A Technique for Verification of Race Conditions in Real-time Systems

Nagendar Telkar, Karam S. Chatha, Yann-Hang Lee, Gerald Gannod, and Eric Wong$^{\$}$

Department of Computer Science and Engineering, Arizona State University, Tempe, AZ, 85287-5406.
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083 $^{\$}$.

Real-time embedded systems can be differentiated from computation-intensive applications by their concurrent threads of control and time-dependent operations. The presence of concurrent threads acted upon by different event arrival instances makes them highly vulnerable to race conditions. The presence of race conditions introduces non-determinism in the system and alters its temporal properties. Thus, leading to potential violation of correct system behavior. Detection of race conditions in real-time systems is the focus of this paper. This paper presents a technique for detection of race conditions based on model checking. The inter-process communication mechanisms in the VxWorks environment have been designed and modeled as networks of timed automata by utilizing UPPAAL - a real-time verification tool. The various templates for the communication mechanisms have been verified for their correctness by utilizing the Timed Computation Tree Logic. The templates are then utilized to model the real-time application and verify the absence of race conditions. The paper presents examples that demonstrate the application of the proposed technique towards verification of race conditions.

1 Introduction

Real-time embedded systems can be differentiated from computation-intensive applications by their concurrent threads of control and time-dependent operations. The presence of concurrent threads acted upon by different event arrival instances makes them highly vulnerable to race conditions. The presence of race conditions introduces non-determinism in the system and alters its temporal properties. Thus, leading to potential violation of correct system behavior. This paper addresses the detection of race conditions in real-time systems.

There are two strategies to ensure the safety and reliability of the real-time system. One strategy is to employ engineering techniques like structured programming principles to minimize implementation errors, and then utilize testing techniques to uncover the errors in the implementation. The other strategy is to use formal analysis and verification techniques to ensure that the implemented system satisfies the safety constraints under all specified conditions. The first approach can only increase the confidence in the correctness of the system but the second approach can guarantee that a verified system always satisfies the safety properties.

We apply model-checking (a formal analysis technique) for verification of race conditions in real-time systems. Model checking is a technique for formal verification of finite-state concurrent systems. Timed automata is a finite-state automaton extended with a finite set of real-valued clock variables to represent time. Timed automata can be utilized to model the real-time system. Real time temporal logic or Timed Computation Tree Logic (TCTL) can then be utilized to verify the correctness of the system. Our goal is to determine, based on model checking, whether or not a given real-time application is free from races.

Race conditions are caused due to non-determinism in the inter-process communication (IPC) mechanisms. We utilize formal verification and timed automaton based models to statically detect race conditions. In our approach we first instrument the code and measure the execution time of application processes. As part of our approach we have developed detailed UPPAAL (a real-time verification tool) models for the various IPC mechanisms in VxWorks real-time operating system (RTOS). The application processes are modeled in UPPAAL with the pre-designed IPC templates. The non-relevant portions (from a race condition stand point) of the application code are abstracted away and modeled by a delay state that denotes their execution time. Designer specified TCTL formulas are then utilized to verify the presence of race conditions. The paper presents UPPAAL model templates for various VxWorks IPC mechanisms and verifies their correctness through TCTL formulas. The application of the models for verification of race conditions is demonstrated by experimental examples.

This paper is organized as follows: Section 2 gives the necessary background for race conditions and introduces UPPAAL, Section 3 discusses the related research on race condition detection, Section 4 explains the UPPAAL templates for VxWorks IPC and their verification, and Section 5 presents experimental examples of code segments modeled with our templates and their verification, and finally, Section 6 concludes the paper.

```
Thread 1

begin
{
        :
   amount = read_amount();
   SemTake(mutex);
   balance = balance + amount;
   interest = interest + rate*balance;
   SemGive(mutex);
        :
}
```

```
Thread 2

begin
{
        :
   amount = read_amount();
   if(balance < amount)
      printf("No Funds");
   else
   {
      balance = balance − amount;
      interest = interest + rate*bakance;
   }
          :
}
```

Figure 1: Program Segment illustrating Data Races

```
Thread 1                        Thread 2                        Thread 3
begin                           begin                           begin
{                               {                               {
   static int i = 0;               static int i = 0;               static int i = 0;
   static int A[] = {0,0,0};       static int A[] = {0,0,0};       static int A[] = {0,0,0};

   SemTake(mutex);                 SemTake(mutex);                 SemTake(mutex);
   i = read_value_i();             i = read_value_i();             i = read_value_i();
   A[i] = i;                       A[i] = i;                       A[i] = i;
   update_value_i;                 update_value_i;                 update_value_i;
   SemGive(mutex);                 SemGive(mutex);                 SemGive(mutex);
}                               }                               }
```

Figure 2: Program Segment illustrating General Races

2 Background

This section introduces race conditions and UPPAAL model checking environment.

2.1 Race Conditions

A race condition is defined as a situation in which multiple threads or processes read or write to a shared data object and the final result depends on the order of execution [1]. There are three conditions that must occur for a race condition to occur [2]:

- two or more processing elements have access to the same shared variable,
- at least one processing element is writing to the shared data item, and
- there is no mechanism in place to guarantee the temporal order of their access to that variable.

Race conditions are classified into data races and general races. A data race is defined as a race condition in which the accesses are not ordered by system visible synchronization or program order [3]. Explicit synchronization is added to shared-memory parallel programs to implement critical sections that are blocks of code intended to execute as if they were atomic. The atomic execution of critical sections can be guaranteed only if the shared variables that are read and modified by the critical section are not modified by any other concurrently executing section of code [4]. Data races are mostly a result of improper synchronization and can be removed by modifying the synchronization. Figure 1 illustrates a brief example of a data race. Thread 1's access of the shared variables *balance* and *interest* is well protected by the semaphores. But in thread 2, the shared variables *balance* and *interest* are not protected by any synchronization mechanism. Hence, when the thread 2 accesses and updates *balance*, the thread 1 may acquire the semaphore *mutex* and update the *balance*, and then thread 2 may calculate the *interest*. This certainly constitutes a data race.

A general race is a race condition in which the order of accesses to the shared resource is not enforced by the program's synchronization [5]. In such a case, the final outcome depends on the order of access to the shared resource by the tasks utilizing it. Such a race is more general than a data race. Such a race condition can be eliminated by imposing an order on the tasks in the way they access the shared resource. Figure 2 gives a brief example of a general race. One can see from Figure 2 that although the atomicity of the instructions is preserved, the order of accesses to the shared resource is not preserved. Though the program contains critical sections they can still be non deterministic. The system may interleave the threads in any order. That is, the order of execution of the threads can be 1, 2, 3 or 2, 3, 1 or any other combination of threads is a possibility. As a result, the array *A[]* can be either $\{1, 2, 3\}$ or $\{2, 3, 1\}$ based on the order of execution of the threads.

2.2 UPPAAL model checking environment

Real-time systems are computing systems that must react within precise time constraints to events in the environment. Timed automata [6] introduced in 1990 by Alur and Dill is a well-established timed extension of finite-state systems. It can be utilized to represent real-time systems as networks of timed automata by utilizing UPPAAL. UPPAAL [7] is an integrated tool environment for modeling, validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems that are modeled as networks of timed automata.

Global Declarations

Clock x;
Chan a;

Process Assignments

b1 := B(1);
b2 := B(2);

Private Declarations for B

int n;

System Definition

system A, b1, b2;

Parameters Passed to B

const me;

Figure 3: Declarations and Definitions

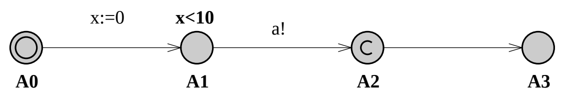

Figure 4: Snap shot illustrating the Process A

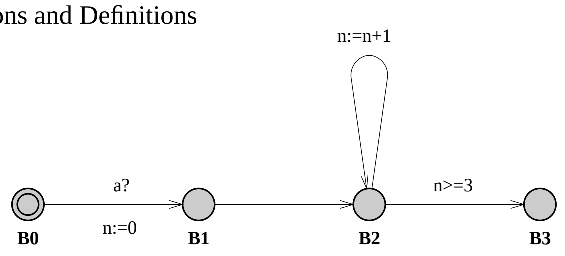

Figure 5: Snap shot illustrating the Process B

UPPAAL consists of three main parts: a graphical user interface to form the design, a simulator to simulate the system behavior, and a model checker to verify the correctness of the system. The graphical user interface allows the modeling of the system behavior in terms of networks of timed automata extended with data variables. The simulator and the model-checker are designed for the interactive and automated analysis of the system behavior by manipulating and solving constraints that represent the state space of the system description. Prior to verification, the simulator enables the user to examine the possible dynamic behavior of the system by its interactive simulation mechanism. The verifier is more formal, it needs Timed Computation Tree Formulas (TCTL) to verify a particular system behavior.

UPPAAL is based on timed automata, that is finite-state machine with clocks. Time is handled in UPPAAL by utilizing the concept of clocks. Time is continuous and the clocks measure time progress. Each transition is allowed to test the value of a clock or to reset it. Time will progress globally at the same pace for the whole system. A system in UPPAAL is composed of concurrent processes, each of them modeled as an automaton. The automaton has a set of *locations*. Transitions are utilized to change location. To control when to fire a transition, it is possible to have a guard and a synchronization.

The following sections describe the syntax and semantics of UPPAAL statements. A real-time system is considered to be a network of non-deterministic sequential processes communicating with each other over channels [7].

2.2.1 Syntax

As UPPAAL facilitates the modeling of real time systems, its modeling language is targeted to be as close as possible to a high level real-time programming language.

- **Guards**: Guards express conditions on the values of clocks and integer variables that must be satisfied in order for the edge to be taken. In Figure 5, the transition from *B2* to *B3* can occur only if the value of the corresponding integer variable *n* is greater than or equal to 3. Guards are conjunctions of timing and data constraints. A timing constraint is of the form: $x \sim n$ or $x - y \sim n$, where *n* is a natural number and $\sim \in \{ \leq, \geq, =, <, > \}$ a data constraint is of the form $i \sim j$ or $i - j \sim k$, with k being an arbitrary integer. The default guard of an edge is *true*.
- **Invariants**: Invariants specify how long the transition stays put in a particular location. An invariant specifies a progress condition. For example in Figure 4, the process is not allowed to stay in location A1 for more than 10 units of time.
- **Reset-Operations**: When the transition is taking an edge, the data or the clock variable may be subject to a simple manipulation. It may be reset to an expression or a constant. A clock variable can only be reset to a constant. Reset operation on a clock variable is basically of the form *x:=n* where *x* is a clock variable and *n* is the constant that is applied to it. In the Figure 4, the edge from the location *A0* to the location *A1* has got the clock reset operation on clock variable *x*. A data variable can be reset to an expression. In Figure 5, there is a reset operation on the integer variable *n* where it

Property	Name	Description	Equivalent to
E<>p	Possibly	there exists a path where p eventually holds	not E<> not p
A[]p	Invariantly	for all paths p always holds	
E[]p	Potentially always	there exists a path where p always holds	
A<>p	Eventually	for all paths p will eventually hold	not E[] not p
p-->q	Leads to	whenever p holds q will eventually hold	A[] (p imply A<>q)

Figure 6: UPPAAL Property Specification

is assigned to 0. Similarly, there is another reset operation on the location *B2* where the integer variable *n* is incremented by 1.

- **Channels, Synchronization, and Urgency**: A UPPAAL model consists of a network of timed automata. These automata communicate with each other via global integer variables or through communication channels. The global integer variables utilized are similar to shared memory variables. Figure 3 shows the declaration of the channel *a*. The synchronization mechanisms are blocking in nature. They are denoted by *a!* and *a?*. *a!* is the initiator of the synchronization, and *a?* supports it. If only the initiator *a!* is available or the supporter *a?* is available, the system does not advance till the other is available. The two transitions occur together. In Figure 5, the process B is blocked at *B0* till the process reaches the location *A1*, where it can do a *a!*. If a channel is declared to be urgent, the communicating components do not tolerate any delay. Whenever possible, they should be the next ones to execute. The corresponding edges may not have any guards on clocks.
- **Committed Locations**: A committed location is a location that must be left immediately. If a timed automaton has reached a particular location that is declared to be committed, the next transition in the system has to be from the location that is committed. If the transition involving the committed location is blocked, the system is blocked. In Figure 4, once the system has reached the location *A2*, and the system in Figure 5 be in any of the locations, the next transaction has to be *A2*-*A3* as *A2* is a committed location.

2.2.2 Semantics

A UPPAAL model determines the following two types of transitions between states.

- **Delay Transitions:** As long as none of the invariants of the control nodes in the current location are violated, time may progress without affecting the control node vector and all clock values are incremented with the elapsed duration of time. In Figure 4 and Figure 5, time may elapse 9 time units from the initial state ((A1,B0), x = 0, n = 0) leading to the state ((A1,B0), x = 9, n = 0). However, the clock cannot go beyond 10, if so, it would invalidate the invariant in the location *A1*.
- **Action Transitions:**
  - Two complimentary labeled edges of two different automaton can synchronize and proceed to the next state. Thus, in the state ((A1, B0), x = 9, n = 0) the automaton can proceed to next state to ((A2, B1), x = 9, n = 0).
  - If a component has an internal edge enabled, the edge can be taken without any synchronization. Thus, in state ((A2, B1), x = 9, n = 0), the process B can proceed to location B2 to the state ((A2, B2), x = 9, n = 0) without any synchronization.

2.2.3 Properties Specification

The UPPAAL model checker is designed to check for the invariant and reachability properties. It contains a verifier that can be utilized to verify certain properties by utilizing TCTL formulas. The Figure 6 specifies the semantics of the UPPAAL requirement specification language. The UPPAAL requirement specification supports five types of properties as shown in the Figure 6. The operators *Possibly* and *Potentially* are described as follows.

- **Possibly:**The property E$<>$p evaluates to true for a timed transition system if and only if there is a sequence of alternating delay transitions and action transitions s0$\rightarrow$s1$\rightarrow$...$\rightarrow$sn, where s0 is the initial state and sn satisfies p.

- **Potentially Always:**The property E[]p evaluates to true for a timed transition system if and only if there is a sequence of alternating delay or action transitions s0→s1→...→si→... for which p holds in all states si and which either:
  - is infinite, or
  - ends in a state (Ln, vn) such that either
    - for all d: (Ln, vn + d) satisfies p and Invariant(Ln), or
    - there is no outgoing transition from (Ln, vn)
- **State Properties:** The state properties specify the properties that are valid on a particular state. A state is represented as (L, v), where L is the location and v is the valuation of the clocks at a particular instant of time.
  - **Location:** Expressions of the form P.l where P is a process and l is a location, evaluate to true in a state (L, v) if and only if P.l is in location L.
  - **Deadlocks:**The state property deadlock evaluates to true for a state (L, v) if and only if for all $d \geq 0$ there is no action successor of (L, v+d).

3 Previous Work

A number of researchers have addressed the problem of verification of race conditions. Methods for the verification of race conditions can be categorized as follows [5]: static analysis of the program, dynamic analysis and post-mortem analysis of the program traces.

Ahead-of-time analysis: Ahead-of-time analysis or static analysis of race conditions is usually preformed during compile time. It tries to yield a high coverage by considering the space of all possible program executions and identifying race conditions that might occur in any of them. Static analysis methods [8], [9] report all the potential races and many of which may not really occur in real executions. Boyapati et al. [10] introduce a new static type system, which is a variant of ownership types, to detect both data races and deadlocks. Ownership types provide a statically enforceable way of specifying object encapsulation so that the lock protecting an object can also protect its encapsulated objects. [11], [12] and [13] present static techniques to detect race conditions by utilizing the properties of object-oriented languages. Static or ahead-of-time race condition detection involves overhead in terms of annotation cost and runtime performance, and they are prone to report false races to the user. Many of the race conditions that are detected statically do not take the actual system dynamics into consideration. Hence, many of the reported races do not actually occur when the system is run. Static analysis may be too difficult to implement in practice if the system happens to be too huge. Moreover, it is not always possible to gain accessibility to the source code of the program to be analyzed. However, some ideas of the static analysis can be combined with other dynamic approaches to inherit the best of both techniques.

Dynamic Analysis: While static or ahead-of-time race condition detection schemes concentrated on exploring all the possible races in a system, the dynamic or on-the-fly race condition detection techniques are concerned with precisely locating a race condition when it occurs during a particular execution of the system. The techniques presented in [14], [15] and [16] present dynamic race detection techniques by utilizing the Lamports ”happen-before” relation [17] to construct partial orders between critical events distributed among parallel processes, resulting in a partial order execution graph (POEG). Eraser [2], another dynamic race condition detector, utilizes binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behavior is observed. It utilizes a Lockset algorithm that enables itself to detect race conditions that are not apparent from a particular execution. Eraser also monitors shared memory locations directly instead of variables declared in programs, so that it can handle different types of programs, as long as the mutex lock synchronization is utilized. [18] presents an on-the-fly method. For a certain class of programs, a single execution instance is sufficient to determine the existence of an access anomaly for a given input when the proposed method is utilized.

The most significant problem of the on-the-fly detection of data races is that, they report only actual data races that occur in a particular execution of the program, but, with an overhead of space and time. A typical on-the-fly method incurs an overhead in the range of 3x to 30x to the original execution time. In addition to the runtime overhead, the detection intrusion may even cause the program to behave in a manner different from the original execution.

Post-Mortem Analysis: To avoid these problems, a series of post-mortem approaches have been developed, which usually combine an efficient record phase and a replay phase when the time-consuming race detection is performed. Post-mortem analysis of the execution traces detects just those data-races that occur during a particular execution in which the trace was generated. They may be utilized when a race cannot be verified statically. The source analyzer utilized in [9] is utilized to do the trace analysis. Incremental tracing [19] can be done to generate a coarse tree during runtime, and it can be expanded in replay for a detailed trace. Choi and Min introduced the concept of Race Frontier [16], which can be utilized to limit the number of entries in the access history of each shared variable and only report the latest entries involved in race condition. Techniques like RecPlay [14] are a combination of record/replay with automatic on-the-fly data race detection. This enables us to limit the record phase to the more efficient recording of the synchronization operations, while deferring the time-consuming data race detection to the replay phase.

The most significant problem of on-the-fly methods is the runtime overhead, in terms of time and space. In order for a dynamic approach to be followed, one has to instrument the code and let this instrumented code execute in its environment. But, instrumenting a real-time code would essentially change the mode of execution of the system. A real-time system cannot tolerate such a variation in its timing parameters. In addition to the runtime overhead, the detection intrusion may even cause the program to behave in a manner different from the original execution. Dynamic analysis of the system can only report the race conditions that occur in that particular execution of the system. They do not exhaustively investigate all the possible derivations of the program execution caused by different event arrival instances.

Comparison with our approach: We utilize formal verification and timed automaton based models to statically detect race conditions. In our approach we model the non-relevant (from a race condition verification standpoint) portions of the application by delay states. Hence, our approach can be categorized as a hybrid static and post-mortem analysis approach. We are distinguished from the previous work by the utilization of detailed timed automaton based models for the IPC mechanisms and race condition verification through TCTL formulas.

4 UPPAAL Models

We have designed and developed UPPAAL models for the various IPC mechanisms present in VxWorks real-time operating system (RTOS). Although, our models are specific for VxWorks RTOS they can be easily extended to other RTOS. This section discusses the models of the various IPC mechanisms and their verification by timed automata conditions. We describe the modeling of a real-time application as a network of timed automaton by utilizing our templates. We describe how TCTL can be utilized to detect race conditions in the UPPAAL models. The inter-process communication mechanisms that were modeled include binary semaphores, counting semaphores, and mutual-exclusion semaphores. Message queues were designed by utilizing the semaphores templates. The different options that were considered for queuing the semaphores included First-in-First-Out (FIFO) and Priority. WAIT, NO-WAIT, and Timeout mechanisms were considered to specify the semaphore waiting time for a process. Due to page limit restrictions, we limit the discussion to binary semaphores and message queues.

4.1 Semaphores

Each semaphore has three models associated with it, namely, the initialization model, the enqueing model, and the dequeuing model. Figure 7 specifies the required declarations necessary for a semaphore model. The Figures 8, 9, 10, 11, and 12 discuss the binary Semaphore (FIFO Option) that is used as a synchronization semaphore. Figures 8, and 9 specify the processes that give and take the semaphores respectively. Figures 10, 11, and 12 discuss the initialization, enqueing, and the dequeuing models respectively.

The declarations section shown in Figure 7 gives a list of all the global clocks, variables, constants, and channels. These declarations specify the necessary variables that are needed to keep track of the state of the semaphore. For example, the *fifoQ*, stores the *pids* of the processes that are blocked on the semaphore when the NO-WAIT option is set to 0. These declarations represent the internals of the VxWorks. To model semaphores or any other IPC mechanism in VxWorks, they have to be declared and initialized prior to their usage. To utilize our templates, the user must declare all the variables as shown in Figure 7 in the global declarations section, for each of the IPC mechanism.

The *sem* variable is the semaphore that is simulated as an integer value, and it is constrained to take only 0 or 1. If it is 1, it indicates that the semaphore is available and vice versa. Three different channels namely *semReq*, *semGive*, and *semTake* are utilized. The channel *semReq* is utilized to request a semaphore, *semTake* is utilized to send a signal to the process that is waiting to take the semaphore, and the channel *semGive* is utilized by the process that is giving the semaphore. The *noofProcesses* is the actual number of processes that utilize the semaphore. The *fifoQ* is the queue into which the process identities (*pid*s) are queued, when the semaphore is not available. The *fifoQ* is simulated as a circular queue whose maximum size is equal to the *noofprocesses*. The *fptr* specifies a pointer to the front of the queue, and *bptr* to the rear of the queue. The *tasksW* variable specifies the number of tasks that are waiting at any point of time for the semaphore. The *toRun* variable contains the *pid* of the next process that gets the semaphore. *NO_WAIT* is utilized to simulate the NO-WAIT and WAIT-FOR-EVER options

```
// declarations of global clocks, variables, constants and channels

int[0,1]    sem;                        // the actual semaphore

chan        semReq, semGive, semTake;   //signals to request, give and take semaphores

const       noofProcesses    3;         // number of user processes

int         fifoQ[noofProcesses];       // fifo queue of hold the waiting processes

int         fptr, bptr;                 // front and back pointers of the FIFOQ

int         tasksW;                     //number of  tasks waiting for the semaphore

int[1, 3]   toRun;                      //next process to run

int         pid;                        // indicates the process Id

const       NO_WAIT       0;            // wait or not on the semaphores

chan        noWait;                     // Channel utilized when NO_WAIT = 1
```

Figure 7: Declarations required for the FIFO binary synchronization semaphore

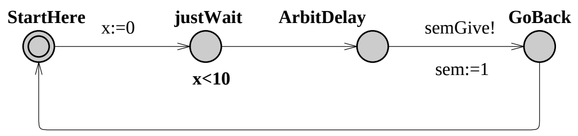

Figure 8: User ProcessB that gives the FIFO binary synchronization semaphore

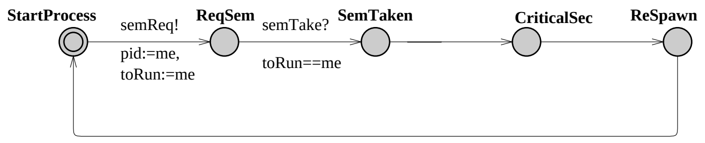

Figure 9: User ProcessA that takes the FIFO binary synchronization semaphore

of VxWorks. When *NO_WAIT* is set to 1, it indicates the NO-WAIT option of the VxWorks. When *NO_WAIT* is set to 0, it indicates the WAIT-FOR-EVER option. The *NO_WAIT* is initially set to 0. The channel *noWait* is utilized to support NO-WAIT and WAIT-FOR-EVER options.

4.1.1 Binary Semaphore

Figure 10 shows the initialization model of a binary semaphore, which is executed before any other model, as it has a committed start location. It initializes the front pointer *fptr* to 0, and the rear pointer *bptr* to -1, and creates the semaphore by setting *sem* to 1. The Figure 9 shows the user process that takes the semaphore via the channel *semReq*. Once a request is placed, the control is transferred to the location *Decide* in Figure 11.

- If a semaphore is available, it is signaled to the user via the channel *semTake*. The user process, that is at the location *ReqSem* gets the semaphore, and the semaphore is set to 0 by the enqueing model to make it unavailable.
- If the semaphore is not available,
  - the process is queued into the *fifoQ* using the *fptr*, and the process identity (*pid*),
  - the *tasksW* variable that denotes number of processes that are waiting is incremented by 1, and
  - the *fptr* is updated to point to the next location.

The Figure 8 shows the user process that is giving the semaphore via the channel *semGive*. It uses a clock variable *x*, and an invariant $x<10$ to simulate a delay in the location. It is done in order to regulate the process that is giving the semaphores. Once the semaphore is given,

- if *tasksw* == 0, that is, no tasks are waiting to take the semaphores, the sem is set to 1, and the control returns,
- if there are tasks waiting to take the semaphore, that is $tasksW>0$,
  - the task to run next is identified using the *bptr*,
  - the *toRun* variable is updated to contain the process that is to run next,

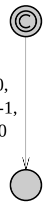

Figure 10: Initialization of a FIFO binary synchronization semaphore

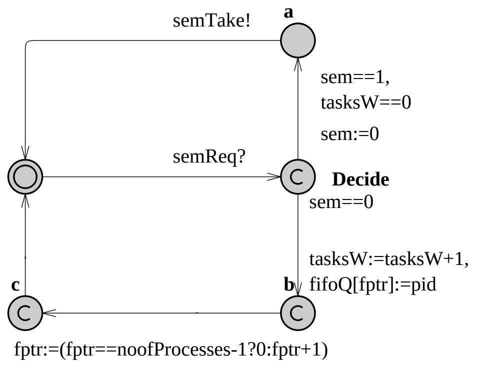

Figure 11: Enqueing of tasks waiting for the FIFO binary synchronization semaphore

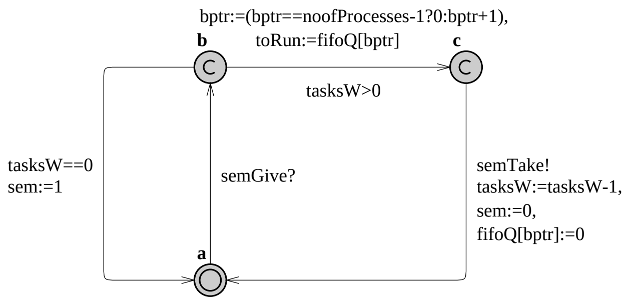

Figure 12: UPPAAL Model illustrating the dequeuing of tasks

1. A[] noofProcesses==3
2. process1.ReqSem −−> process1.SemTaken (Verified for all processes)
3. process1.StartProcess −−> process1.ReqSem (Verified for all processes)
4. process1.SemTaken −−> process1.CriticalSec (Verified for all processes)
5. process1.CriticalSec −−> process1.ReSpawn (Verified for all processes)
6. process1.ReSpawn −−> process1.StartProcess (Verified for all processes)
7. *A[] (process1.CriticalSec imply sem==0)* (Verified for all processes)
8. *A[] (process1.SemTaken imply sem==0)* (Verified for all processes)
9. A[] (process1.ReqSem or process2.ReqSem or process3.ReqSem imply sem == 0)
10. A[] ((fifoQ[0]==0 and fifoQ[1] ==3 and fifoQ[2] == 1 and bptr == 1 and DequeSems.c)
    imply (toRun==3)) (Verified for all processes)
11. A[] (((process1.SemTaken or process1.CriticalSec) and process2.ReqSem
    and EnqueSems.b and fifoQ[0] == 3 imply
    (fifoQ[0]==3 and fifoQ[1]==2)) or
    ((process1.SemTaken or process1.CriticalSec) and process2.ReqSem
    and EnqueSems.b and fifoQ[1] == 3 imply
    (fifoQ[1]==3 and fifoQ[2]==2)) or
    ((process1.SemTaken or process1.CriticalSec) and process2.ReqSem
    and EnqueSems.b and fifoQ[2] == 3 imply
    (fifoQ[2]==3 and fifoQ[0]==2)))
    (Verified for all processes)

Figure 13: TCTL verification conditions for the FIFO binary semaphore

– the tasks waiting(*tasksW*) is decremented by 1, and
– the waiting task is signaled via the channel *semTake*.

Figure 13 shows certain properties of the current model, expressed in TCTL. All the conditions are designed to pass the verifier test, except those shown in italics. *process1*, *process2* and *process3* are instantiations of the *processA*.

1. The first condition verifies that the number of processes in the system is always equal to 3. It makes sure that the system parameters are not modified by the model.
2. This condition verifies for starvation of the processes for the semaphores. It checks to see that, on all paths, if the *process1* has reached the location *ReqSem*, it will eventually reach the location *SemTaken*. It states that if *process1* has requested the semaphore, it will eventually get it. Similarly, it can be shown for all the other processes in the system.
3. On all the paths, if a process has reached the location *StartProcess*, it will eventually reach the location *ReqSem*.
4. On all the paths, if a process has reached the location *SemTaken*, it will eventually reach the location *CriticalSec*.

```
//Insert declarations of global clocks, variables, constants and channels.

urgent chan msgReq, msgGive, msgTake, msgPut, msgIsPut;          //signals to request, give and take semaphores

urgent chan msgReqPut;

const noofProcesses 2;                                           //Number of Processes

const buffSize 2;                                                //Message Q Size

urgent chan msgQFreed, msgRdy;

int[0,1] msgQ[buffSize];                                         //the msgQ that holds the message

int fifoQG[noofProcesses];                                       //FIFO Queue to hold the waiting processes Getting Msgs
int fifoQP[noofProcesses];                                       //FIFO Queue to hold the waiting processes Putting Msgs

int gfptr, gbptr;                                                //Front Pointer and Back Pointer of the FIFOQ

int mbptr, mfptr;                                                //Front and Back Pointer of the msgQ

int pfptr, pbptr;                                                //Front and back Pointers of the FIFOQP

int tasksWg;                                                     //Number of tasks waiting to get message
int tasksWp;                                                     //Number of tasks waiting to put message

int toRunG;                                                      //Next Process to Run
int toRunP;                                                      //Next Process to Run

int pid;                                                         //indicates the Process Id

const NO_WAIT_G 1;                                               //Option to set the block/unblocking property
```

Figure 14: UPPAAL Model illustrating the declarations necessary for a Message Queue

5. On all the paths, if a process has reached the location *CriticalSec*, it will eventually reach the location *ReSpawn*.

6. On all the paths, if a process has reached the location *ReSpawn*, it will eventually reach the location *StartProcess*.

   The conditions 2, 3, 4, 5, and 6 check for the absence of deadlocks in each of the process models. They can be extended to all the other processes.

7. This condition fails because, when a process is in the critical section, the process giving the semaphore may give another semaphore and set the *sem* variable to 1.

8. This condition fails, because if a process has taken the semaphore, the *processB* that gives the semaphore, may give another semaphore making the *sem* to be equal to 1.

9. This describes the situation where the *processB* has not yet started giving the semaphores. It also describes a case where each of the processes has consumed the semaphore, and has come back to request another semaphore, even before *processB* generated one.

10. If the queue is filled with the *pid*s of *process3*, and *process1* in the 1st and 2nd locations respectively, the next process to run would be the process with the *pid* equal to 3, as the *bptr* always points to the front of the queue. This checks the dequeuing model for its correctness. This condition is verified for all processes.

11. This condition checks the enqueing model. It checks to see that, if a semaphore is in use, and a process is in the *fifoQ* blocked on the semaphore, in such a case, any other process requesting the semaphore is queued into the next location in the *fifoQ*.

4.2 Message Queues

Message queues have been developed by utilizing the semaphore models. A message queue consists of a queue into which the messages are placed. If the message queue is full, any task trying to write to the message queue has to be blocked till a next free location is available in the queue. If the message queue is empty, any task trying to read the message queue is blocked till at-least one location in the queue is filled up. To simulate message queues we need three queues, one to store the messages, one queue to store the writer processes that are blocked, and one queue to store the reader processes that are blocked.

The required declarations for a message queue are shown in Figure 14. The channels *msgReq*, *msgGive*, *msgTake*, *msgPut*, *msgReqPut*, and *msgIsPut* are utilized to request, write, and read the messages. The *noofProcesses* is the number of processes

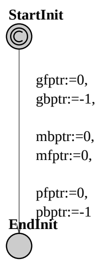

Figure 15: Initialization of a Message Queue

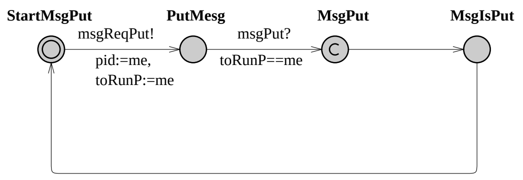

Figure 16: UPPAAL Model illustrating a writer task in the Message Queue Models

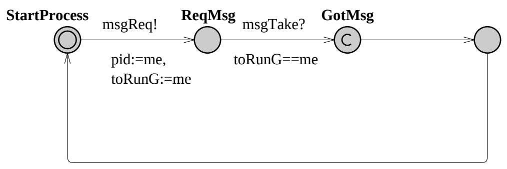

Figure 17: UPPAAL Model illustrating a reader task in the Message Queue Models

that are currently active in the system. *msgQ* is the queue to hold the messages. *mbptr* and *mfptr* are the rear and front pointers of the message queue *msgQ*. The *fifoQG* is the FIFO queue to hold the processes that are blocked, and waiting to read the messages from the message queue. *gbptr* and *gfptr* are the rear and front pointers of the *fifoQG*. The *fifoQP* is the FIFO queue to hold the processes that are blocked, waiting to write messages to the message queue. *pbptr* and *pfptr* are the rear and front pointers of the *fifoQP*. The *tasksWg* indicates the number of threads that are waiting to get the messages from the message queue. It indicates the total number of reader threads that are blocked. The *tasksWp* indicates the number of threads that are waiting to put the messages to the message queue. It indicates the total number of writer threads that are blocked. *toRunG* contains the *pid* of the reader process that is due to run next. *toRunP* contains the *pid* of the writer process that is due to run next. The NO-WAIT variable can be utilized to set the NO-WAIT or WAIT-FOR-EVER option.

The initialization model for a message queue is shown in Figure 15. It initializes the front and the rear pointers for all the queues utilized, namely *msgQ*, *fifoQG* and *fifoQP*. The user process that reads a message from the message queue is shown in Figure 17. The user process that writes a message to the message queue is shown in Figure 16. The process in Figure 17 initially makes a request for the message via the channel *msReq*. The control is now transferred to the model in Figure 19 to the location *Decide*. Now, if the message queue is not empty,

- The control is transferred to the location *Updatembptr*.
- The message is read from the message queue, and the corresponding location in the *msgQ* is set free by setting it to 0.
- The *mbptr* is made to point to the next available location.
- The user process is signaled via *msgTake*, and the control returns to the user process in Figure 17.
- When at the location *Decide* in Figure 19, as the data is read, a slot is set free in the *msgQ*. So, any thread that is waiting to write to the shared location has to be now signaled, so that it can write to the new location that was set free. This is done via the channel *msgQFreed*.
- Through the channel *msgQFreed*, the control gets transferred to the location *Decide* in Figure 20. Now,
  - If the *tasksWp* is equal to 0, that is there are no blocked writer threads, the control just returns.
  - If *tasksWp* is greater than 0, that means there are blocked writer tasks.
    * One among these set of blocked threads has to signaled. This is done via the channel *msgPut*.
    * The next thread to run is identified by the *pbptr*, and the *toRunP* variable is updated to contain the process id or *pid* of the next writer process to run.
    * The message queue is now written to by making use of the *mfptr* after which, *mfptr* itself is updated.
    * The number of writer tasks that are waiting, that is *tasksWp* is decremented by 1.

  Alternatively, if the message queue is empty,

  - Control is transferred to the location *Updategfptr*.
  - The reader process is queued in the *fifoQG* utilizing its *pid*, and the *gfptr* is updated.
  - The *tasksWg* is incremented by 1, to denote that there exists a reader process that is blocked in the queue *fifoQG*.

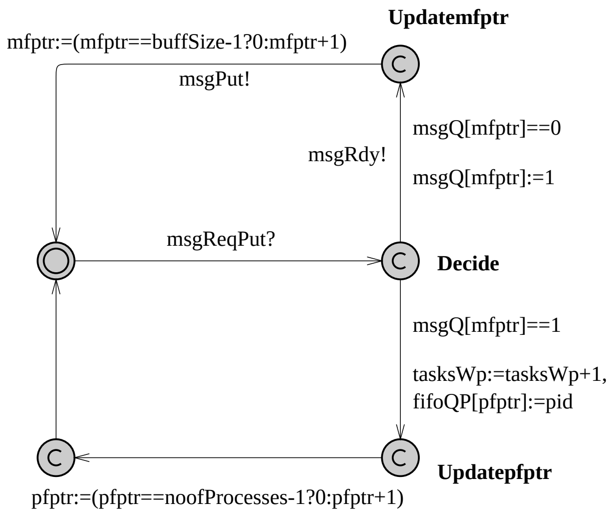

Figure 18: Enqueing model for writer tasks

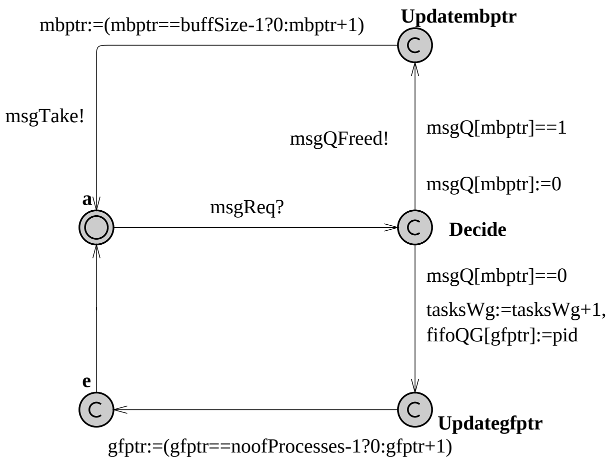

Figure 19: Enqueing model for reader tasks

The user process that writes a message to the message queue *msgQ* is shown in Figure 16. It initially makes a request to place a message in the message queue via the channel *msgReqPut*. The control is now transferred to Figure 18 to the location *Decide*. Now,

- if any of the slots in the message queue are free,
  - Control is transferred to the location *Updatemfptr*.
  - the message is placed in the message queue and the *mfptr* is updated.
  - At the same time any task that is waiting for the to read the message has to be signaled via *msgRdy*.
  - When signaled via the channel *msgRdy*, the control is transferred to Figure 21.
  - Utilizing the *gbptr* and the *fifoQG*, the next reader process is decided, and signaled via *msgTake*.
- if the message queue is filled up, the writer process is queued up in the *fifoQP*, and the *pfptr* is updated.

Figure 22 shows the timed automata verification conditions that are relevant for the Message Queue. The following paragraph describes the timed automata conditions shown in Figure 22. The number on the left specifies the serial number of the timed automata condition that is shown in Figure 22. All the conditions are designed to pass the verifier test for satisfiability, except those shown in italics.

1. It checks for absence of deadlocks. The model has to be free from deadlocks to make sure that it runs uninterrupted.
2. This condition checks for the case where, the message queue has got empty slots, but the writer processes have got queued into the *fifoQP*. This is an error condition.
3. This condition checks for the case where the message queue has got messages, but the reader processes have got queued in the *fifoQG*.
4. Whenever the *fifoQP* is got tasks that are waiting for the slots in the message queue, it implies that message queue is full.
5. The absence of messages in the message queue implies that, any task that wishes to write a message can write to the queue without getting blocked in the *fifoQP*. So, for all the paths, whenever the message queue is empty, the *fifoQP* is empty.
6. Whenever, a message is present in the message queue, the reader processes can read the message without blocking in the *fifoQG*. So, whenever there exist any messages in the message queue, the *fifoQG* is empty.

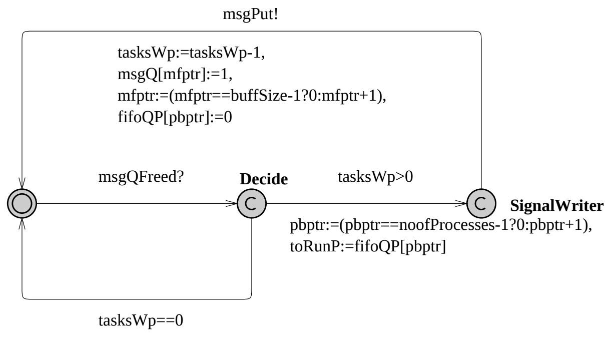

Figure 20: Dequeuing model for writer tasks

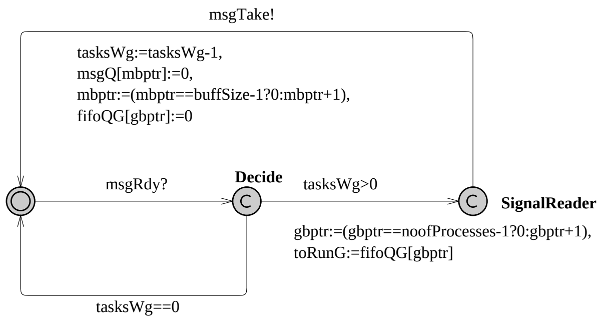

Figure 21: Dequeuing model for reader tasks

1. A[] not deadlock
2. *E<> ((msgQ[0]==0 or msgQ[1]==0) and (fifoQP[0]==1 or fifoQP[1]==1))*
3. *E<> ((msgQ[0]==1 or msgQ[1]==1) and (fifoQG[0]==1 or fifoQG[1]==1))*
4. A[]((fifoQP[0]==1 or fifoQP[1]==1) imply (msgQ[0]==1 and msgQ[1]==1))
5. A[] ((msgQ[0]==0 and msgQ[1]==0) imply (fifoQP[0]==0 and fifoQP[1]==0))
6. A[] ((msgQ[0]==1 or msgQ[1]==1) imply (fifoQG[0]==0 and fifoQG[1]==0))
7. A[] (fifoQG[0]==2 and fifoQG[1]==1 and msgQ[0]==1 and gbptr == 0 imply toRunG==2)
8. A[] (fifoQP[0]==4 and fifoQP[1]==5 and pbptr == 0 imply toRunP==4)
9. A[] (msgQ[0]==1 and msgQ[1]==1 and mbptr == 0 and MsgGet.Updategfptr imply msgQ[0]==0)

Figure 22: Timed Automata verification conditions for the Message Queue

7. If the *process2* is queued in the *fifoQG[0]*, and the *process1* in the *fifoQG[1]*, and the *msgQ[0]* is 1, the *gbptr* is 0, the next reader process to run would be *process2*. This condition makes sure that the dequeuing model at the reader end works as intended. The condition can be checked for other combinations as well.

8. It is similar to the previous condition, but it is for writer processes, so it checks the *fifoQP*. It makes sure that the dequeuing mechanism at the writer end works as intended.

9. If the *msgQ* is filled in the locations 0 and 1, then once the message is read at the location *Updategfptr* in Figure 19, the *msgQ[0]* is read first. This follows the FIFO property of the message queue.

All the above mechanisms are made available as templates so that the user can use them to construct his own models from these templates.

5 Experimental examples

This section illustrates experimental examples that have been modeled to illustrate the various types of race conditions that can occur when two different threads communicate with each other. The corresponding verification mechanisms have also been shown.

5.1 Example 1

Figure 23 illustrates a situation in which two threads shake hands by utilizing binary semaphores and exchange messages. The threads utilize two binary semaphores *A* and *B* that are initially empty. So each of them gives a semaphore signaling the other thread to proceed. The current thread blocks till it gets the semaphore that is given by a thread in the other group. It is only after taking the semaphore can the thread access the shared data item.

There are various race conditions that can occur in the example in Figure 23. Let *Thread-A* give the semaphore *B*, and then wait to take the semaphore *A* at line 4. Once it gets the semaphore *A*, it may go ahead and update the shared location *Buf-A*. It may

```
     Binary Semaphore A = 0, B = 0;
     int         Buf_A, Buf_B;

     Thread_A(...)                          Thread_B(...)
     {
1        int Var_A;                             int Var_B;

2        while(1)                               while(1)
         {                                      {
           .....                                  .....
           Var_A = ...;                           Var_B = ...;
3          SemGive (B);                           SemGive (A);
4          SemTake (A);                           SemTake (B);
5          Buf_A = Var_A;                         Buf_B = Var_B;
6          Var_A = Buf_B;                         Var_B = Buf_A;
           ...                                    ...
7        }                                      }

     }                                      }
```

Figure 23: Example 1 illustrating Data Races

```
1. A[] not deadlock

2. E<> (Thread_A1.SemTaken and checkA == 1)

3. E<> (Thread_A2.SemTaken and forA == 1)
```

Figure 24: UPPAAL verifier model Thread-A, Example 1

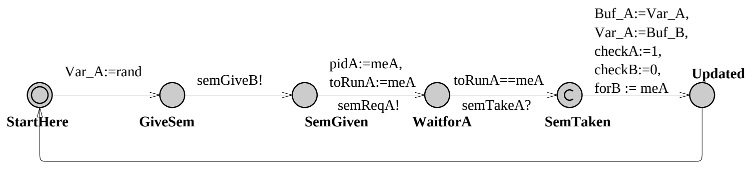

Figure 25: UPPAAL model for Thread-A, Example 1

later proceed to read the value from the shared location *Buf-B* even before *Thread-B* updates it. This situation is due to the lack of synchronization between the different thread groups.

There is another potential race condition which may occur due to the lack of mutual exclusion mechanism among the threads of each group. It may result when two threads in group A try to exchange messages with the same thread in group B. Consider the threads A1, A2, B1 and B2. Initially A1 and B1 each give the binary semaphore *B* and *A* respectively. Once the semaphores are available, the threads A1 and B1 may proceed to take A and B semaphores doing a *SemTake* at line 4. Threads A1 and B1 are yet to execute line 5, at which point they may be swapped out. Now, thread A2 of thread group thread-A may get the time slice, and it may give a Semaphore *B*, and wait for semaphore *A* at line 4. Thread A2 is now swapped out, thread B2 may get the time slice to run. Thread B2 may now give the semaphore *A*, and swap out at line 4, allowing the thread A2 to take it. Thread B1 may now resume, and update the shared location at line 6. Thread A1 may go over to line 6, and access the shared location to read the value set by B1, after which, the thread A2 may go ahead and read the same location. Thus, thread A2 is reading a value set by the thread B1 for the thread A1. This is a typical case of a race condition where the threads are not synchronized properly.

Figure 25 shows the UPPAAL model of the *thread-A* along with the code inserted into the model. Figure 24 shows the set of race conditions expressed in timed automata. The first condition in Figure 24 checks all the possible paths to see if the model is free from deadlocks. Figure 25 also shows the code that is inserted in the model. The variables *checkA*, *checkB*, *forA*, and *forB* are utilized to instrument the model. Whenever *Buf-B/A* is read, *checkB/A* is reset. Whenever *Buf-B/A* is written to, *checkB/A* is set. So if the *thread-A* has reached the location *SemTaken*, and it finds *checkA* is equal to 1, it implies that it is over-writing the shared location *Buf-A*, even before it is read. The second timed automata condition checks for this race condition. It tries to find if there exists a path, wherein *thread-A1* has reached the location *SemTaken*, and *checkA* is still set to 1, that implies, the thread is over writing the buffer before it is read. The third timed automata condition verifies the occurrence of the race condition where, a thread reads the value that is set for another thread. At any point of time, *forA* and *forB* contain the process identity of the process which has to access the data from the buffer. Hence, this condition checks to see if there exists a path in which *thread-A2* is trying to read the buffer that is set for *thread-A1*.

5.2 Example 2

In the example in Figure 23, we have seen that we cannot prevent the threads in the same group from rushing in and overwriting the existing message before it is taken. The current example shown in Figure 26, utilizes two different semaphores to force a thread in group A to wait until a thread in group B completes its task.

```
     Binary Semaphore Aready = 1, Bready = 1;
     Binary Semaphore Adone = 0, Bdone = 0;
     int         Buf_A, Buf_B;

     Thread_A(...)
     {
          int Var_A;

1         while(1)
          {
             .....
2            SemTake (Bready);
3                Buf_A = Var_A;
4                SemGive (Adone);
5                SemTake (Bdone);
6                Var_A = Buf_B;
7            SemGive (Aready);
8            ...
          }
     }

     Thread_B(...)
     {
          int Var_B;

          while(1)
          {
             .....
             SemTake (Aready);
                 Buf_B = Var_B;
                 SemGive (Bdone);
                 SemTake (Adone);
                 Var_B = Buf_A;
             SemGive (Bready);
             ...
          }
     }
```

Figure 26: Example 2 illustrating Data Races

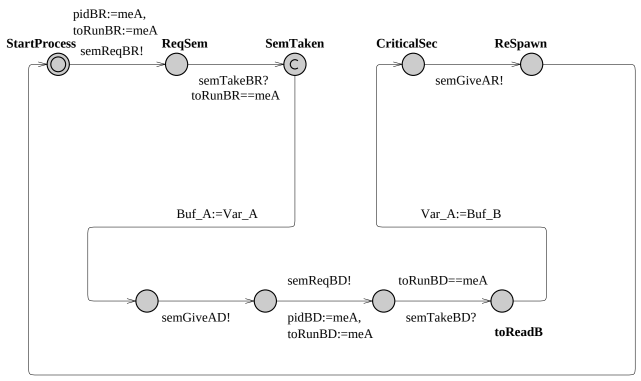

Figure 27: UPPAAL model for Thread-A, Example 2

1. A[] not deadlock
2. E<> (Thread_A2.toReadB and forA == 1)

Figure 28: UPPAAL verifier model Thread-A, Example 2

It utilizes four binary semaphores, two for each thread group, of which two are initially full, and two are empty. Initially the semaphores *Aready* and *Bready* are full, so the threads *thread-A1* and *thread-A2* can take them and proceed to update the buffers *Buf-A* and *Buf-B* at line 3. Once the buffers are updated, each can give the binary semaphore *Adone* and *Bdone* signaling the thread in the other group to read the value set at line 3. The semaphores *Adone* and *Bdone* are taken at line 5 by *thread-B* and *thread-A* respectively. Once the value is read at line 6, the threads can give the semaphore at line 7, and allow any other thread that is waiting at line 2 to enter the critical section.

This mechanism prevents threads in the same group from rushing in, and overwriting the existing message before it was read. If a thread has to reach line 3 to write to the shared variable, it must take the semaphore at line 2, but that semaphore is given by thread in the other group only when the value in the shared buffer is read by it. So a thread can write to the shared buffer only when the buffer has been read.

The current example however does not prevent the case in which the message meant for one thread is read by another thread in the same thread group. For example, if two threads A1 and A2 are waiting at line 2 for the semaphore *Bready*, when the thread B1 gives the semaphore any one of them may take the semaphore. Since there is no ordering among the threads to access the shared data item it is a general race condition. In VxWorks this situation is avoided by utilizing FIFO or priority options with the semaphores.

Figure 27 illustrates the UPPAAL model of the example in Figure 26. It utilizes two variables *forA* and *forB* to do the instrumentation similar to the example in Figure 27. The race conditions are illustrated in Figure 28.

6 Conclusion

We presented a technique for verification of race conditions in real-time systems. We modeled the various IPC mechanisms in VxWorks as timed-automatons in the UPPAAL tool suite. The correctness of these models was verified using TCTL formulas. We presented a set of examples which illustrate the various possible race conditions that occur in a set of communicating threads. The corresponding timed automata race condition verification conditions were also explained.

The future work consists of automating this entire process, right from modeling the application to the specification of timed automata race condition verification mechanisms. Currently, our technique has targeted the VxWorks real-time operating system environment. Future work will be directed towards making this more generic, to cater to various other real time operating system environments.

References

[1] Steve Carr, Jean Mayo and Ching-Kuang Shene. ”Race Conditions: A Case Study”. *The Journal of Computing in Small Colleges, Vol. 17*, pages 88–102, October 2000.

[2] Stefan Savage, Michael Burrows, and Greg Nelson. ”Eraser: A Dynamic Data Race Detector for Multithreaded Programs”. *Transactions on Computer Systems, Vol. 15, No.4*, pages 391–411, November 1997.

[3] Dejan Perkovi and Peter J. Keleher. ”Online Data-Race Detection via Coherency Guarantees”. *The Second Symposium on Operating Systems Design and Implementation (OSDI 96)*, pages 47–57, October 1966.

[4] A. J. Bernstein. ”Analysis of programs for parallel processing, Vol. 15”. *IEEE Transactions on Electronic Computers*, pages 757–762, 1966.

[5] Robert H. B. Netzer and Barton P. Miller. ”What are race conditions? Some issues and formalizations”. *ACM Letters on Programming Languages and Systems*, pages 74–78, March 1992.

[6] Alur, R. and Dill, D. L. ”A Theory of Timed Automata”. *Theoretical Computer Science, 126(2)*, pages 183–235, 1994.

[7] Kim G. Larsen, Paul Pettersson, Wang Yi. ”UPPAAL in a Nutshell”. *Springer International Journal of Software Tools for Technology Transfer 1(1+2)*, 1997.

[8] V. Balasundaram and K. Kennedy. ”Compile-time detection of race conditions in a parallel program”. *Proceedings of the 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[9] P. Emrath and D. Padua. ”Automatic detection of nondeterminacy in parallel programs”. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[10] Chandrasekhar Boyapati, Robert Lee and Martin Rinard. ”Ownership types for safe programming: preventing data races and deadlocks”. *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, November 2002.

[11] Cormac Flanagan and Stephen N. Freud. ”Detecting race conditions in large programs”. *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96, June 2001.

[12] Cormac Flanagan and Stephen N. Freud. ”Type-based race detection for Java”. *Proceedings of the ACM SIGPLAN ’00 conference on Programming language design and implementation, Vancouver, British Columbia, Canada*, pages 219–232, 2000.

[13] C. Boyapati and M. Rinard. ”A parameterized type system for race-free java programs”. *ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 56–69, 2001.

[14] Michiel Ronse and Koen De Bosschere. ”RecPlay: A Fully Integrated Practical Record/Replay System”. *ACM Transactions on Computer Systems, Vol.17, No.2*, pages 133–152, May 1999.

[15] Robert H. B. Netzer, Timothy W. Brennan, Suresh K, Damodaran-Kamal. ”Debugging race conditions in message-passing programs”. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, January 1996.

[16] Jong-Deok Choi and Sang Lyul Min. ”Race Frontier: reproducing data races in parallel-program debugging”. *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, April 1991.

[17] Leslie Lamport. ”Time, clocks, and the ordering of events in a distributed system”. *Communications of the ACM, Volume 21, Issue 7*, pages 558–565, July 1978.

[18] Anne Dinning and Edith Schonberg. ”Detecting access anomalies in programs with critical sections”. *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, December 1991.

[19] Barton Miller, and Jong-Deok Choi. ”A Mechanism for Efficient Debugging of Parallel Programs”. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, June 1988.