Procedure-Level Verification of Real-time Concurrent Systems*

Farn Wang Chia-Tien Lo

Institute of Information Science, Academia Sinica, Taipei, Taiwan 115, Republic of China
+886-2-7883799 ext. 2420; FAX +886-2-7824814; farn@iis.sinica.edu.tw

Abstract. We want to develop verification techniques for real-time concurrent system specifications with high-level behavior structures. Nowadays, there is a big gap in between the classical verification theories and the engineering practice in real-world projects. This work identifies two common engineering guidelines respected in the development of real-world software projects, structured programming and local autonomy in concurrent systems, and experiments with special verification algorithm based on those engineering wisdoms. The algorithm we have adopted respects the integrity of program structures, treats each procedure as an entity instead of as a group of statements, allows local state space search to exploit the local autonomy in concurrent systems without calculating the Cartesian products of local state spaces, and derives from each procedure declaration characteristic information which can be utilized in the verification process anywhere the procedure is invoked. We have endeavored to implement our idea, test it against an abstract version of a real-world protocol in a mobile communication environment, and report the data.

1 Introduction

There is a great disparity between the engineering practice and the classical theories in verifying sophisticated computer systems. It is the goal of this work to investigate this disparity and experiment with verification techniques which combine the engineering wisdom with the classical verification theories.

Facing each nontrivial industrial problem human has encountered, there are in general two types of approaches used to surpass it and push the technology frontier forward. The first is engineering and the second is scientific. With the engineering approach, people strive to solve the problem using common wisdoms derived from their experiences in the field. With the scientific approach, people emphasize understanding the nature of the problem, by building mathematical models which simulate the problem, and design solution techniques based on

* The work is partially supported by NSC, Taiwan, ROC under grant NSC 85-2213-E-001-005 and by the Communication Technology Division, Computer & Communication Research Laboratories, Industrial Technology Research Institute, Taiwan, ROC under a new grant for 1995-1996.

the models. In the evolution of industry, these two approaches usually benefit from each other. On one hand, engineering wisdom reveals the true nature of the problem and inspires people to build better models. On the other hand, better understanding of the problem nature corroborates the engineering wisdom and may eventually lead to better solution.

As the computer systems we would like to build become more and more sophisticated, nowadays these two types of approaches are also employed in the task of system verification. On the engineering side, people have developed various engineering guidelines from their experience in the field and have been successful in constructing some really big real-world systems with complex high-level behavior [28, 18, 30]. Examples of those successes include network layer communication protocols, software systems with abstract data types, parallel databases. On the theoretical side, basic mathematical models have been proposed to help people better understanding the intrinsic nature, e.g. complexities, of the problem [1, 4, 9, 10, 12, 14, 15, 25, 31]. Indeed, it has been reported that the advancement in the classical theory has led to the successful verification of several small real-world products, including physical layer communication protocols[7] and integrated circuit design[4, 23].

But ironically, if we look at the common guidelines respected by computer system engineers, we find that they are very hard to mechanize and really do not fit into the classical verification theories. For example, in building sophisticated system, people adopt the guideline of *structured programming* to structurally divide the design into smaller functional parts like procedures and loops and to refrain themselves from using arbitrary connection among the parts. But in the classical theories, basic models are usually assumed to be equivalent to random graphs. If there is a procedure, the standard treatment is just to use it as a macro expansion regardless of its functional integrity.

Another example of the disparity regards the practice in using clocks. For engineers, clocks sometimes serve as convenient devices in simplifying interaction among different threads in concurrent systems. Here is a hypothetical example. A gentleman named Mike drives his friend Frank to a shopping mall and tells Frank that he will come back to pick him up at 5pm. In this case the interaction gets simplified to a number, i.e. the deadline to meet. Both of them do not care what the other party plans to do before 5pm as long as the deadline is met. But from the viewpoint of the verification theory, clock is really not so pleasant a device because it always blows up the worst case complexities by an exponential factor. Also interestingly enough, even clock readings are intuitively numbers, the most, if not the only, used property of clock readings in real-time system verification theories is that if you increment a clock reading by noninfinitesimal amount for enough number of times, it will eventually be bigger than any given finite constant.

The third example of the disparity concerns with the way people use concurrency. In the design of sophisticated systems involving the interaction of several parties, it is the common engineering wisdom to localize the design consideration so that the reliability and safety of the whole systems can be derived from the

verification of local properties of each party. But in the classical verification theory, calculation of Cartesian products of the local state spaces is usually adopted as the safe and complete technique. One exception is the composition and hiding operation proposed in process algebra [15, 25]. But still successful application to a real-world project with high-level behavior structure is yet to be observed.

This work integrates the two engineering guidelines cited in the above, i.e. structured programming and local autonomy in concurrent systems, into the design of verification algorithm which can be efficient in verifying well-designed computer systems. We target our research on real-time concurrent systems with timed atomic actions, synchronization primitives, procedures, loops, nondeterminism, and concurrency. A real-time system performs by giving out correct response at the correct moment. A concurrent system may allow several *threads*[11] (basic autonomous sequential executions) running concurrently. Recently, a theoretical framework for this purpose was proposed in [33] in which the verification complexities for both recursive and nonrecursive real-time concurrent systems are discussed and an algorithm is developed for the nonrecursive ones. One desirable feature of the algorithm is that it respects the procedure and loop structures of the systems by treating a rendezvousless execution of a procedure (loop) as a numerical entities, i.e. its execution time, which once analyzed, can be used in the verification process anywhere the procedure (loop) is invoked. Local autonomy of concurrent systems is utilized by a technique called *timing coincidence analysis* which determines the coincidibility of two states by telling if there are two synchronization-less local state sequences with the same execution time leading to the two states respectively. Such a technique enables us to construct global analysis from local state space search outcomes and is supposed to work well in systems with long autonomous executions and procedure invocations in between synchronization among the concurrent threads.

In this paper, we report the implementation of the algorithm proposed in [33]. However, we shall redesign the specification language to make it look like a traditional programming language. Several techniques to improve the average-case performance of the algorithm are incorporated which make the analysis of an abstract version of a general session setting control protocol (GSSC) in a wireless communication environment feasible. First, some related work will be discussed in section 2. We then formally introduce our new specification language and review the complexity issues in section 3. The reachability analysis algorithm proposed in [33] is then rewritten to fit our specification language in section 4. The important techniques we employed in the implementation are discussed in section 5. GSSC is discussed in section 6 with the performance on the reachability analysis of two states, one consistent and one inconsistent, reported. Section 7 discusses some extension to the implementation on the way and some future work.

We shall adopt the following notations. Given a set or sequence K, $|K|$ is the number of elements in K. We let $\mathcal{N}$ be the set of nonnegative integers, $\mathcal{Z}$ the set of integers, $\mathcal{N}^{\{\infty\}} = \mathcal{N} \cup \{\infty\}$, and $\mathcal{N}^{\{*\}} = \mathcal{N} \cup \{*\}$.

2 Related work

With the theoretical development of real-time and hybrid automata[1, 14, 21, 26, 27, 32] and the successful engineering of automated verification tools[2, 8, 4, 23], the research of computer-aided verification has received much attention. At this moment, various state-based [1, 4, 9, 10, 12, 14, 31] or event-based[20] model description languages are available, to which the standard verification technique of global space reachability analysis is usually applied. So far, several small real-world examples have been verified using this approach [4, 7, 23]. However such abstractions, although very elegant, may be at too low a level to make automatic verifiers efficiently uncover the behavior structures hidden in big system model descriptions.

Process algebra[15, 25] takes advantage of hiding and composition operators to construct complex systems from submodules. Since the verification algorithms for process algebra take care of general specifications without special program structure in mind, it may not be able to exploit the regularity of high-level behavior structure resulted from the observance of engineers to those guidelines.

People also use first-order logic or even higher-order logics to verify system designs. In those cases, very high-level behavior structures can be specified and verified[6, 13, 29]. For example, in [19], it was proposed to use positive cycles as intuitive refutation units in verifying real-time systems. The drawback is that the verification software can only do as much as proof checking and the engineers are pretty much left to their own. Still several remarkable benchmarks have been passed because of the industrious ingenuity of researchers in the community[6].

A good integration of engineering wisdom and scientific principles is formal methods in the line of VDM[18] and Z[30]. In that approach, a set of guidelines for system construction and a set of rules for verification are proposed and have been successfully applied in several real-world projects with benefits recorded. For example, in 1992, a Queen's Award of Britain was given to Oxford University Computing Laboratory and IBM UK Laboratories at Hursley Park for the development of IBM CICS using Z notation.

3 Real-time concurrent systems

3.1 Syntax

The underlying concept of our approach is DAG_1^1 *procedure*, defined in [33], which is a single-source single-destination directed acyclic graph in which each node represents a compound statement of procedure-call and atomic operation. Here we shall redefine DAG_1^1 procedure in the concept of traditional programming languages. As in the traditional imperative languages, a DAG_1^1 procedure is constructed from three types of statements, *timed atomic, rendezvous, procedure-calling*, and two types of *statement structures, switch, concatenation*. A timed atomic statement is executed with a prespecified earliest starting time and deadline. Unlike in Ada, here rendezvous is fulfilled by having all the participating parties executing the same rendezvous type at the same time. A procedure-call

represents either a fixed or a nondeterministic number of sequential execution of a DAG_1^1 procedure. A switch statement structure represents a nondeterministic choice, while a concatenation statement structure represents a successive execution of two statement structures. We use the following example to give intuition to the readers before the formal definition.

Example 1. In Figure 1, we illustrate three procedures, P, Q, R. P may loop nondeterministically many times to invoke R. Q invokes R also as the body of a 2-iteration loop. σ is a rendezvous type. An interval, like [4, 9], represents a

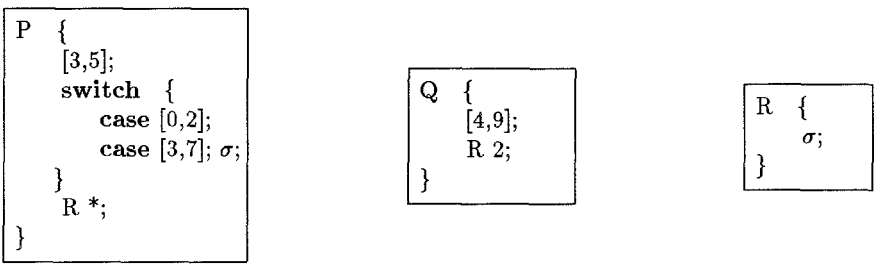

Fig. 1. Several simple procedures

timed atomic statement with the earliest starting times and deadlines. ||

Definition 1 : DAG_1^1 procedure. A DAG_1^1 *procedure* P is constructed from finite application of the following rules.

$$P ::= w\{B\}$$
$$B ::= [i,j]; \mid \sigma; \mid w_1 n; \mid w_1 *; \mid B_1 B_2 \mid \mathbf{switch}\{\mathbf{case}B_1 \mathbf{case}B_2 \ldots \mathbf{case}B_n\}$$

Here w, w_1 are procedure names (character strings) and B, B_1, B_2, B_n are statement structures. $i \in \mathcal{N}$, $j \in \mathcal{N} \cup \{\infty\}$, and n is a nonnegative integer. σ is a rendezvous type. Given $P = w\{B\}$, we shall use $\bar{P}$ as the notation for the name w of P. ||

Given a set Π of DAG_1^1 procedures, we treat each statement position as the true identity of the corresponding statement. That is given two statement positions which both execute a syntactically identically statement (say a rendezvous "σ;," or a procedure-calling "$\bar{P}n$;," or a timed atomic statement "$[i,j]$;"), we shall still treat these two positions as two different statements. Given a DAG_1^1 procedure P, we let $S(\bar{P})$ be the set of statements used in P. We also define $S_0(\bar{P})$ and $S_f(\bar{P})$ to be the set of first and final statements in P respectively in the following.

- If $\bar{P}\{B\}$ is defined, then $S_0(\bar{P}) = S_0(B)$
- If B is a statement, then $S_0(B) = \{B\}$.
- $S_0(B_1 B_2) = S_0(B_1)$.

- $S_0(\textbf{switch } \{\textbf{case } B_1 \ldots \textbf{case } B_n\}) = S_0(B_1) \cup \ldots \cup S_0(B_n)$

Similarly, we have the following definition for $S_f(\bar{P})$.

- If $\bar{P}\{B\}$ is defined, then $S_f(\bar{P}) = S_f(B)$
- If B is a statement, then $S_f(B) = \{B\}$.
- $S_f(B_1B_2) = S_f(B_2)$.
- $S_f(\textbf{switch } \{\textbf{case } B_1 \ldots \textbf{case } B_n\}) = S_f(B_1) \cup \ldots \cup S_f(B_n)$

Given a concatenation statement structure B_1B_2 in a DAG_1^1 procedure P, for each $s_1 \in S_f(B_1)$ and $s_2 \in S_0(B_2)$, we call s_2 a successor of s_1.

Notice that the looping in DAG_1^1 procedures are restrained as a special type of procedure calls. Conceptually, a real-time concurrent system allows many threads running concurrently. By giving additional information on the starting statement of each thread and the participating parties of each rendezvous type, a set of procedure definitions can be grouped to define a real-time concurrent system.

Definition2 : Real-time concurrent system. Given a set Π of DAG_1^1 procedures, we let Σ^{Π} be the set of rendezvous types used in procedures in Π. A *real-time concurrent system* is a tuple $\langle \Pi, \Omega, \tau \rangle$ satisfying the following properties.

- Π is a set of DAG_1^1 procedure definitions such that for every procedure P defined in Π, any procedure referenced in P is also defined in Π.
- Ω is a sequence $\langle \bar{P}_1, \ldots, \bar{P}_m \rangle$ of procedure names in $\overline{\Pi}$ and declares the m threads in the system. For each $1 \leq i \leq m$, thread i starts by invoking P_i.
- $\tau : \Sigma^{\Pi} \mapsto 2^{\{1,\ldots,|\Omega|\}}$ defines the set of parties participating in each rendezvous. For each $i \in \tau(\sigma)$, thread i is expected to participate in each instance of rendezvous σ.

A real-time concurrent system is *recursive* iff its procedure-calls are recursive. ||

Example 2. Assume we have the four procedure definitions in example 1. Then $R = \langle \{P, Q, R\}, \langle \bar{P}, \bar{Q} \rangle, \{\alpha \rightarrow \{1, 2\}\} \rangle$ is a legitimate nonrecursive and unambiguous real-time concurrent system. ||

3.2 An operational semantics for systems with single discrete clock

Given a real-time concurrent system with m threads, the *states* of the system can be described by an array of *local states* of the m threads. The procedure-calling scheme for the threads in our real-time concurrent systems resembles the push-pop operation of stacks[17] which have often been used as theoretical abstraction of nested procedure-callings. The *local state* of a thread can be conceptually recorded in a structure like the *control stack* (the name we shall adopt henceforth) in [3] and the *activation record* in [16, 34]. All possible executions of a thread from a moment can be deduced from the contents of the corresponding control stack.

Given a real-time concurrent system R, we let $S^{(R)}$ be the set of statements used in procedures defined in R. Given a thread local state (control stack) Γ represented as the following sequence

$$\begin{array}{lllll} \text{bottom} & & & & \text{top} \\ (s_0,t_0) & (s_1,t_1) & (s_2,t_2) & \ldots\ldots & (s_{m-1},t_{m-1}) \quad (s_m,t_m) \end{array}$$

we let (s_0,t_0) and (s_m,t_m) be the *bottom* and *top* respectively of Γ. A control stack, say the above-mentioned Γ, must satisfy the following conditions.

- For each $0 \leq i < m$, either
  - for some P and n, s_i is a procedure-calling statement "$\bar{P}n$;" with $0 \leq t_i < n$; or
  - for some P, s_i is a procedure-calling statement "$\bar{P}*$;" with $t_i = 0$.
- One of the following must be true for the top (s_m,t_m) of Γ.
  - s_m is a procedure-calling statement "$\bar{P}n$;" and $0 \leq t_m \leq n$.
  - s_m is a procedure-calling statement "$\bar{P}*$;" and $t_m = 0$.
  - s_m is a timed atomic statement "$[i,j]$;" and either $0 \leq t_m \leq j \neq \infty$ or $0 \leq t_m \leq i \vee t_m = j = \infty$.
  - s_m is a rendezvous statement "σ;" and either $t_m = 0$ or $t_m = \sigma$.

Given a control stack Γ, we let $\text{top}(\Gamma)$ symbolically denote the top of Γ. $\Gamma\gamma$ is a new control stack obtained by pushing γ into Γ. $\text{pop}(\Gamma)$ is a new control stack obtained by popping the top element from Γ. Given $\text{top}(\Gamma) = (s,t)$ with $t, c \in \mathcal{Z}$, we let Γ^{+c} be an abbreviation of $\text{pop}(\Gamma)(s, t+c)$, i.e. the local state obtained by incrementing the top counter value by c.

Suppose we are given a local state $\Gamma = (s_0,t_0)(s_1,t_1)\ldots(s_m,t_m)$. For each $0 \leq i \leq m$, when s_i is a procedure-calling statement "Pn;," it means the thread is now in the middle of executing procedure P and is going to invoke P consecutively for $|n - t_i|$ more times.

Similar to [17], we can define the succession of local states which follows the intuition of control stack evolution during procedure-calling and strongly matches the relation among paths in activation trees as discussed in [3]. However, for the convenience of our algorithm design, we shall present the following concept of local state successions.

Definition 3 : Succession of thread local states. The succession relation, $\vdash$, between local states are defined in the following way. Suppose we are given a local state Γ whose top is (s,t).

- **Fixed-loop procedure-call :** Suppose s is a procedure-calling statement "$\bar{P}n$;." If $t < n$, then for each $s_0 \in S_0(\bar{P})$, $\Gamma \vdash \Gamma^{+1}(s_0,0)$. If $t = n$, then for each successor statement s' of s, $\Gamma \vdash \text{pop}(\Gamma)(s',0)$.
- ***-loop procedure-call :** If s is a procedure-calling statement "$\bar{P}*$;," then for each $s_0 \in S_0(\bar{P})$, $\Gamma \vdash \Gamma(s_0,0)$ and for each successor statement s' of s, $\Gamma \vdash \text{pop}(\Gamma)(s',0)$.
- **Return from procedure-call :** When $s \in S_f(\bar{P})$, if one of the following three condition is true,
  - s is a timed statement "$[i,j]$" and $i \leq t \leq j$.
  - $t = \sigma$ form some rendezvous type σ.
  - s is a procedure-calling statement "$\bar{Q}n$;" and $t = n$.
  - s is a procedure-calling statement "$\bar{Q}*$;."

  $\Gamma \vdash \text{pop}(\Gamma)$

- **Timed statement :** Suppose s is a timed atomic statement "$[i, j]$;."
  - If $t < i$, then $\Gamma \vdash \Gamma^{+1}$.
  - If $i \leq t < j \neq \infty$, then $\Gamma \vdash \Gamma^{+1}$.
  - If $i \leq t < j = \infty$, then $\Gamma \vdash \text{pop}(\Gamma)(s, \infty)$.
  - If $i \leq t \leq j$, then for each successor statement s' of s, $\Gamma \vdash \text{pop}(\Gamma)(s', 0)$.
- **Rendezvous statement :** If s is a rendezvous statement and $t = 0$, then for each successor statement s' of s, $\Gamma \vdash \text{pop}(\Gamma)(s, \sigma) \vdash \text{pop}(\Gamma)(s', 0)$. ||

Based on the concept of local state succession, we are now ready to define the computation in multi-thread real-time concurrent systems.

Definition4 : States and runs. Suppose we are given a real-time concurrent system $R = \langle \Pi, \Omega, \tau \rangle$. A *state* of R is a sequence of $|\Omega|$ local states. A finite sequence $(\Delta_0, g_0)(\Delta_1, g_1) \ldots (\Delta_m, g_m)$ is called a Δ_0-*run* of R, where for each $0 \leq k \leq m$, Δ_k is a state and $g_k \in \{0, 1\}$ indicates the presence of a global clock tick. Assume, for each $0 \leq k \leq m$, $\Delta_k = \langle \Gamma_k^{(1)}, \ldots, \Gamma_k^{(n)} \rangle$. The following requirements are imposed on a Δ_0-run.

- For each $0 \leq k < m$ and $1 \leq i \leq n$, either $\Gamma_k^{(i)} \rightarrow \Gamma_{k+1}^{(i)}$ or $\Gamma_k^{(i)} = \Gamma_{k+1}^{(i)}$.
- **Enforcement of synchrony to global clock** : For each $0 \leq k < m$, $g_k = 1$ iff every thread increments its time reading by 1, that is for each $1 \leq i \leq n$ such that $\text{top}(\Gamma_k^{(i)}) = (s, t)$ and $\text{top}(\Gamma_{k+1}^{(i)}) = (s', t')$, $s = s'$ and either $t + 1 = t'$ or $t = t' = \infty$.
- **Enforcement of rendezvous** : For each $\sigma \in \Sigma^{\Pi}$ and each $0 \leq k \leq m$, if $\text{top}(\Gamma_k^{(i)}) = (s, \sigma)$ for some s and $i \in \tau(\sigma)$, then for each $j \in \tau(\sigma)$, $\text{top}(\Gamma_k^{(j)}) = (s', \sigma)$ for some s'.

Given a state Δ and a Δ-run $\Psi = (\Delta_0, g_0)(\Delta_1, g_1) \ldots (\Delta_m, g_m)$, for each $0 \leq k \leq m$, the time of the k-th state in Ψ, in symbols $time_\Psi(k)$, is defined inductively by two cases : (1) $time_\Psi(0) = 0$. (2) For each $0 \leq k < m$, $time_\Psi(k+1) = time_\Psi(k) + g_k$. ||

Example 3. Assume that we have the real-time concurrent system in example 2. It can be seen that while thread 1 may loop nondeterministically many times, thread 2 terminates after executing two instances of rendezvous α. Thus the whole system only has runs with two rendezvous instances. ||

3.3 Complexities of reachability analysis

We shall introduce a basic version of the reachability problem here. Such a version is instrumental in constructing other interesting versions.

Definition5 : State reachability. Given a real-time concurrent system $R = \langle \Pi, \Omega, \tau \rangle$, a state Δ' is said *reachable from* another state Δ in R iff there is a Δ-run $(\Delta_0, g_0)(\Delta_1, g_1) \ldots (\Delta_m, g_m)$ such that $\Delta' = \Delta_m$. ||

The reachability problem in our real-time concurrent systems can then be formulated in the following way. Given a real-time concurrent system R and two of its states Δ, Δ', the corresponding *state reachability problem* instance asks if Δ' is reachable from Δ in R. We cite the following theorem and lemmas from [33] to show the complexities of the problem for recursive and nonrecursive systems.

Theorem 6. *Two-counter machine halting problem[22] is reducible to the reachability problem of real-time concurrent systems.* ||

Lemma 7. *QBF[17] is reducible in PTIME to the state reachability problem for nonrecursive real-time concurrent systems.* ||

Lemma 8. *The state reachability problem of nonrecursive real-time concurrent systems is in PSPACE.* ||

4 Reachability analysis for nonrecursive systems

Our algorithm will be presented in two steps. First, we shall give a skeleton view of the algorithm in subsection 4.1 in which the implementation of one particular code line is not detailed. The skeleton describes how we exploit the autonomy of each thread in between hitting rendezvous to reduce the size of state space. Second, details about that one code line will be supplemented in subsection 4.2 and 4.3.

4.1 Skeleton view of the algorithm

We formalize the concept of thread autonomy in between hitting rendezvous with the following definition.

Definition 9 : Successor through rendezvous-less run. Given a real-time concurrent system R, and two states Δ and Δ', we say Δ' is a *successor through rendezvous-less run (or $\neg r$-successor)* of Δ iff there is a finite Δ-run $\Psi = (\Delta_0, g_0) \ldots (\Delta_m, g_m)$ of R such that

- Ψ ends at Δ', i.e. $\Delta' = \Delta_m$; and
- for each $0 < k < m$, Δ_k does not mark the completion of a rendezvous, that is, assuming $\Delta_k = \langle \Gamma_k^{(1)}, \ldots, \Gamma_k^{(n)} \rangle$, for each $1 \leq i \leq n$, there is no $\sigma \in \Sigma^{II}$ s.t. $\text{top}(\Gamma_k^{(i)}) = (s, \sigma)$ for some s. ||

In between hitting rendezvous, each thread executes in an independent way. We say a state $\langle \Gamma^{(1)}, \ldots, \Gamma^{(n)} \rangle$ is *at the stage of completion of rendezvous σ* iff for some $1 \leq i \leq n, s$, $\text{top}(\Gamma^{(i)}) = (s, \sigma)$. A major source of efficiency of our algorithm comes from the fact that we only work with states which are at the completion stage of rendezvous. The algorithm in table 1 takes this characteristic of real-time concurrent systems into consideration to answer instances of state reachability problem. Succession relation among such states is figured out by manipulation of arithmetic set expressions as defined in subsection 4.2 The correctness of procedure Reachable() has been proven in [33] and restated in the following lemma.

Lemma 10. *Given two states Δ, Δ' of a nonrecursive real-time systems R, with the oracle for $\neg r$-successorship, Δ' is reachable from Δ in R iff Reachable(R, Δ, Δ') is TRUE.* ||

Reachable(R, Δ, Δ')
/* $R = \langle \Pi, \Omega, \tau \rangle$, is a nonrecursive real-time concurrent system. */ {
(1) Generate the set X of all states in which a rendezvous is at the stage of completion.
(2) Determine the pairwise $\neg r$-successor relation in $X \cup \{\Delta, \Delta'\}$ and call it Y.
(3) If Δ' is reachable from Δ in $(X \cup \{\Delta, \Delta'\}, Y)$, answer TRUE; else answer FALSE.
}

Table 1. Algorithm for state reachability problem

All but the second code line in table 1 are obvious. In subsection 4.2, we shall introduce arithmetic set expressions, as the abstraction tool used to construct a solution for the second code line. In subsection 4.3, we shall use the technique of timing coincidibility analysis to determine the $\neg r$-successor relation between two states.

4.2 Arithmetic set expressions and their operations

The transitions in our system models are carried out within intervals bounded by earliest starting times and deadlines. Since earliest starting times, and deadlines alike, of a sequence of consecutive transitions can be accumulated during the analysis of thread behavior, it is natural to define the addition and integer multiplication of integer intervals. And indeed our reachability analysis algorithm is presented based on this kind of arithmetic set operations.

Our arithmetic set expression is constructed by the following rules.

$$H ::= \{c\} \mid H_1 \cup H_2 \mid H_1 \cap H_2 \mid H_1 + H_2 \mid H_1 k \mid H_1 *$$

c, k are natural numbers. $\{c_1, \ldots, c_n\}$ is a shorthand for $\{c_1\} \cup \ldots \cup \{c_n\}$. Conceptually, we treat an integer interval $[a, b)$ as a shorthand for the set $\{a, a+1, \ldots, b-1\}$. Especially, $[a, \infty)$ is a shorthand for $\{a\} + \{1\}^*$. The meaning of these set expression is inductively given in the following.

- Case $H = \{c\}$, H is the set of integer c.
- Case $H = H_1 \cup H_2$ ($H = H_1 \cap H_2$), H is the union (intersection) of H_1 and H_2.
- Case $H = H_1 + H_2$, $H = \{a + b \mid a \in H_1; b \in H_2\}$.
- Case $H = H_1 k$, (1) when $k = 0$, then $H = \{0\}$; (2) otherwise $H = H_1(k-1) + H_1$.
- Case $H = H_1 *$, $H = \bigcup_{k \geq 0} H_1 k$.

Note $\emptyset$ acts as a nullifier in arithmetic set addition, i.e. for all integer set expression H, $H + \emptyset = \emptyset$. Also, we allow distribution of addition against union and intersection.

An arithmetic set expression H is said to be in *periodical normal form (PNF)* iff $H = \bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)$. In the following, we give a set of rewriting rules to transform arithmetic set expressions into PNF arithmetic set expressions.

1) $\left(\bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)\right) \cap \left(\bigcup_{1\le j\le n}(\{b_j\} + \{d_j\}*)\right)$
$= \bigcup_{1\le i\le m;1\le j\le n} K_\cap(\{a_i\} + \{c_i\}*, \{b_j\} + \{d_j\}*)$
where $K_\cap(\{a\} + \{c\}*, \{b\} + \{d\}*)$ is defined by the following two cases.
   - If there is no integer solution i, j to $a + ci = b + dj$, then $K_\cap(\{a\} + \{c\}*, \{b\} + \{d\}*) = \emptyset$
   - Otherwise, let $\bar{i}, \bar{j}$ be the minimum integer solution. $K_\cap(\{a\} + \{c\}*, \{b\} + \{d\}*) = \{a + c\bar{i}\} + \{\mathrm{lcm}(c, d)\}*$

2) $\left(\bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)\right) + \left(\bigcup_{1\le j\le n}(\{b_j\} + \{d_j\}*)\right)$
$= \bigcup_{1\le i\le m;1\le j\le n} \left(\begin{array}{l} \{a_i + b_j + \mathrm{lcm}(c_i, d_j)\} + \{\gcd(c_i, d_j)\}*) \\ \cup \{a_i + b_j + c_i h + d_j k \,|h, k \in \mathcal{N}; c_i h + d_j k < \mathrm{lcm}(c_i, d_j)\} \end{array} \right)$

3) $\left(\bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)\right) k$
$= \begin{cases} \{0\} & \text{if } k = 0 \\ \left(\bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)\right)(k-1) + \bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*) & \text{if } k > 0 \end{cases}$
After application of the rule, rule 2 should be used immediately.

4) $\left(\bigcup_{1\le i\le m}(\{a_i\} + \{c_i\}*)\right)* = \sum_{1\le i\le m}(\{a_i\}* + \{c_i\}*)$
After application of the rule, rule 2 and distribution of addition against unions should be used iteratively to transform the formula to its PNF.

4.3 Timing coincidibility analysis

The technique of *timing coincidibility analysis* is based on the following observation. Given two concurrent threads starting their execution simultaneously, after running without interaction (rendezvous) for t time units according to the global clock, they may get to local states Γ, Γ' respectively. Then due to the strong synchrony in global clock systems, we can conclude that Γ, Γ' may happen at the same time during the two threads rendezvous-less executions respectively. This implies that we can separately work with the subtasks of searching in the local state space of each thread while analyzing the reachability between states. By figuring out the general time patterns between pairs of local states in the local state spaces, we can tell the $\neg r$-successor relations by intersecting those time representations. In this approach, time representations are often very concise since it tends to ignore the difference among different state sequences as long as they have the same time values.

The following definition formalizes the concept of local state space search.

Definition 11 : Local state sequence. Given a real-time concurrent system $R = \langle \Pi, \Omega, \tau \rangle$, a finite local state sequence $\Phi = \Gamma_0\Gamma_1 \ldots \Gamma_m$, with $\Gamma_k \vdash \Gamma_{k+1}$ for each $0 \le k < m$, defines a legitimate thread execution in R from Γ_0 and is called a Γ_0-*sequence*. Φ is *rendezvous-less* iff for every $0 < k < m$, Γ_k is not at the completion stage of a rendezvous, i.e. $\forall \sigma \in \Sigma^\Pi \forall s(\mathrm{top}(\Gamma_k) \neq (s, \sigma))$. ||

Definition 12 : Rendezvous-less time expressions. Given a DAG_1^1 procedure $\bar{P}\{B\}$, the *rendezvous-less time expression* of P, $texp_{\neg r}(\bar{P})$, is defined inductively as follows.

- then $texp_{\neg r}(\bar{P}) = texp_{\neg r}(B)$.
- $texp_{\neg r}([i,j];) = [i,j]$
- $texp_{\neg r}(\sigma) = \emptyset$
- $texp_{\neg r}(\bar{P}_1 n;) = (texp_{\neg r}(\bar{P}_1))n$
- $texp_{\neg r}(\bar{P}_1 *;) = (texp_{\neg r}(\bar{P}_1))*$
- $texp_{\neg r}(B_1 B_2) = texp_{\neg r}(B_1) + texp_{\neg r}(B_2)$
- $texp_{\neg r}(\mathbf{switch}\{\mathbf{case}B_1 \ldots \mathbf{case}B_n\}) = texp_{\neg r}(B_1) \cup \ldots \cup texp_{\neg r}(B_n)$

Suppose we are given a finite rendezvous-less local state sequence $\Phi = \Gamma_0 \ldots \Gamma_m$ with $\mathrm{top}(\Gamma_i) = (s_i, t_i)$ for each $0 \leq i \leq m$. We conveniently let $texp_{\neg r}(\Phi)$ equal to

$$\sum_{0 \leq i < m; s_i = [i,j]; 0 \leq t_i \neq \infty; 0 \leq t_{i+1} \neq \infty} (t_{i+1} - t_i) + \delta_\Phi * + \sum_{0 \leq i < m; s_i = \text{“}\bar{Q}*;\text{”}} ((texp_{\neg r}(\bar{Q}))*)$$

be our notation for the time expression for rendezvous-less local state sequence Φ. Here δ_Φ is $\{1\}$ when $\exists 0 \leq i \leq m (t_i = \infty)$; $\{0\}$ otherwise. Also we let

$$texp_{\neg r}(\Gamma, \Gamma') = \bigcup_{(\ \Phi \text{ is a finite simple rendezvous-less } \Gamma\text{-sequence of } R \text{ ends at } \Gamma'\ )} texp_{\neg r}(\Phi)$$

where "simple" means no two local states are the same. ||

The execution times of all rendezvous-less execution sequences between two local states can be figured out by doing some arithmetic on time expressions. The meaning of the time expression is given by the following lemma proven in [33].

Lemma 13. *Given two local states Γ, Γ' of a nonrecursive real-time concurrent system, there is a rendezvous-less execution sequence of time t from Γ to Γ' iff $t \in \mathrm{texp}_{\neg r}(\Gamma, \Gamma')$ which is computable.* ||

With definition 11, 12, and lemma 13, we have made the concept of autonomous execution of a single thread precise and proven our derivation of rendezvous-less thread execution time expression correct. Now all these can be readily combined to prove the correctness of the technique of timing coincidibility analysis.

Lemma 14. *Given $t \in \mathcal{N}$ and two states of a nonrecursive real-time concurrent system R, $\Delta = \langle \Gamma^{(1)}, \ldots, \Gamma^{(n)} \rangle$ and $\Delta' = \langle \Gamma'^{(1)}, \ldots, \Gamma'^{(n)} \rangle$, Δ' is a successor of Δ through rendezvous-less run of t time units in R iff $t \in \bigcap_{1 \leq i \leq n} \mathrm{texp}_{\neg r}(\Gamma^{(i)}, \Gamma'^{(i)})$*

Proof. According to definition 9, 11, 12, and lemma 13. ||

5 Implementation techniques

We have employed several techniques to take advantage of the behavior structure of procedure-callings and local autonomy to make the reachability analysis more efficient. As we have observed, such techniques are valuable and often result in orders of magnitude in verification performance improvement.

5.1 Nonpeak execution path

Given a stack Γ, remember that we use $|\Gamma|$ to denote the height of Γ. A local state sequence $\Gamma_0\Gamma_1 \ldots \Gamma_m$ is called a peak local state sequence iff there are integers i, j, k with $0 \leq i < j < k \leq m$ such that $|\Gamma_i| < |\Gamma_j|$ and $|\Gamma_j| > |\Gamma_k|$.

In an interval of the rendezvous-less execution of a thread, a peak in its execution sequence represents a procedure-call which does not incur any rendezvous. The effect of the rendezvous-less procedure-call can be treated as purely numerical values, i.e. its rendezvous-less time expressions. Thus we introduce the following new concept of local state succession involving encapsulated procedure-calls.

Definition 15 **Successors of local states with encapsulated procedure-call.** If s is a procedure-calling statement "$\bar{P}n$;" with $t < n$, then for each $1 \leq c \leq n-t$, $\Gamma \vdash \Gamma^{+c}$. ||

We have observed that while calculating the rendezvous-less time expressions between two local states, we only have to consider nonpeak path. While calculating the time expression of a rendezvousless local state sequence Φ between two local states, if we find out that two consecutive local states along Φ are connected by the above-defined successor relations, we shall include $(texp_{\neg r}(\bar{P}))c$ in $texp_{\neg r}(\Phi)$.

5.2 Starting timed atomic local states

While computing the time expression of rendezvousless local state sequence, we ignore those local states which mark the execution in the middle of timed atomic statements. That is we can ignore all local states Γ with $\text{top}(\Gamma) = (s, t)$, $s = [i, j];$, and $t \neq 0$ by introducing the following new local state successor relationship.

Definition 16 **Successors from starting timed atomic local states.** If $\text{top}(\Gamma) = (s, 0)$ and s is a timed statement $[i, j];$, then for each successor statement s' of s, $\Gamma \vdash \text{pop}(\Gamma)(s', 0)$. ||

Suppose we are given a rendezvousless local state sequence $\Phi = \Gamma_0 \ldots \Gamma_m$, if $\text{top}(\Gamma_0) = (s, t)$ and $\text{top}(\Gamma_m) = (s', t')$ with $s = [i, j];$, $s' = [i', j'];$, $t \neq 0$, and $t' \neq 0$. Let $\bar{\Phi}$ be the local state sequence identical to Φ except that

- $\bar{\Phi}$ starts t time units earlier than Φ in s; and
- $\bar{\Phi}$ ends t' time units earlier than Φ in s'.

Then we can compute $texp_{\neg r}(\Phi)$ to be $texp_{\neg r}(\bar{\Phi}) - [t, t] + [t', t']$. Note however that our original time expression definition does not deal with subtraction. But since we always reduce time expression to PNF, a time expression like $[a, b] + [c, c] * -[t, t]$ is equivalent to $[a + cg - t, b + cg - t] + [c, c]*$ where g is the smallest integer solution for x to $a + cx \geq t$.

5.3 Local state space segmentation

Given a real-time concurrent system with m threads, a naive approach in calculating states is to calculate all the Cartesian products of m local states. This usually results in huge number of pseudo states most of which can never be reached from the system initial state. The trick we use in the implementation is to segment the local state space of each thread by rendezvous local states. Thus given a state $\Delta = \langle \Gamma_1, \ldots, \Gamma_m \rangle$, while calculating the states reachable from Δ through rendezvous-less runs, we only compute the Cartesian product of the local state subspaces $K_1, \ldots, K_m$ defined in the following way. For each $1 \leq i \leq m$, if Γ_i is a rendezvous local state, then K_i is the next segment reachable from local states in the segment where Γ_i is in. Otherwise it is the present segment where Γ_i is in.

6 General Session Setup Control protocol

We have tested our implementation against an abstract version of a real-world project which deals with the communication link setup in a mobile phone environment. The test example GSSC comes from the Wireless Communication Department, Computer & Communication Research Laboratories, Industrial Technology Research Institute, Taiwan, ROC. It deals with setting up and later releasing a communication channel between a client and a server. Five threads are involved in the system, the client, the system service for client (SS_C), the line control unit (LCU), the system service form server (SS_S), and the server. The protocol must take care of communication failure incurred by, e.g. timeout, server busy, ...

In our experiment, we test two cases, one for the reachability of an inconsistent states and one for that of a consistent one. We thus give a brief description to all these five threads. After this, we shall then present part of the protocol in our description language.

- Reaction sequence from the client's viewpoint :
  1) The client starts the whole session by sending an SS_SETUP_req message to SS_C.
  2) If an SS_RELEASE_ind message is received from SS_C, quit the session. If an SS_FACILITY_ind message is received, send an SS_FACILITY_req message to SS_C to request for the facility and wait for further response from SS_C.
  3) If an SS_RELEASE_ind message is received from SS_C, quit the session. If an SS_FACILITY_ind message is received, then send an SS_RELEASE_req message to release the resources and quit.
- Reaction sequence from SS_C's viewpoint :
  1) When an SS_SETUP_req message is received from the client, send an LCU_DATA_req message with identifier 4 to LCU and starts a timer which timeouts at 10 time units later.

2) If an LCU_RELEASE_ind message is received from LCU within 10 time units, send an SS_RELEASE_ind message to the client and quit the session.
   If a timeout occurs, send an SS_RELEASE_ind message to the client and an LCU_RELEASE_req message with identifier 4 to LCU to quit the session.
   If an LCU_DATA_ind message is received from LCU within 10 time units, send an SS_FACILITY_ind message back to the client and wait for its further response.
3) If an SS_RELEASE_req message is received from the client, send an LCU_RELEASE_req message with identifer 4 to LCU to quit.
   If an SS_FACILITY_req message is received from the client, send an LCU_DATA_req message with identifier 4 to LCU and starts a timer which timeouts at 10 time units later.
4) If an LCU_RELEASE_ind message is received from LCU within 10 time units, send an SS_RELEASE_ind message to the client and quit.
   If a timeout occurs, send an SS_RELEASE_ind message to the client and an LCU_RELEASE_req message with identifier 4 to LCU to quit the session.
   If an LCU_DATA_ind message is received from LCU within 10 time units, send an SS_FACILITY_ind message back to the client and wait for its further response.
5) Upon the reception of an SS_RELEASE_req message from the client, send an LCU_RELEASE_req message with identifier 4 to LCU to quit the session.

- Reaction sequence from LCU's viewpoint :
  1) If an LCU_DATA_req message is received with identifier 4, send an LCU_DATA_ind message to SS_S. Go back to initial state.
  2) If an LCU_DATA_req message is received with identifier 132, send an LCU_DATA_ind message to SS_C. Go back to initial state.
  3) If an LCU_RELEASE_req message is received with identifier 4, send an LCU_RELEASE_ind message to SS_S. Go back to initial state.
  4) If an LCU_RELEASE_req message is received with identifier 132, send an LCU_RELEASE_ind message to SS_C. Go back to initial state.
- Reaction sequence from SS_S's viewpoint :
  1) When an LCU_DATA_ind message is received from LCU, if it is an SS_START message, then send an SS_SETUP_ind message to the server and wait its response; otherwise quit.
  2) If an SS_RELEASE_req message is received from the server, send an LCU_RELEASE_req message with identifier 132 to LCU to quit.
     If an SS_FACILITY_req message is received from the server, send an LCU_DATA_req message with identifier 132 to LCU and starts a timer which timeouts at 10 time units later.
  3) If an LCU_RELEASE_ind message is received from LCU within 10 time units, send an SS_RELEASE_ind message back to the server and quit.

If a timeout occurs, send an SS_RELEASE_ind message to the server and an LCU_RELEASE_req message with identifier 132 to LCU to quit. If an LCU_DATA_ind message is received from the LCU within 10 time units, send an SS_FACILITY_ind message to the server and wait for its further response.

4) If an SS_RELEASE_req message is received from the server, then send an LCU_RELEASE_req message with identifier 132 to LCU to quit. If an SS_FACILITY_req message is received from the server, send an LCU_DATA_req message with identifier 132 to LCU and starts a timer which timeouts at 10 time units later.
5) If an LCU_RELEASE_ind message is received from LCU within 10 time units, send an SS_RELEASE_ind message back to the client and quit the session.
6) If a timeout happens, send an SS_RELEASE_ind message to the server and an LCU_RELEASE_req messge with identifer 132 to LCU to quit.

- Reaction sequence from the server's viewpoint :
  1) Upon receipt of an SS_SETUP_ind message from SS_S, the server either sends an SS_RELEASE_req message back to SS_S if the service is not available and goes back to the initial state, or sends an SS_FACILITY_req message to SS_S to notify that the service is available and waits for further response from SS_S.
  2) Upon receipt of the an SS_RELEASE message, the server knows that the the service is no longer needed and go back to the initial state. On the other hand, if an SS_FACILITY_ind message is received from SS_S, the server may either send an SS_RELEASE_req message to SS_S in case the server has to abort the service, or an SS_FACILITY_req message to provide the service and wait for further response from SS_S.
  3) Upon receipt of the an SS_RELEASE message, the server knows that the the service has completed and goes back to the initial state.

In our specification, we treat each message as a rendezvous. Message to and from LCU with different identifiers are also treated as of different message types for convenience. Messages between the client and SS_C all begin with prefix SS_C_. Messages between the server and SS_S all begin with prefix SS_S_. In table 2 and 3. we list our specifications for thread LCU, and SS_C. Because of page limit, we shall not present the specification for the client, SS_S, and the server. Also in table 4, we list the thread and rendezvous type declaration.

We have performed two reachability analyses. One is for an unreachable state which says that the server has aborted the service while the client is still using the service. The total number of states generated is 3675 and the CPU time is 5075 seconds on a Sparc 10 clone.

The second is for a reachable state which says that the client is using the service provided by the server. The total number of states generated is 3676 and the CPU time is 6649 seconds on a Sparc 10 clone.

```
/* Line Control Unit */
LCU  {
    LCU_body *;
}

LCU_body  {
    [0,\infty]; /* NULL */
    switch {
        case LCU_DATA_req_132;    /* LCU_DATA_req */
             LCU_DATA_ind_132;
        case LCU_DATA_req_4;  /* LCU_DATA_req */
             LCU_DATA_ind_4;
        case LCU_RELEASE_req_4;
             LCU_RELEASE_ind_4;  /* LCU_RELEASE_ind */
        case LCU_RELEASE_req_132;
             LCU_RELEASE_ind_132;  /* LCU_RELEASE_ind */
    }
}
```

Table 2. Thread LCU, Line Control Unit

7 The challenge ahead

It is easy to see that given an unstructured real-time system specification, its timing behavior structure can be horribly difficult to analyze. But this kind of input assumption usually contradicts the common practice of structured programming in software engineering, the high-level semantics of programming languages, and the design rules in real-time systems. We advocate procedure-level model descriptions and verifications for real-time concurrent systems for their potentially better average-case performance in automated verification. At the current stage, we have implemented an verification system which respects and utilizes the high-level behavior structures demonstrated in the program structures and local autonomy of real-time concurrent system specifications. An abstract version of a real-world protocol is tested with performance reported. However, we are still looking for test examples which exhibit more program structures like nested procedure-calling.

The concept of statements with earliest starting times and deadlines was also adopted in formal frameworks like [20, 24]. Special techniques for reachability analysis has also been reported under this kind of framework[20, 5]. However, we do not know if there is any previous formal framework for real-time concurrent systems aimed at taking advantage of local autonomy and program structures to improve the verification performance.

```
/* Supplementary Service A */
/* Any service request to LCU is with identifier 4 */
SS_C  {
    SS_C_body *;
}

SS_C_body  {
    [0,\infty]; /* NULL */
    SS_C_SETUP_req;
    [0,0]; /* PD_TI := 4 */
    LCU_DATA_req_4;
    [0,0];  /* set timer = 10; Initiated  */
    switch {
    case  [0,10]; LCU_RELEASE_ind_132; SS_C_RELEASE_ind;
    case  [0,10]; LCU_DATA_ind_132; SS_C_FACILITY_ind; [0,\infty];
          switch  {
          case  SS_C_RELEASE_req;
                [0,0];  /* PD_TI := 4 */
                LCU_RELEASE_req_4;
          case  SS_C_FACILITY_req;
                [0,0]; /* PD_TI := 4 */
                LCU_DATA_req_4;
                [0,0]; /* set timer = 10 */
                switch  {
                case  [0,10]; LCU_RELEASE_ind_132; SS_C_RELEASE_ind;
                case  [0,10]; LCU_DATA_ind_132; SS_C_FACILITY_ind;
                      [0,\infty]; SS_C_RELEASE_req;
                      [0,0]; /* PD_TI := 4 */
                      LCU_RELEASE_req_4;
                case  [10,10]; SS_C_RELEASE_ind; LCU_RELEASE_req_4;
                }
          }
    case  [10,10]; [0,0]; SS_C_RELEASE_ind; LCU_RELEASE_req_4;
    }
}
```

Table 3. Thread SS_C, System Service for the Client

Acknowledgments

We thanks Professor Clarke and his crew at CMU for offering SMV[4, 23], which were used in our reachability analysis part of our implementation, and the associated technical advice. SMV is used as a routine for calculating the transitive closure in the rendezvous state graph in table 1.

```
/*=Thread description=================================================*/
<C, SS_C, LCU, SS_S, S>

/*=Rendezvous description=============================================*/

/* Between SS_C and LCU */
LCU_DATA_req_4 : 1,2;           LCU_RELEASE_req_4 : 1,2;
LCU_DATA_ind_132 : 1,2;         LCU_RELEASE_ind_132 : 1,2;

/* Between Client and SS_C */
SS_A_SETUP_req : 0,1;           SS_A_RELEASE_ind : 0,1;
SS_A_FACILITY_ind : 0,1;        SS_A_RELEASE_req : 0,1;
SS_A_FACILITY_req : 0,1;

/* Between SS_S and LCU */
LCU_RELEASE_req_132 : 2,3;      LCU_DATA_req_132 : 2,3;
LCU_RELEASE_ind_4 : 2,3;        LCU_DATA_ind_4 : 2,3;

/* Between Server and SS_S */
SS_B_SETUP_ind : 3,4;           SS_B_RELEASE_req : 3,4;
SS_B_FACILITY_req : 3,4;        SS_B_RELEASE_ind : 3,4;
SS_B_FACILITY_ind: 3,4;
```

Table 4. Thread and rendezvous type declaration

References

1. R. Alur, C. Courcoubetis, D.L. Dill. Model Checking for Real-Time Systems, IEEE LICS, 1990.
2. R. Alur, T.A. Henzinger, P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. in Proceedings of 1993 IEEE Real-Time System Symposium.
3. A.V. Aho, R. Sethi, J.D. Ullman. *Compliers - Principles, Techniques, and Tools*, pp.393-396, Addison-Wesley Publishing Company, 1986.
4. 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.
5. B. Berthomieu, M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE TSE, Vol. 17, No.3, March 1991.
6. Boyer, Moore. A Computational Logic Handbook, Academic Press, 1988.
7. D. Bosscher, I. Polak, F. Vaandrager. Verification of an Audio Control Protocol. Proceedings of Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, 1994; in LNCS, Springer-Verlag.
8. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., C-35(8), 1986.
9. E. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, Proceedings of Workshop on Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

10. E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems 8(2), 1986, pp. 244-263.
11. H.M. Deitel. An Introduction to Operating Systems, pp.110-115, Addison-Wesley, 1984.
12. E.A. Emerson, C.-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back, Science of Computer Programming 8 (1987), pp.275-306, Elsevier Science Publishers B.V. (North-Holland).
13. M.J.C. Gordon. HOL - A Proof Generating System for Higher-Order Logic. Cambridge University, Computer Laboratory, 1987.
14. T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic Model Checking for Real-Time Systems, IEEE LICS 1992.
15. C.A.R. Hoare. Communicating Sequential Processes, Prentice Hall, 1985.
16. E. Horowitz. Fundamentals of Programming Languages, Computer Science Press, 1984.
17. J.E. Hopcroft, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
18. C.B. Jones. Systematic Software Development using VDM, 2nd ed., Prentice Hall, 1990.
19. F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems, IEEE Transactions on Software Engineering, Vol.SE-12, No9, 1986, pp. 890-904.
20. F. Jahanian, D.A. Stuart. A Method for Verifying Properties of Modechart Specifications. IEEE RTSS 1988.
21. Y. Kesten, A. Pnueli, J. Sifakis, S. Yovine. Integration Graphs: a Class of Decidable Hybrid Systems. In Proc. Workshop on Theory of Hybrid Systems, LNCS 736, Springer-Verlag, 1993.
22. H.R. Lewis. Unsolvable Classes of Quantificational Formulus, 1979, Addison-Wesley Pub. Co.
23. K.L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, Boston, MA, 1993.
24. P. Merlin, D.J. Faber. Recoverability of Communication Protocols. IEEE Trans. Commun, Vol. COM-24, no. 9, Sept. 1976.
25. R. Milner. Communication and Concurrency. Prentice Hall, 1989.
26. O. Maler, Z. Manna, A. Pnueli. From Timed to Hybrid Systems. In Real Time : Theory in Practice, LNCS 600, pp. 447-484, Springer-Verlag, 1991.
27. Z. Manna, A. Pnueli. Verifying Hybrid Systems. In Proc. Workshop on Theory of Hybrid Systems, LNCS 736, Springer-Verlag, 1993.
28. R.S. Pressman. Software Engineering, A Practitioner's Approach. McGraw-Hill, 1982.
29. K. Slind. HOL90 Users Manual. Technical report, 1992.
30. J.M. Spivey. The Z Notation, A Reference Manual, second edition. Prentice Hall, 1992.
31. F. Wang, A.K. Mok, E.A. Emerson. Real-Time Distributed System Specification and Verification in APTL. ACM TOSEM, Vol. 2, No. 4, Octobor 1993, pp. 346-378.
32. F. Wang. Timing Behavior Analysis for Real-Time Systems. IEEE LICS 1995.
33. F. Wang. Reachability Analysis at Procedure Level through Timing Coincidence. in Proceedings of the 6th CONCUR, Philadelphia, USA, August 1995, LNCS 962.
34. W. Wulf, M. Shaw, P. Hilfinger, L. Flon. Fundamentals of Computer Science, Addison-Wesley, Reading, Mass., 1981.