inserted at the beginning of the switch-case W4_L2CAP_CONNECT_RSP of the slave TC process program. S_Con is the counterpart of M_Con. The assertion is thus not satisfied.

The third OVL assertion is

```
//assert_change #([0,60],0)
  c(master_status==W4_L2CAP_CONNECT_RSP,       (c)
  master_status==W4_L2CAP_CONNECT_RSP)
```

which says that if the master enters state W4_L2CAP_CONNECT_RSP, then it will eventually leave the state. The assertion is inserted at the beginning of the master TC process. This is satisfied because of the timeout issued from timer M_RTX.

The fourth OVL assertion is

```
//assert_time #([0,oo),1)
   d(slave_status==W4_L2CAP_DISCONNECT_RSP,    (d)
   slave_status==W4_L2CAP_DISCONNECT_RSP)
```

which says that if the slave enters state W4_L2CAP_DISCONNECT_RSP, then it will never leave the

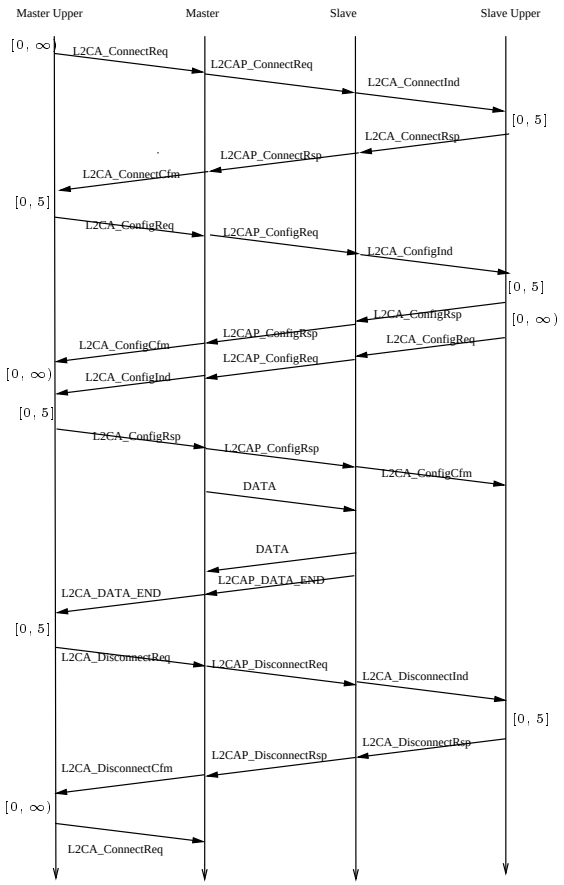

Figure 6: A message sequence chart of L2CAP

state. "oo" is our notation for infinity ∞. The assertion is inserted at the beginning of the slave TC process. This is NOT satisfied because of the timeout issued from timer S_RTX.

The verification performance of RED 4.0 with and without localized abstraction technique against the four assertions is shown in table 2. The sizes of SCTAs for the four assertions, before and after optimizaton, are also reported. In the following, we analyze the meaning of the performance data.

9.3 Performance effect of optimization

With our optimization techniques discussed in subsection 5.3, significant reduction in SCTA size is achieved for each of the assertions. In all four assertions, the numbers of modes in optimized SCTAs are reduced to around one tenth of those in unoptimized SCTAs. Also the numbers of transitions are also reduced to less than half. In our experience, the time needed to model-check timed automata is exponential to the size of input. Thus we do expect that the unoptimized SCTA will be much harder to verify. This expectation is justified by comparing the verification performance for the optimized and unoptimized SCTAs. In all cases, the optimzed SCTAs allow efficient verification within less than 1 min while the corresponding SCTAs do not allow verification tasks to finish in 20 mins. The performance data in table 2 shows that our SCTA optimization techniques are indeed indispensible.

9.4 Performance effect of localized abstractions

In table 2, for each assertion against their optimized SCTAs, we see that the verification performances with localized abstraction technique are all better than the one without. This is because that in the L2CAP process, there are local variables `M_Con` and `S_Con` and in the upper layer and timeout processes, there are local clocks `metric`. For the four assertions, only the process in whose program the assertion is written is significant. With the localized abstraction technique, state information on local variables of insiginificant processes can be eliminated to some extent and the state-space representations can be manipulated more efficiently. We believe that from the performance comparison, we find that our localized abstraction technique can indeed be of use in practice.

Among the three localized abstraction functions, we also observe difference in performance. Initially, since $L^{\alpha}()$ eliminate more state-information than $L^{\alpha}_{m}()$ and $L^{\alpha}_{d}()$ do, we expect $L^{\alpha}()$ will result in the most reduced state-space representations and the best verification performance. To our surprise, function $L^{\alpha}()$ performs the worst against three of the four assertions. We spent sometime to look into the intermediate data generated with $L^{\alpha}()$. We found that because information like `M_Con==1` can be eliminated, state-space representations with both `M_Con==0` and `M_Con==1` will be generated. But the corresponding state-space with `M_Con==0` may otherwise be unreachable without the abstraction of $L^{\alpha}()$. Such false reachable state-spaces can in turn trigger more transitions, which are otherwise not triggerable. Thus, with $L^{\alpha}()$, we actually may waste time/space in computing representations for unreachable state-spaces. This explains why there is the performance difference among the three localized abstraction functions.

10 Deadlock detection

In the programming of complex systems, it is easy to fall into the situation that synchronization falls apart and the system stops progressing. This can happen when either a waiting period is set wrong or a message never appears. A situation, from which no progress of computation is possible, is called a *deadlock* and happens very often in practice.

In fact, in our modelling of the L2CAP, we have identified many bugs which trap the system behavior

optimization?	abstraction?	size or performance?	assertion (a)	assertion (b)	assertion (c)	assertion (d)
optimized	no	#modes/#transitions	25/151	25/151	24/150	28/166
		time/memory	21.61s/845k	23.71s/845k	34.95s/858k	49.27s/1869k
	$L^{\alpha}()$	time/memory	18.83s/845k	22.36s/845k	32.63s/858k	48.81s/1869k
	$L^{\alpha}_{d}()$	time/memory	19.22s/845k	19.82s/845k	28.74s/858k	40.63s/1869k
	$L^{\alpha}_{m}()$	time/memory	19.22s/845k	22.25s/845k	31.46s/858k	47.57s/1869k
not optimzed	no	#modes/#transitions	258/360	258/360	258/360	262/376
		time	>20min	>20min	>20min	>20min

Data collected in cygwin environment on a Pentium 4 with 1.7GHz, 256MB, running MS Windows XP.

Table 2: Verification performance of assertions with various options

in deadlock. In one instance, while the master process receives event ?M_L2CA_ConnectRsp from upper layer in state W4_L2CA_CONNECT_RSP, we should expect the master prcess to send out event !M_L2CAP_ConnectRsp to the slave process and changes to state CONFIG. But instead, we mistakenly put down event ?M_L2CAP_ConnectRsp and the synchronization falls apart. In the modelling of a tightly synchronized communication systems, such mistakes can happen very often. Engineers are in dire need for any automated tool in discovering the possibility of deadlocks in their system design.

To support quick discovery of deadlock possibilities, we have enhanced the functionality of RED. When invoked with option "-Dx," RED will automatically detect the existence of such deadlocks. When invoked additionally with option "-c," RED will also generate a trace demonstrating how such deadlocks can be reached.

We here give a brief description on how to do deadlock detection in RED. But due to page-limit, we do not plan to go into details. In RED, we have a big BDD named FX, which represents the combination of process transitions for LG-transitions. We can also add the triggering condition of each process transitions to FX so that it becomes the combination of process transitions with triggering conditions for LG-transitions. Then the negation of FX denotes the state-space in which no further LG-transition can be triggered. Namely, ¬FX represents the condition for deadlock. Then we use ¬FX as the risk condition and perform reachability analysis with RED to see if a deadlock state is reachable.

When we found that our model (with the just-mentioned deadlock bug) did not generate an infinite computation, we suspected the existence of a deadlock. We experiemented to use this new function of RED 4.0 to try to discover any deadlock bug. And RED is capable of generating a trace leading to the deadlock state in 10.36 sec in CPU time and 834k memory usage. The experiment was also performed in the cygwin environment on a Pentium 4 with 1.7GHz clock rate and 256MB memory, running MS Windows XP.

11 Conclusion

This paper describes a new tool supporting formal OVL assertion-checking of dense-time concurrent systems. A formal state-transition graph model of the system and TCTL formulae of the properties are constructed from a description written in the TC language. We show how to mechanically translate TC-programs into optimized SCTAs. To take advantage of the information coming with OVL assertions for better verification performance, We demonstrate the power of new techniques by verifying the wireless communication L2CAP in BlueTooth.

Since our framework are based on RED, which supports high-performance full TCTL symbolic model checking, we feel hopeful that the techniques presented here can be applied to real world industry projects. The major motivation of this work is to provide a natural and friendly verification process to reduce the entry barrier to CAV technology, especially for engineers of real-time and embedded systems. And our experiment data on the real-world L2CAP indeed shows great promise of verification in the style of OVL assertion-checking for dense-time concurrent systems.

References

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

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

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, Y. Zbar The ForSpec Temporal Logic: A New Temporal Property-Specification Language (2001), TACAS'2002.

[4] A.V. Aho, R. Sethi, J.D. Ullman. *Compliers - Principles, Techniques, and Tools*, pp.393-396, Addison-Wesley Publishing Company, 1986.

[5] J. Bhasker. A VHDL Primer, third edition, ISBN 0-13-096575-8, Prentice Hall, 1999.

[6] Basic Spin Manual, http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html

[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] 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.

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

[10] 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.

[11] Specification of the Bluetooth System Version 1.1, Feb, 2001. http://www.bluetooth.org

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

[13] S. Campos, E. Clarke, W. Marrero and M. Minea . Verus: a tool for quantitative analysis of finite-state real-time systems. In: Workshop on Languages, Compilers and Tools for Real-Time Systems, 1995.

[14] E.M. Clarke, S.M. German, Y. Lu, H. Veith, D. Wang. Executable protocol specificatoin in esl, FMCAD'2000, LNCS 1954, pp.197-216, Springer-Verlag.

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

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

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

[18] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gubstafsson, H. Hansson. Worst-case execution-time analysis for embedded real-time systems. Journal of Software Tools for Technology Transfer, 2001. 14

[19] 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.

[20] F. Haque, K. Khan, J. Michelson. The Art of Verification with VERAR, 2001, Verification Central Com.

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

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

[23] 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.

[24] R.P. Kurshan. FormalCheck User's Manual, Cadence Design, Inc., 1998.

[25] M.J. Morley. Semantics of temporal e. Banff'99 Higher Order Workshop (Formal Methods in Computation). University of Glasgow, Dept. of Computer Science Technical Report, 1999.

[26] F. Nekoogar. Timing Verification of Application-Specific Integrated Circuits (ASICs), 2000, ISBN: 0-13-794348-2, Prentice-Hall.

[27] http://www.verificationlib.com/

[28] S. Palnitkar Verilog HDL: A Guide to Digital Design and Synthesis ISBN 0-13-451675-3, Sun Microsystems Press.

[29] P.Rashinkar, P. Paterson, L. Singh. System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, 2000; ISBN: 0792372794.

[30] V. Sagdeo. The Complete VERILOG Book Kluwer Academic Publishers, 1998; ISBN: 0792381882.

[31] Superlog, Co-Design Automation, Inc. 1998-2002; `http://www.superlog.org/`

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

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

[34] 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.

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

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

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

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

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

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

[41] 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 non-negative 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 **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
SCTA (Synchronized Concurrent Timed Automata) 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 **states** 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 TCTL semantics

Definition 5 (Satisfaction of TCTL formulae): We write in notations $\nu \models \phi$ to mean that ϕ is satisfied at state ν in S. The satisfaction relation is defined inductively as follows.

- The base case of $\phi \in B(P, X)$ was previously defined;
- $\nu \models \phi_1 \vee \phi_2$ iff either $\nu \models \phi_1$ or $\nu \models \phi_2$
- $\nu \models \neg\phi_1$ iff $\nu \not\models \phi_1$
- $\nu \models \exists\phi_1 \mathcal{U}_\mathcal{I} \phi_2$ iff there exist a ν-run $= ((\nu_1, t_1), (\nu_2, t_2), \ldots)$ in A, an $i \geq 1$, and a $\delta \in [0, t_{i+1} - t_i]$, s.t.
  - $t_i + \delta - t_1 \in \mathcal{I}$,
  - $\nu_i + \delta \models \phi_2$,
  - for all j, δ' s.t. either $(0 \leq j < i) \wedge (\delta' \in [0, t_{j+1} - t_j])$ or $(j = i) \wedge (\delta' \in [0, \delta))$, $\nu_j + \delta' \models \phi_1$.

  (In words, there exists a ν-run along which ϕ_2 eventually holds at some point in time ($\sim t_1 + c$) in the time interval $[t_i, t_{i+1}]$, for some i, and before reaching that point ϕ_1 always holds.)
- $\nu \models \forall\phi_1 \mathcal{U}_\mathcal{I} \phi_2$ iff for every ν-run $= ((q_1, \nu_1, t_1), (q_2, \nu_2, t_2), \ldots)$ in A, for some $i \geq 1$ and $\delta \in [0, t_{i+1} - t_i]$,
  - $t_i + \delta - t_1 \in \mathcal{I}$,
  - $\nu_i + \delta \models \phi_2$,

– for all j, δ' s.t. either $(0 \leq j < i) \wedge (\delta' \in [0, t_{j+1} - t_j])$ or $(j = i) \wedge (\delta' \in [0, \delta))$, $\nu_j + \delta' \models \phi_1$.

Given a shared-variable concurrent timed automaton S and a TCTL formula ϕ, we say S is a *model* of ϕ, written as $S \models \phi$, iff $\mathbf{0} \models \phi$ where $\mathbf{0}$ is the mapping that maps $\texttt{mode}_p$ to $q_{p,0}$, all global variables and all clocks to zeros. ||

C TC program for L2CAP with an assertion

```
/*UB: 5,  RTX: 6, ERTX: 10*/
process master_upper () {
  while (1) {
    switch event {
    case <!M_L2CA_ConnectReq> :
      break;
    case <?M_L2CA_ConnectInd>:
      [0,5];
      switch event {
      case <!M_L2CA_ConnectRsp>:
        break;
      case <!M_L2CA_ConnectRspNeg>:
        break;
      }
      break;
    case <?M_L2CA_ConfigInd>:
      [0,5]; <!M_L2CA_ConfigRsp>; break;
    case <!M_L2CA_ConfigReq>:
      break;
    case <!M_L2CA_DataWrite>:
      break;
    case <?M_L2CA_ConnectCfm>:
      [0,5]; <!M_L2CA_ConfigReq>; break;
    case <?M_L2CA_ConnectCfmNeg>:
      break;
    case <?M_L2CA_ConfigCfm> :
      break;
    case <?M_L2CA_ConfigCfmNeg>:
      break;
    case <?M_L2CA_DataRead>:
      break;
    case <!M_L2CA_DisconnectReq>:
      break;
    case <?M_L2CA_DisconnectInd>:
      [0,5]; <!M_L2CA_DisconnectRsp>; break;
    case <?M_L2CA_DisconnectCfm>:
      break;
    case <?M_L2CA_TimeOutInd>:
      break;
    }
  }
}

process master() {
  enum{CLOSED, W4_L2CAP_CONNECT_RSP, W4_L2CA_CONNECT_RSP, CONFIG, OPEN, W4_L2CA_DISCONNECT_RSP, W4_L2CAP_DISCONNECT_RSP} master_status;
  int M_Buffer, M_Con;

  while(1) {
    switch (master_status) {
    case CLOSED:
      switch event {
      case <?S_L2CAP_ConnectReq !M_L2CA_ConnectInd>:
        M_Con =0; [0,0]; master_status=W4_L2CA_CONNECT_RSP; break;
      case <?S_L2CAP_ConfigReq !M_L2CAP_Reject>:
        break;
      case <?M_L2CA_ConnectReq !M_L2CAP_ConnectReq !start_M_RTX>:
        M_Con =0; [0,0]; master_status=W4_L2CAP_CONNECT_RSP; break;
      case <?M_L2CA_ConfigReq !M_L2CA_ConfigCfmNeg>:
        break;
      case <?S_L2CAP_DisconnectReq !M_L2CAP_DisconnectRsp>:
        break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case W4_L2CAP_CONNECT_RSP:
      //assert_always(M_Con == 0)
      switch event {
      case <? S_L2CAP_ConnectRsp !M_L2CA_ConnectCfm !disable_M_RTX>:
        master_status = CONFIG; break;
      case <?S_L2CAP_ConnectRspPnd !M_L2CA_ConnectPnd !disable_M_RTX !start_M_ERTX>:
        break;
      case <? S_L2CAP_ConnectRspNeg !M_L2CA_ConnectCfmNeg !disable_M_RTX !disable_M_ERTX> :
        master_status = CLOSED; break;
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd> :
        master_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case W4_L2CA_CONNECT_RSP:
      switch event {
      case <?M_L2CA_ConnectRsp !M_L2CAP_ConnectRsp>:
        master_status = CONFIG; break;
      case <?M_L2CA_ConnectRspNeg !M_L2CAP_ConnectRspNeg>:
        master_status = CLOSED; break;
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd>:
        master_status = CLOSED; break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case CONFIG:
      switch event {
      case <?S_L2CAP_ConfigReq !M_L2CA_ConfigInd>:
        break;
      case <?S_L2CAP_ConfigRsp !M_L2CA_ConfigCfm !disable_M_RTX> M_Con==0:
        M_Con=1; break;
```

```
      case <?S_L2CAP_ConfigRsp !M_L2CA_ConfigCfm !disable_M_RTX> M_Con==1:
        master_status = OPEN; break;
      case <?S_L2CAP_ConfigRspNeg !M_L2CA_ConfigCfmNeg !disable_M_RTX> :
        break;
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd> :
        master_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?M_L2CA_ConfigReq !M_L2CAP_ConfigReq !start_M_RTX> :
        break;
      case <?M_L2CA_ConfigRsp !M_L2CAP_ConfigRsp> M_Con==0:
        M_Con=1; break;
      case <?M_L2CA_ConfigRsp !M_L2CAP_ConfigRsp> M_Con==1:
        master_status = OPEN; break;
      case <?M_L2CA_ConfigRspNeg !M_L2CAP_ConfigRspNeg>:
        break;
      case <?M_L2CA_DisconnectReq !M_L2CAP_DisconnectReq !start_M_RTX>:
        master_status = W4_L2CAP_DISCONNECT_RSP; break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case OPEN:
      switch event {
      case <?S_L2CAP_ConfigReq !M_L2CA_ConfigInd>:
        master_status = CONFIG; break;
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd>:
        master_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?S_L2CAP_Data !M_L2CA_DataRead>:
        break;
      case <?M_L2CA_ConfigReq !M_L2CAP_ConfigReq !start_M_RTX>:
        master_status = CONFIG; break;
      case <?M_L2CA_DisconnectReq !M_L2CAP_DisconnectReq !start_M_RTX>:
        master_status = W4_L2CAP_DISCONNECT_RSP; break;
      case <?M_L2CA_DataWrite !M_L2CAP_Data>:
        break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case W4_L2CA_DISCONNECT_RSP :
      switch event {
      case <?M_L2CA_DisconnectRsp !M_L2CAP_DisconnectRsp> :
        master_status = CLOSED; break;
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd> :
        break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      }
      break;
    case W4_L2CAP_DISCONNECT_RSP:
      switch event {
      case <?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd> :
        master_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?S_L2CAP_DisconnectRsp !M_L2CA_DisconnectCfm !disable_M_RTX> :
        master_status = CLOSED; break;
      case <?M_RTX_timeout !M_L2CA_TimeOutInd>:
        master_status = CLOSED; break;
      case <?M_ERTX_timeout !M_L2CA_TimeOutInd> :
        master_status = CLOSED; break;
      }
      break;
    }
  }
}

process M_RTX(){
  while(1){
    switch event {
    case <?start_M_RTX>:
      switch event {
      case <?disable_M_RTX>:
        break;
      timeout [60,60]:
        <!M_RTX_timeout>; break;
      }
      break;
    case <?disable_M_RTX>:
      break;
    }
  }
}

process M_ERTX(){
  while(1){
    switch event {
    case <?start_M_ERTX>:
      switch event {
      case <?disable_M_ERTX>:
        break;
      timeout [60,60]:
        <!M_ERTX_timeout>; break;
      }
      break;
    case <?disable_M_ERTX>:
      break;
    }
  }
}

process slave_upper () {
  while (1) {
    switch event {
```

```
    case <!S_L2CA_ConnectReq> :
      break;
    case <?S_L2CA_ConnectInd>:
      [0,5];
      switch event {
      case <!S_L2CA_ConnectRsp>:
        break;
      case <!S_L2CA_ConnectRspNeg>:
        break;
      }
      break;
    case <?S_L2CA_ConfigInd>:
      [0,5]; <!S_L2CA_ConfigRsp>; break;
    case <!S_L2CA_ConfigReq>:
      break;
    case <!S_L2CA_DataWrite>:
      break;
    case <?S_L2CA_ConnectCfm>:
      [0,5]; <!S_L2CA_ConfigReq>; break;
    case <?S_L2CA_ConnectCfmNeg>:
      break;
    case <?S_L2CA_ConfigCfm> :
      break;
    case <?S_L2CA_ConfigCfmNeg>:
      break;
    case <?S_L2CA_DataRead>:
      break;
    case <!S_L2CA_DisconnectReq>:
      break;
    case <?S_L2CA_DisconnectInd>:
      [0,5]; <!S_L2CA_DisconnectRsp>; break;
    case <?S_L2CA_DisconnectCfm>:
      break;
    case <?S_L2CA_TimeOutInd>:
      break;
    }
  }
}

process slave(){
  enum{CLOSED, W4_L2CAP_CONNECT_RSP, W4_L2CA_CONNECT_RSP, CONFIG, OPEN, W4_L2CA_DISCONNECT_RSP, W4_L2CAP_DISCONNECT_RSP} slave_status;
  int S_Buffer, S_Con;

  while(1){
    switch (slave_status) {
    case CLOSED:
      switch event {
      case <?M_L2CAP_ConnectReq !S_L2CA_ConnectInd>:
        S_Con =0; [0,0]; slave_status=W4_L2CA_CONNECT_RSP; break;
      case <?M_L2CAP_ConfigReq !S_L2CAP_Reject>:
        break;
      case <?S_L2CA_ConnectReq !S_L2CAP_ConnectReq !start_S_RTX>:
        S_Con =0; [0,0]; slave_status=W4_L2CAP_CONNECT_RSP; break;
      case <? S_L2CA_ConfigReq !S_L2CA_ConfigCfmNeg>:
        break;
      case <?M_L2CAP_DisconnectReq !S_L2CAP_DisconnectRsp>:
        break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      }
      break;
    case W4_L2CAP_CONNECT_RSP:
      switch event {
      case <? M_L2CAP_ConnectRsp !S_L2CA_ConnectCfm !disable_S_RTX>:
        slave_status = CONFIG; break;
      case <?M_L2CAP_ConnectRspPnd !S_L2CA_ConnectPnd !disable_S_RTX !start_S_ERTX>:
        break;
      case <? M_L2CAP_ConnectRspNeg !S_L2CA_ConnectCfmNeg !disable_S_RTX !disable_S_ERTX> :
        slave_status = CLOSED; break;
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd> :
        slave_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED;
        break;
      }
      break;
    case W4_L2CA_CONNECT_RSP:
      switch event {
      case <?S_L2CA_ConnectRsp !S_L2CAP_ConnectRsp>:
        slave_status = CONFIG; break;
      case <?S_L2CA_ConnectRspNeg !S_L2CAP_ConnectRspNeg>:
        slave_status = CLOSED; break;
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd>:
        slave_status = CLOSED; break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      }
      break;
    case CONFIG:
      switch event {
      case <?M_L2CAP_ConfigReq !S_L2CA_ConfigInd>:
        break;
      case <?M_L2CAP_ConfigRsp !S_L2CA_ConfigCfm !disable_S_RTX> S_Con==0:
        S_Con = 1; break;
      case <?M_L2CAP_ConfigRsp !S_L2CA_ConfigCfm !disable_S_RTX> S_Con==1:
        slave_status = OPEN; break;
      case <?M_L2CAP_ConfigRspNeg !S_L2CA_ConfigCfmNeg !disable_S_RTX> :
        break;
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd> :
        slave_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?S_L2CA_ConfigReq !S_L2CAP_ConfigReq !start_S_RTX> :
```

```
        break;
      case <?S_L2CA_ConfigRsp !S_L2CAP_ConfigRsp> S_Con==0:
        S_Con =1; break;
      case <?S_L2CA_ConfigRsp !S_L2CAP_ConfigRsp> S_Con==1:
        slave_status = OPEN; break;
      case <?S_L2CA_ConfigRspNeg !S_L2CAP_ConfigRspNeg>:
        break;
      case <?S_L2CA_DisconnectReq !S_L2CAP_DisconnectReq !start_S_RTX>:
        slave_status = W4_L2CAP_DISCONNECT_RSP; break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      }
      break;
    case OPEN:
      switch event {
      case <?M_L2CAP_ConfigReq !S_L2CA_ConfigInd>:
        slave_status = CONFIG; break;
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd>:
        slave_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?M_L2CAP_Data !S_L2CA_DataRead>:
        break;
      case <?S_L2CA_ConfigReq !S_L2CAP_ConfigReq !start_S_RTX>:
        slave_status = CONFIG; break;
      case <?S_L2CA_DisconnectReq !S_L2CAP_DisconnectReq !start_S_RTX>:
        slave_status = W4_L2CAP_DISCONNECT_RSP; break;
      case <?S_L2CA_DataWrite !S_L2CAP_Data>:
        break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      }
      break;
    case W4_L2CA_DISCONNECT_RSP :
      switch event {
      case <?S_L2CA_DisconnectRsp !S_L2CAP_DisconnectRsp> :
        slave_status = CLOSED; break;
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd> :
        break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      }
      break;
    case W4_L2CAP_DISCONNECT_RSP:
      switch event {
      case <?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd> :
        slave_status = W4_L2CA_DISCONNECT_RSP; break;
      case <?M_L2CAP_DisconnectRsp !S_L2CA_DisconnectCfm !disable_S_RTX> :
        slave_status = CLOSED; break;
      case <?S_RTX_timeout !S_L2CA_TimeOutInd>:
        slave_status = CLOSED; break;
      case <?S_ERTX_timeout !S_L2CA_TimeOutInd> :
        slave_status = CLOSED; break;
      }
      break;
    }
  }
}

process S_RTX(){
  while(1){
    switch event {
    case <?start_S_RTX>:
      switch event {
      case <?disable_S_RTX>:
        break;
      timeout [60,60]:
        <!S_RTX_timeout>; break;
      }
      break;
    case <?disable_S_RTX>:
      break;
    }
  }
}

process S_ERTX(){
  while(1){
    switch event {
    case <?start_S_ERTX>:
      switch event {
      case <?disable_S_ERTX>:
        break;
      timeout [60,60]:
        <!S_ERTX_timeout>; break;
      }
      break;
    case <?disable_S_ERTX>:
      break;
    }
  }
}
```

D Optimized SCTA and TCTL for L2CAP

```
#define CLOSED 0
#define W4_L2CAP_CONNECT_RSP 1
#define W4_L2CA_CONNECT_RSP 2
#define CONFIG 3
```

```
#define OPEN 4
#define W4_L2CA_DISCONNECT_RSP 5
#define W4_L2CAP_DISCONNECT_RSP 6

process count = 8;
local clock metric ;
local discrete master_status:0..7;
local discrete M_Buffer:0..5;
local discrete M_Con:0..5;
local discrete slave_status:0..7;
local discrete S_Buffer:0..5;
local discrete S_Con:0..5;
global synchronizer M_L2CA_ConnectReq, M_L2CA_ConnectInd, M_L2CA_ConnectRsp, M_L2CA_ConnectRspNeg, M_L2CA_ConfigInd, M_L2CA_ConfigRsp, M_L2CA_ConfigReq, M_L2CA_DataWrite, M_L2CA_ConnectCfm,
                    M_L2CA_ConnectCfmNeg, M_L2CA_ConfigCfm, M_L2CA_ConfigCfmNeg, M_L2CA_DataRead, M_L2CA_DisconnectReq, M_L2CA_DisconnectInd, M_L2CA_DisconnectRsp, M_L2CA_DisconnectCfm,
                    M_L2CA_TimeOutInd, S_L2CAP_ConnectReq, S_L2CAP_ConfigReq, M_L2CAP_Reject, M_L2CAP_ConnectReq, start_M_RTX, S_L2CAP_DisconnectReq, M_L2CAP_DisconnectRsp, M_RTX_timeout,
                    M_ERTX_timeout, S_L2CAP_ConnectRsp, disable_M_RTX, S_L2CAP_ConnectRspPnd, M_L2CA_ConnectPnd, start_M_ERTX, S_L2CAP_ConnectRspNeg, disable_M_ERTX, M_L2CAP_ConnectRsp,
                    M_L2CAP_ConnectRspNeg, S_L2CAP_ConfigRsp, S_L2CAP_ConfigRspNeg, M_L2CAP_ConfigReq, M_L2CAP_ConfigRsp, M_L2CA_ConfigRspNeg, M_L2CAP_ConfigRspNeg, M_L2CAP_DisconnectReq,
                    S_L2CAP_Data, M_L2CAP_Data, S_L2CAP_DisconnectRsp, S_L2CA_ConnectReq, S_L2CA_ConnectInd, S_L2CA_ConnectRsp, S_L2CA_ConnectRspNeg, S_L2CA_ConfigInd, S_L2CA_ConfigRsp,
                    S_L2CA_ConfigReq, S_L2CA_DataWrite, S_L2CA_ConnectCfm, S_L2CA_ConnectCfmNeg, S_L2CA_ConfigCfm, S_L2CA_ConfigCfmNeg, S_L2CA_DataRead, S_L2CA_DisconnectReq,
                    S_L2CA_DisconnectInd, S_L2CA_DisconnectRsp, S_L2CA_DisconnectCfm, S_L2CA_TimeOutInd, S_L2CAP_Reject, start_S_RTX, S_RTX_timeout, S_ERTX_timeout, disable_S_RTX,
                    M_L2CAP_ConnectRspPnd, S_L2CA_ConnectPnd, start_S_ERTX, disable_S_ERTX, S_L2CA_ConfigRspNeg;

/*************************************************************
*** Process [1] will be initialed here!
*/
mode master_upper_whilestart1 true {
 /*1*/   when ?M_L2CA_ConnectInd true may metric = 0;  goto master_upper_intervalzone4;
 /*2*/   when ?M_L2CA_ConfigInd true may metric = 0;  goto master_upper_intervalzone9;
 /*3*/   when ?M_L2CA_ConnectCfm true may metric = 0;  goto master_upper_intervalzone13;
 /*4*/   when ?M_L2CA_DisconnectInd true may metric = 0;  goto master_upper_intervalzone17;
 /*5*/   when !M_L2CA_ConnectReq true may  goto master_upper_whilestart1;
 /*6*/   when !M_L2CA_ConfigReq true may  goto master_upper_whilestart1;
 /*7*/   when !M_L2CA_DataWrite true may  goto master_upper_whilestart1;
 /*8*/   when ?M_L2CA_ConnectCfmNeg true may  goto master_upper_whilestart1;
 /*9*/   when ?M_L2CA_ConfigCfm true may  goto master_upper_whilestart1;
 /*10*/   when ?M_L2CA_ConfigCfmNeg true may  goto master_upper_whilestart1;
 /*11*/   when ?M_L2CA_DataRead true may  goto master_upper_whilestart1;
 /*12*/   when !M_L2CA_DisconnectReq true may  goto master_upper_whilestart1;
 /*13*/   when ?M_L2CA_DisconnectCfm true may  goto master_upper_whilestart1;
 /*14*/   when ?M_L2CA_TimeOutInd true may  goto master_upper_whilestart1;
}

mode master_upper_intervalzone4 metric<=5 {
 /*15*/   when !M_L2CA_ConnectRsp metric>=0 may  goto master_upper_whilestart1;
 /*16*/   when !M_L2CA_ConnectRspNeg metric>=0 may  goto master_upper_whilestart1;
}

mode master_upper_intervalzone9 metric<=5 {
 /*17*/   when !M_L2CA_ConfigRsp metric>=0 may  goto master_upper_whilestart1;
}

mode master_upper_intervalzone13 metric<=5 {
 /*18*/   when !M_L2CA_ConfigReq metric>=0 may  goto master_upper_whilestart1;
}

mode master_upper_intervalzone17 metric<=5 {
 /*19*/   when !M_L2CA_DisconnectRsp metric>=0 may  goto master_upper_whilestart1;
}


/*************************************************************
*** Process [2] will be initialed here!
*/
mode master_whilestart21 true {
 /*20*/   when master_status == 1 may  goto master_switchstart40;
 /*21*/   when ?S_L2CAP_ConnectReq !M_L2CA_ConnectInd master_status == 0 may M_Con = 0; metric = 0;  goto master_intervalzone28;
 /*22*/   when ?M_L2CA_ConnectReq !M_L2CAP_ConnectReq !start_M_RTX master_status == 0 may M_Con = 0; metric = 0;  goto master_intervalzone33;
 /*23*/   when ?S_L2CAP_ConfigReq !M_L2CAP_Reject master_status == 0 may  goto master_whilestart21;
 /*24*/   when ?M_L2CA_ConfigReq !M_L2CA_ConfigCfmNeg master_status == 0 may  goto master_whilestart21;
 /*25*/   when ?S_L2CAP_DisconnectReq !M_L2CAP_DisconnectRsp master_status == 0 may  goto master_whilestart21;
 /*26*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 0 may master_status = CLOSED;  goto master_whilestart21;
 /*27*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 0 may master_status = CLOSED;  goto master_whilestart21;
 /*28*/   when ?M_L2CA_ConnectRsp !M_L2CAP_ConnectRsp master_status == 2 may master_status = CONFIG;  goto master_whilestart21;
 /*29*/   when ?M_L2CA_ConnectRspNeg !M_L2CAP_ConnectRspNeg master_status == 2 may master_status = CLOSED;  goto master_whilestart21;
 /*30*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd master_status == 2 may master_status = CLOSED;  goto master_whilestart21;
 /*31*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 2 may master_status = CLOSED;  goto master_whilestart21;
 /*32*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 2 may master_status = CLOSED;  goto master_whilestart21;
 /*33*/   when ?S_L2CAP_ConfigReq !M_L2CA_ConfigInd master_status == 3 may  goto master_whilestart21;
 /*34*/   when ?S_L2CAP_ConfigRspNeg !M_L2CA_ConfigCfmNeg !disable_M_RTX master_status == 3 may  goto master_whilestart21;
 /*35*/   when ?M_L2CA_ConfigReq !M_L2CAP_ConfigReq !start_M_RTX master_status == 3 may  goto master_whilestart21;
 /*36*/   when ?M_L2CA_ConfigRspNeg !M_L2CAP_ConfigRspNeg master_status == 3 may  goto master_whilestart21;
 /*37*/   when ?S_L2CAP_ConfigRsp !M_L2CA_ConfigCfm !disable_M_RTX master_status == 3 and M_Con == 0 may M_Con = 1;  goto master_whilestart21;
 /*38*/   when ?S_L2CAP_ConfigRsp !M_L2CA_ConfigCfm !disable_M_RTX master_status == 3 and M_Con == 1 may master_status = OPEN;  goto master_whilestart21;
 /*39*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd master_status == 3 may master_status = W4_L2CA_DISCONNECT_RSP;  goto master_whilestart21;
 /*40*/   when ?M_L2CA_ConfigRsp !M_L2CAP_ConfigRsp master_status == 3 and M_Con == 0 may M_Con = 1;  goto master_whilestart21;
 /*41*/   when ?M_L2CA_ConfigRsp !M_L2CAP_ConfigRsp master_status == 3 and M_Con == 1 may master_status = OPEN;  goto master_whilestart21;
 /*42*/   when ?M_L2CA_DisconnectReq !M_L2CAP_DisconnectReq !start_M_RTX master_status == 3 may master_status = W4_L2CAP_DISCONNECT_RSP;  goto master_whilestart21;
 /*43*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 3 may master_status = CLOSED;  goto master_whilestart21;
 /*44*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 3 may master_status = CLOSED;  goto master_whilestart21;
 /*45*/   when ?S_L2CAP_Data !M_L2CA_DataRead master_status == 4 may  goto master_whilestart21;
 /*46*/   when ?M_L2CA_DataWrite !M_L2CAP_Data master_status == 4 may  goto master_whilestart21;
 /*47*/   when ?S_L2CAP_ConfigReq !M_L2CA_ConfigInd master_status == 4 may master_status = CONFIG;  goto master_whilestart21;
 /*48*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd master_status == 4 may master_status = W4_L2CA_DISCONNECT_RSP;  goto master_whilestart21;
 /*49*/   when ?M_L2CA_ConfigReq !M_L2CAP_ConfigReq !start_M_RTX master_status == 4 may master_status = CONFIG;  goto master_whilestart21;
 /*50*/   when ?M_L2CA_DisconnectReq !M_L2CAP_DisconnectReq !start_M_RTX master_status == 4 may master_status = W4_L2CAP_DISCONNECT_RSP;  goto master_whilestart21;
 /*51*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 4 may master_status = CLOSED;  goto master_whilestart21;
 /*52*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 4 may master_status = CLOSED;  goto master_whilestart21;
 /*53*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd master_status == 5 may  goto master_whilestart21;
 /*54*/   when ?M_L2CA_DisconnectRsp !M_L2CAP_DisconnectRsp master_status == 5 may master_status = CLOSED;  goto master_whilestart21;
 /*55*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 5 may master_status = CLOSED;  goto master_whilestart21;
 /*56*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 5 may master_status = CLOSED;  goto master_whilestart21;
 /*57*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd master_status == 6 may master_status = W4_L2CA_DISCONNECT_RSP;  goto master_whilestart21;
 /*58*/   when ?S_L2CAP_DisconnectRsp !M_L2CA_DisconnectCfm !disable_M_RTX master_status == 6 may master_status = CLOSED;  goto master_whilestart21;
 /*59*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd master_status == 6 may master_status = CLOSED;  goto master_whilestart21;
 /*60*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd master_status == 6 may master_status = CLOSED;  goto master_whilestart21;
}
```

```
mode master_intervalzone28 metric<=0 {
 /*61*/   when metric>=0 may master_status = W4_L2CA_CONNECT_RSP;  goto master_whilestart21;
}

mode master_intervalzone33 metric<=0 {
 /*62*/   when metric>=0 may master_status = W4_L2CAP_CONNECT_RSP;  goto master_whilestart21;
}

mode master_switchstart40 true {
 /*63*/   when ?S_L2CAP_ConnectRspPnd !M_L2CA_ConnectPnd !disable_M_RTX !start_M_ERTX true may  goto master_whilestart21;
 /*64*/   when ?S_L2CAP_ConnectRsp !M_L2CA_ConnectCfm !disable_M_RTX true may master_status = CONFIG;  goto master_whilestart21;
 /*65*/   when ?S_L2CAP_ConnectRspNeg !M_L2CA_ConnectCfmNeg !disable_M_RTX !disable_M_ERTX true may master_status = CLOSED;  goto master_whilestart21;
 /*66*/   when ?S_L2CAP_DisconnectReq !M_L2CA_DisconnectInd true may master_status = W4_L2CA_DISCONNECT_RSP;  goto master_whilestart21;
 /*67*/   when ?M_RTX_timeout !M_L2CA_TimeOutInd true may master_status = CLOSED;  goto master_whilestart21;
 /*68*/   when ?M_ERTX_timeout !M_L2CA_TimeOutInd true may master_status = CLOSED;  goto master_whilestart21;
}


/*************************************************************
*** Process [3] will be initialed here!
*/
mode M_RTX_whilestart114 true {
 /*69*/   when ?start_M_RTX true may metric = 0;  goto M_RTX_switchstart118;
 /*70*/   when ?disable_M_RTX true may  goto M_RTX_whilestart114;
}

mode M_RTX_switchstart118 metric<=60 {
 /*71*/   when ?disable_M_RTX true may  goto M_RTX_whilestart114;
 /*72*/   when !M_RTX_timeout metric>=60 may  goto M_RTX_whilestart114;
}


/*************************************************************
*** Process [4] will be initialed here!
*/
mode M_ERTX_whilestart122 true {
 /*73*/   when ?start_M_ERTX true may metric = 0;  goto M_ERTX_switchstart126;
 /*74*/   when ?disable_M_ERTX true may  goto M_ERTX_whilestart122;
}

mode M_ERTX_switchstart126 metric<=60 {
 /*75*/   when ?disable_M_ERTX true may  goto M_ERTX_whilestart122;
 /*76*/   when !M_ERTX_timeout metric>=60 may  goto M_ERTX_whilestart122;
}


/*************************************************************
*** Process [5] will be initialed here!
*/
mode slave_upper_whilestart130 true {
 /*77*/   when ?S_L2CA_ConnectInd true may metric = 0;  goto slave_upper_intervalzone133;
 /*78*/   when ?S_L2CA_ConfigInd true may metric = 0;  goto slave_upper_intervalzone138;
 /*79*/   when ?S_L2CA_ConnectCfm true may metric = 0;  goto slave_upper_intervalzone142;
 /*80*/   when ?S_L2CA_DisconnectInd true may metric = 0;  goto slave_upper_intervalzone146;
 /*81*/   when !S_L2CA_ConnectReq true may  goto slave_upper_whilestart130;
 /*82*/   when !S_L2CA_ConfigReq true may  goto slave_upper_whilestart130;
 /*83*/   when !S_L2CA_DataWrite true may  goto slave_upper_whilestart130;
 /*84*/   when ?S_L2CA_ConnectCfmNeg true may  goto slave_upper_whilestart130;
 /*85*/   when ?S_L2CA_ConfigCfm true may  goto slave_upper_whilestart130;
 /*86*/   when ?S_L2CA_ConfigCfmNeg true may  goto slave_upper_whilestart130;
 /*87*/   when ?S_L2CA_DataRead true may  goto slave_upper_whilestart130;
 /*88*/   when !S_L2CA_DisconnectReq true may  goto slave_upper_whilestart130;
 /*89*/   when ?S_L2CA_DisconnectCfm true may  goto slave_upper_whilestart130;
 /*90*/   when ?S_L2CA_TimeOutInd true may  goto slave_upper_whilestart130;
}

mode slave_upper_intervalzone133 metric<=5 {
 /*91*/   when !S_L2CA_ConnectRsp metric>=0 may  goto slave_upper_whilestart130;
 /*92*/   when !S_L2CA_ConnectRspNeg metric>=0 may  goto slave_upper_whilestart130;
}

mode slave_upper_intervalzone138 metric<=5 {
 /*93*/   when !S_L2CA_ConfigRsp metric>=0 may  goto slave_upper_whilestart130;
}

mode slave_upper_intervalzone142 metric<=5 {
 /*94*/   when !S_L2CA_ConfigReq metric>=0 may  goto slave_upper_whilestart130;
}

mode slave_upper_intervalzone146 metric<=5 {
 /*95*/   when !S_L2CA_DisconnectRsp metric>=0 may  goto slave_upper_whilestart130;
}


/*************************************************************
*** Process [6] will be initialed here!
*/
mode slave_whilestart150 true {
 /*96*/   when ?M_L2CAP_ConnectReq !S_L2CA_ConnectInd slave_status == 0 may S_Con = 0; metric = 0;  goto slave_intervalzone157;
 /*97*/   when ?S_L2CA_ConnectReq !S_L2CAP_ConnectReq !start_S_RTX slave_status == 0 may S_Con = 0; metric = 0;  goto slave_intervalzone162;
 /*98*/   when ?M_L2CAP_ConfigReq !S_L2CAP_Reject slave_status == 0 may  goto slave_whilestart150;
 /*99*/   when ?S_L2CA_ConfigReq !S_L2CA_ConfigCfmNeg slave_status == 0 may  goto slave_whilestart150;
 /*100*/   when ?M_L2CAP_DisconnectReq !S_L2CAP_DisconnectRsp slave_status == 0 may  goto slave_whilestart150;
 /*101*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 0 may slave_status = CLOSED;  goto slave_whilestart150;
 /*102*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 0 may slave_status = CLOSED;  goto slave_whilestart150;
 /*103*/   when ?M_L2CAP_ConnectRspPnd !S_L2CA_ConnectPnd !disable_S_RTX !start_S_ERTX slave_status == 1 may  goto slave_whilestart150;
 /*104*/   when ?M_L2CAP_ConnectRsp !S_L2CA_ConnectCfm !disable_S_RTX slave_status == 1 may slave_status = CONFIG;  goto slave_whilestart150;
 /*105*/   when ?M_L2CAP_ConnectRspNeg !S_L2CA_ConnectCfmNeg !disable_S_RTX !disable_S_ERTX slave_status == 1 may slave_status = CLOSED;  goto slave_whilestart150;
 /*106*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 1 may slave_status = W4_L2CA_DISCONNECT_RSP;  goto slave_whilestart150;
 /*107*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 1 may slave_status = CLOSED;  goto slave_whilestart150;
 /*108*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 1 may slave_status = CLOSED;  goto slave_whilestart150;
 /*109*/   when ?S_L2CA_ConnectRsp !S_L2CAP_ConnectRsp slave_status == 2 may slave_status = CONFIG;  goto slave_whilestart150;
 /*110*/   when ?S_L2CA_ConnectRspNeg !S_L2CAP_ConnectRspNeg slave_status == 2 may slave_status = CLOSED;  goto slave_whilestart150;
 /*111*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 2 may slave_status = CLOSED;  goto slave_whilestart150;
 /*112*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 2 may slave_status = CLOSED;  goto slave_whilestart150;
 /*113*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 2 may slave_status = CLOSED;  goto slave_whilestart150;
 /*114*/   when ?M_L2CAP_ConfigReq !S_L2CA_ConfigInd slave_status == 3 may  goto slave_whilestart150;
```

```
 /*115*/   when ?M_L2CAP_ConfigRspNeg !S_L2CA_ConfigCfmNeg !disable_S_RTX slave_status == 3 may  goto slave_whilestart150;
 /*116*/   when ?S_L2CA_ConfigReq !S_L2CAP_ConfigReq !start_S_RTX slave_status == 3 may  goto slave_whilestart150;
 /*117*/   when ?S_L2CA_ConfigRspNeg !S_L2CAP_ConfigRspNeg slave_status == 3 may  goto slave_whilestart150;
 /*118*/   when ?M_L2CAP_ConfigRsp !S_L2CA_ConfigCfm !disable_S_RTX slave_status == 3 and S_Con == 0 may S_Con = 1;  goto slave_whilestart150;
 /*119*/   when ?M_L2CAP_ConfigRsp !S_L2CA_ConfigCfm !disable_S_RTX slave_status == 3 and S_Con == 1 may slave_status = OPEN;  goto slave_whilestart150;
 /*120*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 3 may slave_status = W4_L2CA_DISCONNECT_RSP;  goto slave_whilestart150;
 /*121*/   when ?S_L2CA_ConfigRsp !S_L2CAP_ConfigRsp slave_status == 3 and S_Con == 0 may S_Con = 1;  goto slave_whilestart150;
 /*122*/   when ?S_L2CA_ConfigRsp !S_L2CAP_ConfigRsp slave_status == 3 and S_Con == 1 may slave_status = OPEN;  goto slave_whilestart150;
 /*123*/   when ?S_L2CA_DisconnectReq !S_L2CAP_DisconnectReq !start_S_RTX slave_status == 3 may slave_status = W4_L2CAP_DISCONNECT_RSP;  goto slave_whilestart150;
 /*124*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 3 may slave_status = CLOSED;  goto slave_whilestart150;
 /*125*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 3 may slave_status = CLOSED;  goto slave_whilestart150;
 /*126*/   when ?M_L2CAP_Data !S_L2CA_DataRead slave_status == 4 may  goto slave_whilestart150;
 /*127*/   when ?S_L2CA_DataWrite !S_L2CAP_Data slave_status == 4 may  goto slave_whilestart150;
 /*128*/   when ?M_L2CAP_ConfigReq !S_L2CA_ConfigInd slave_status == 4 may slave_status = CONFIG;  goto slave_whilestart150;
 /*129*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 4 may slave_status = W4_L2CA_DISCONNECT_RSP;  goto slave_whilestart150;
 /*130*/   when ?S_L2CA_ConfigReq !S_L2CAP_ConfigReq !start_S_RTX slave_status == 4 may slave_status = CONFIG;  goto slave_whilestart150;
 /*131*/   when ?S_L2CA_DisconnectReq !S_L2CAP_DisconnectReq !start_S_RTX slave_status == 4 may slave_status = W4_L2CAP_DISCONNECT_RSP;  goto slave_whilestart150;
 /*132*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 4 may slave_status = CLOSED;  goto slave_whilestart150;
 /*133*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 4 may slave_status = CLOSED;  goto slave_whilestart150;
 /*134*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 5 may  goto slave_whilestart150;
 /*135*/   when ?S_L2CA_DisconnectRsp !S_L2CAP_DisconnectRsp slave_status == 5 may slave_status = CLOSED;  goto slave_whilestart150;
 /*136*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 5 may slave_status = CLOSED;  goto slave_whilestart150;
 /*137*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 5 may slave_status = CLOSED;  goto slave_whilestart150;
 /*138*/   when ?M_L2CAP_DisconnectReq !S_L2CA_DisconnectInd slave_status == 6 may slave_status = W4_L2CA_DISCONNECT_RSP;  goto slave_whilestart150;
 /*139*/   when ?M_L2CAP_DisconnectRsp !S_L2CA_DisconnectCfm !disable_S_RTX slave_status == 6 may slave_status = CLOSED;  goto slave_whilestart150;
 /*140*/   when ?S_RTX_timeout !S_L2CA_TimeOutInd slave_status == 6 may slave_status = CLOSED;  goto slave_whilestart150;
 /*141*/   when ?S_ERTX_timeout !S_L2CA_TimeOutInd slave_status == 6 may slave_status = CLOSED;  goto slave_whilestart150;
}

mode slave_intervalzone157 metric<=0 {
 /*142*/   when metric>=0 may slave_status = W4_L2CA_CONNECT_RSP;  goto slave_whilestart150;
}

mode slave_intervalzone162 metric<=0 {
 /*143*/   when metric>=0 may slave_status = W4_L2CAP_CONNECT_RSP;  goto slave_whilestart150;
}


/************************************************************
*** Process [7] will be initialed here!
*/
mode S_RTX_whilestart243 true {
 /*144*/   when ?start_S_RTX true may metric = 0;  goto S_RTX_switchstart247;
 /*145*/   when ?disable_S_RTX true may  goto S_RTX_whilestart243;
}

mode S_RTX_switchstart247 metric<=60 {
 /*146*/   when ?disable_S_RTX true may  goto S_RTX_whilestart243;
 /*147*/   when !S_RTX_timeout metric>=60 may  goto S_RTX_whilestart243;
}


/************************************************************
*** Process [8] will be initialed here!
*/
mode S_ERTX_whilestart251 true {
 /*148*/   when ?start_S_ERTX true may metric = 0;  goto S_ERTX_switchstart255;
 /*149*/   when ?disable_S_ERTX true may  goto S_ERTX_whilestart251;
}

mode S_ERTX_switchstart255 metric<=60 {
 /*150*/   when ?disable_S_ERTX true may  goto S_ERTX_whilestart251;
 /*151*/   when !S_ERTX_timeout metric>=60 may  goto S_ERTX_whilestart251;
}


initially
    master_upper_whilestart1[1] and metric[1]==0
and master_whilestart21[2] and master_status[2]==0 and M_Buffer[2]==0 and M_Con[2]==0 and metric[2]==0
and M_RTX_whilestart114[3] and metric[3]==0
and M_ERTX_whilestart122[4] and metric[4]==0
and slave_upper_whilestart130[5] and metric[5]==0
and slave_whilestart150[6] and slave_status[6]==0 and S_Buffer[6]==0 and S_Con[6]==0 and metric[6]==0
and S_RTX_whilestart243[7] and metric[7]==0
and S_ERTX_whilestart251[8] and metric[8]==0

specification
  forall always( master_switchstart40[2] implies M_Con[2] == 0 )
;
```