

Timing and Race Condition Verification of Real-time Systems

Yann-Hang Lee, Gerald Gannod, Karam S. Chatha, 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 [§].

Progress Report

Verification of Race Conditions in Real-Time Systems

Real-time systems are computing systems that must react within precise time constraints to events in the environment. The correctness of these systems is determined not only by their functional behavior, but also by their temporal properties. A reaction that occurs late could not only be useless, but also catastrophic. 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 report.

This report 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 (TCTL). A set of guidelines have been proposed to map any real-time system to its equivalent timed-automata model in UPPAAL by utilizing our templates. TCTL can then be utilized to verify the absence of race conditions. The report also presents a case study and several examples that demonstrate the successful application of the proposed technique towards verification of race conditions. The case study addresses the design and verification of a real-time application that reads a potentiometer. The examples model a number of inter-process communication mechanism, and present TCTL expressions for their verification.

1 Introduction

Real-time systems are computing systems that must react within precise time constraints to events in the environment. A reaction that occurs late could not only be useless but also catastrophic. The correctness of these systems is determined by both their functional and temporal behaviors. 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 report 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 mechanisms. In this report we focus on the inter-process communication mechanisms of the VxWorks real-time operating system. We present timed automata based models for the various inter-process communication mechanisms in the UPPAAL environment (a real-time verification tool). We discuss TCTL based formulae that verify the correctness of the communication models. We also present guidelines for incorporation of the communication mechanism templates towards modeling generic real time applications. The application of the models and the overall technique is demonstrated by a case study and several examples.

This report is organized as follows: Section 2 gives the necessary background for race conditions, model checking, timed automata, and timed computational tree logic. It also introduces the syntax and semantics of UPPAAL. Section 3 explains the other related work for race condition detection. Section 4 discusses the various inter-process communication mechanisms in VxWorks. Section 5 explains the UPPAAL models that we have designed and developed. It also describes a technique for modelling a generic real time application by utilizing our templates, and its verification. Section 6 presents a case study of a real-time application. Section 7 presents several examples of inter-process communication mechanisms and their verification for race conditions. Finally, Section 8 concludes the report.

2 Background

This section briefly discusses all the relevant concepts involved in this report. Section 2.1 gives an introduction to race conditions and their classification. Section 2.2 discusses the concept of model checking and Section 2.3 introduces the concept of timed automata. Section 2.4 introduces Timed Computation Tree Logic and its notations.

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 [2]. There are three conditions that must occur for a race condition to occur [3]:

- 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.

2.1.1 Classification of Race Conditions

Race conditions are classified in two different ways.

- Based on their synchronization primitives, race conditions can be 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 [4]. 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 [5]. 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 [6]. 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.

- Based on static and dynamic analysis, the race conditions can be classified as apparent races, feasible races and actual races.
 - An apparent race is one that appears possible due to the lack of proper synchronization (and not program semantics). They are easier to locate, but are not a very accurate estimation of race conditions. Exhaustively locating all apparent races is still an NP-hard problem.

Thread 1	Thread 2
<pre> begin { : amount = read_amount(); SemTake(mutex); balance = balance + amount; interest = interest + rate*balance; SemGive(mutex); : } </pre>	<pre> begin { : amount = read_amount(); if(balance < amount) printf("No Funds"); else { balance = balance - amount; interest = interest + rate*bakance; } : } </pre>

Figure 1: Program Segment illustrating Data Races

Thread 1	Thread 2	Thread 3
<pre> begin { static int i = 0; static int A[] = {0,0,0}; SemTake(mutex); i = read_value_i(); A[i] = i; update_value_i(); SemGive(mutex); } </pre>	<pre> begin { static int i = 0; static int A[] = {0,0,0}; SemTake(mutex); i = read_value_i(); A[i] = i; update_value_i(); SemGive(mutex); } </pre>	<pre> begin { static int i = 0; static int A[] = {0,0,0}; SemTake(mutex); i = read_value_i(); A[i] = i; update_value_i(); SemGive(mutex); } </pre>

Figure 2: Program Segment illustrating General Races

Initial State	Thread 1	Thread 2
<pre> begin { : value = oldValue updated = False; : } </pre>	<pre> begin { : value = newValue updated = True; : } </pre>	<pre> begin { : while(updated == false); value = oldValue : } </pre>

Figure 3: Program Segment illustrating Actual, Feasible and Apparent Races

The example in Figure 3 shows the actual, feasible and apparent races. In Figure 3 there are no explicit synchronization mechanisms. A naive race detection algorithm will detect two different races, one on *value* and the other on *updated*. It is due to the fact that there are two shared variables namely *value* and *updated*, and they are not protected by any synchronization or mutual exclusion mechanisms. These two different races constitute the apparent races.

- A feasible race is one that did not occur now (in the current program execution), but does occur in another execution. It is based on the possible behavior of the program, that is on the semantics of the program. It requires full analysis of the programs semantics to determine if the execution could have allowed access to same shared variable to execute concurrently.

Taking the program's semantics into account one can say that only one race condition can occur, that is on the variable *value*. The variable *updated* is initialized to *False*. Hence, *thread1* has to complete its execution for *thread2* to reach the statement 2 where the variable *value* is updated. Hence, there is no race on the variable *value*. Hence, there is only one feasible race that is on the variable *updated*.

- An actual race is one that actually occurred in a particular execution [7]. There may be more than one feasible race that can occur in a particular system, but not all of them will occur in a particular run of the system. The race which actually occurs in a particular run of the system is known as an actual race. In the example in Figure 3, the race on the variable *value* is an actual race. It pertains to a particular execution run of the system.

2.2 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems. The basic idea of model checking is to utilize algorithms, executed by computer tools, to verify the correctness of the system. The user inputs a description of a model of the system and a description of the requirements specification, and leaves the verification to the machine. If an error is recognized, the system generates a counter-example showing the circumstances under which the error has occurred.

The algorithms for model checking are typically based on an exhaustive state space search of the model of the system: for each state of the model it is checked whether it behaves correctly, that is, whether the state satisfies the desired property. In its most simple form, this technique is known as reachability analysis. For example, in order to determine whether a system can reach a state in which no further progress is possible (deadlock), it suffices to determine all reachable states and to determine whether there exists a reachable state from which no further progress can be made.

In this report, we represent the concurrent system as a finite-state machine by utilizing timed automata. The specification or safety assertion is expressed in temporal logic formulas. We can then check if the system meets its specifications by utilizing an algorithm known as the model checker.

2.3 Timed Automata

Timed automata supports the modelling of the finite-state concurrent system by utilizing a time domain. It has constructs to specify the timing parameters such as deadline and period. Hence, timed automata is ideal to model our system.

The theory of timed automata was first introduced in [8], and has since then established as a standard for real-time systems. A timed automata is a finite-state automaton extended with a finite set of real-valued variables called clocks. Each node of the state machine represents a location. A state in the timed automaton consists of the current location

and the current values of the clock variables. Clocks are initially set to zero when the system starts. Once initialized, the clocks start incrementing constantly until they are reset. Each location of the timed automaton has a condition on the clock variable that specifies the amount of time that can elapse in that location. It is known as an invariant. The transitions of timed automaton are labelled with a guard. A guard specifies a condition involving clocks or any atomic propositions, an action and a clock reset.

- The transitions are labelled with an action if it is an instantaneous switch from the current node to another.
- They are labelled with a positive real number (time delay) if the automaton stays in the same location by allowing time to pass.

Once a location is reached, the process can remain in the location as long as its invariant is satisfied. The edge will get active only if the guard condition specified on it is satisfied. The transition will take the edge only if that edge is active. Once a transition is taken, the clock reset variables if any, are reset. The same location can have different states based on the value of changing clocks.

A model checker is a tool that takes as input the model of the system along with its properties, and analyzes the system for their correctness. Temporal logics support the formulation of properties of system behavior over time.

2.4 Timed Computation Tree Logic

Temporal logics like Linear Temporal Logic(LTL) and Computational Tree Logic(CTL) facilitate the specification of properties that focus on the temporal order of the events of the automaton. The temporal ordering specified in LTL and CTL is a qualitative notation. Time is not considered quantitatively in LTL or CTL. Timed computation tree logic is an extension of the CTL with a quantitative notion of time. Timed Computation Tree Logic or TCTL allows the specification of time constraints to the usual CTL. It is also known as real-time branching temporal tree logic or real-time temporal logic.

TCTL is a mechanism to specify the properties of the timed automata. Time is supported in TCTL through the concept of clocks. The basic idea of computation here is that, the transition from one state to the next state occurs on a single tick of a clock. TCTL allows simple time constraints as parameters of the usual CTL temporal operators.

A discrete notion of time is considered in TCTL. A continuous time domain could have been chosen to represent time. However, the model to be checked would have infinitely many states. For each value of time, there would be a state, and hence there would be infinitely many states. This would lead to a state space explosion. Hence, the model checking is done by considering a discrete time domain.

The various operators utilized in TCTL are shown below:

- **X:** X is the next state operator, $\mathbf{X} \phi$ denotes the next state to the current state ϕ .
- **U:** U is the until operator, it refers to all the future states until certain condition becomes valid.
- **F:** F is the eventually or future operator, $\mathbf{F}\phi$ denotes that ϕ holds at some point in future.

$$\mathbf{F}\phi \equiv \text{true} \cup \phi$$
- **G:** G is always or global operator, $\mathbf{G}\phi$ denotes that ϕ is always true.

$$\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$$
- **E** ϕ : E expresses a property over some path. There exists a path where the ϕ holds true.

- $\mathbf{A} \phi$: \mathbf{A} expresses a property over all the paths. All paths satisfy the property ϕ .

Let \mathbf{M} be a timed automaton, AP be a set of atomic propositions and D (formulae clocks) be a set of clocks that is disjoint from the clocks(C) of \mathbf{M} .

For $p \in AP$, $z \in D$, and $\alpha \in \psi(C \cup D)$, the set of TCTL formulae can be written as

$$\phi ::= p \mid \alpha \mid \neg\phi \mid \phi \vee \phi \mid z \text{ in } \phi \mid \mathbf{E}[\phi \cup \phi] \mid \mathbf{A}[\phi \cup \phi]$$

In the formula above α is a clock constraint over formula clocks, and clocks of the timed automaton. It allows, for instance, the comparison of a formula clock and an automaton clock. The boolean operators utilized are true, false, \wedge , \Rightarrow and \Leftrightarrow . Clock z in z in ϕ is called a *freeze identifier*. The condition z in ϕ is valid in state s , if ϕ holds in s where clock z starts with value 0 in s . Variants of the temporal operators like \mathbf{EF} and \mathbf{AF} are valid in TCTL. $\mathbf{EF}\phi$ is pronounced as ϕ holds potentially, and $\mathbf{AF}\phi$ is pronounced as ϕ is inevitable.

$$\mathbf{EF}\phi \equiv \mathbf{E}[\text{true} \cup \phi]$$

$$\mathbf{AF}\phi \equiv \mathbf{A}[\text{true} \cup \phi]$$

State is the unit of consideration in TCTL. The *location* determines that propositions are valid, while the clock valuation determines the validity of the clock constraints in the formula. Since clock constraints may contain besides clocks of automaton, formula clocks, an additional clock valuation is utilized to determine the validity of the statements about these clocks.

The semantics of the TCTL is defined by a satisfaction relation(=) that relates a transition system M , a state, a clock valuation over formula clocks occurring in ϕ , and a formula ϕ .

Utilizing the concept of clocks and the operators mentioned above, a TCTL formulae can be written as

$$\mathbf{AG}[p \Rightarrow \mathbf{AF}_{<6} q]$$

The above formulae states that, on all the paths, at all times, if an event p occurs, then, there exists an event q , which would occur on all the paths in a future state within 6 time units. The above formulae specifies that, the maximum delay between the two events p and q cannot be more than 6 time units.

2.5 Introduction to UPPAAL

Real-time systems are computing systems that must react within precise time constraints to events in the environment. Timed automata [8] 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 [9] is an integrated tool environment for modelling, validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems that are modelled as networks of timed automata.

UPPAAL contains a suite of tools featuring the following:

- A graphical user-interface for specifying networks of timed automata.
- An automatic compilation of the graphical definition into a textual format, which is the basic programming language of the tool for timed automata.

Global Declarations	Process Assignments
Clock x;	b1 := B(1);
Chan a;	b2 := B(2);
Private Declarations for B	System Definition
int n;	system A, b1, b2;
Parameters Passed to B	
const me;	

Figure 4: Declarations and Definitions

- A simulator that allows the step by step or random simulation of the model. A traced simulation can also be done.
- A verifier module that allows for the verification of the TCTL logic formulae. In case verification of a particular real-time system fails, a diagnostic trace is automatically reported by UPPAAL in order to facilitate debugging.

In short, 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 TCTL formulae 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 modelled 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.

2.6 Modeling in UPPAAL

This section describes the syntax and semantics of UPPAAL statements, it explains the guard conditions, reset operations, channels, synchronization, urgency, and committed locations. A real-time system is considered to be a network of non-deterministic sequential processes communicating with each other over channels [9].

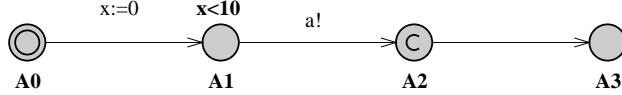


Figure 5: Snap shot illustrating the Process A

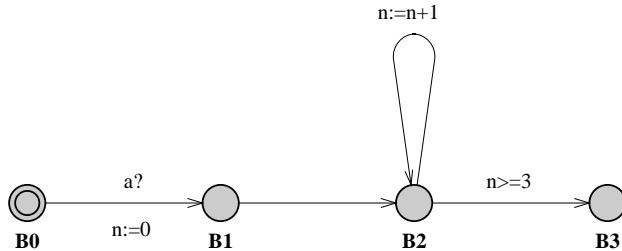


Figure 6: Snap shot illustrating the Process B

2.6.1 Syntax

As UPPAAL facilitates the modelling of real time systems, its modelling 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 6, the transition from B_2 to B_3 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 5, the process is not allowed to stay in location A_1 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 5, the edge from the location A_0 to the location A_1 has got the clock reset operation on clock variable x . A data variable can be reset to an expression. In Figure 6, there is a reset operation on the integer variable n where it is assigned to 0. Similarly, there is another reset operation on the location B_2 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 4 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 6, 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 5, once the system has reached the location $A2$, and the system in Figure 6 be in any of the locations, the next transaction has to be $A2-A3$ as $A2$ is a committed location.

2.6.2 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 5 and Figure 6, 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 labelled 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.

2.6.3 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 Timed Computation Tree Logic (TCTL).

The Figure 7 specifies the semantics of the UPPAAL requirement specification language. The UPPAAL requirement specification supports five types of properties as shown in the Figure 7. 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 $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$, where s_0 is the initial state and s_n 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 $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_i \rightarrow \dots$ for which p holds in all states s_i and which either:
 - is infinite, or

Name	Property	Equivalent to
Possibly	$E\Diamond p$	
Invariantly	$A[] p$	$\neg E\Diamond \neg p$
Potentially always	$E[] p$	
Eventually	$A\Diamond p$	$\neg E[] \neg p$
Leads to	$p \rightarrow q$	$A[] (p \text{ imply } A\Diamond q)$

Figure 7: Semantics of UPPAAL

- ends in a state (L_n, v_n) such that either
 - * for all d : $(L_n, v_n + d)$ satisfies p and $\text{Invariant}(L_n)$, or
 - * there is no outgoing transition from (L_n, v_n)
- **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 [6].

- Static analysis of the program.
- Dynamic analysis.
- Post-mortem analysis of the program traces.

3.1 Ahead-of-time analysis

Ahead-of-time analysis or static analysis of race conditions is usually performed 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 [10], [11] report all the potential races and many of which may not really occur in real executions. Balasundaram et al. [10] describe a graphical approach to model the parallel program into a program execution graph and then obtain a co-graph and analyze it for races. Emrath et al. [11] analyze the source program and construct a partial order execution graph. The partial order execution graph is utilized to identify the conflicting accesses to memory. Boyapati et al. [12] 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. [13], [14] and [15] present static techniques to detect race conditions by utilizing the properties of object-oriented languages. Flanagan et al. [13], [14] designed and implemented a race condition checker *rccjava*, and tested it on a variety of Java programs. *rccjava* was a static race detection analysis technique based on a formal type system that is capable of capturing many common synchronization patterns. A static system has to be instrumented with proper annotations so that a specially designed compiler can retrieve the necessary information it needs to perform the analysis. An annotation assistant called Houdini/rcc [14] was developed to reduce the burden of the programmer. Houdini/cc automatically inserts annotations to analyze the programs. The type system introduced in [15] allows programmers to specify the protection mechanism for each object as the parameters of the object upon the instantiation of the object.

Static or ahead-of-time race condition detection involves a little 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.

3.2 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 [7], [16] and [17] present dynamic race detection techniques by utilizing the Lamports "happen-before" relation [18] to construct partial orders between critical events distributed among parallel processes, resulting in a partial order execution graph (POEG).

Netzer, et al., address the problem of dynamically locating race conditions in the context of explicitly parallel message-passing programs [16]. They address the problem of dynamically locating race conditions utilizing a two-pass on-the-fly algorithm. Its space requirement is independent of the executions length. Eraser [3], 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. [19] 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.

3.3 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 [11] is utilized to do the trace analysis. Incremental tracing [20] can be done to generate a coarse tree during runtime, and it can be expanded in replay for a detailed trace.

Choi et al. combine both the static and dynamic techniques to get the best of both methodologies. Choi and Min also introduce the concept of Race Frontier [17], 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 [7] 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.

To alleviate this problem of run time overhead researchers have developed a series of post-mortem approaches, which usually combine an efficient record phase and a replay phase. During the replay phase, the time-consuming race detection is performed. Even in this case one can only verify the race conditions that occur in that particular execution of the system. They cannot verify all the possible race conditions. Though the problem of intrusion is not eliminated, but it is minimized in this case. But it would not still give a comprehensive solution to detect all the races.

3.4 Comparison with our Approach

We follow the timed automaton approach to detect all the races that are present in a given system. The given system is initially modelled as a state machine.

- In this approach, we do instrument the model, but it does not effect the timing parameters of the system, as we make sure that the instrumentation code does not involve any clock variables.
- UPPAAL is designed to exhaustively analyze all the possible event interleavings, and verify the given timed computation logic. Since it explores all the possible interleavings, it can verify all the possible TCTL conditions that are input to the verifier module.
- Trace analysis can be done utilizing the traces that the tool generates for the TCTL conditions that are not satisfied by the model.

Once the system is modelled, its properties are to be verified. The requirements that are given by the user or the programmer of the real-time tasks are now framed utilizing the real-time temporal logic. The formulae are now

tested on the modelled system. If the model satisfies the formulae then the system is verified for the formulae or else the formulae is not valid on the system.

4 Inter Task Communication Mechanisms in VxWorks

Vxworks is a multi-tasking real-time operating system that supports inter-task communications. A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution, and set of system resources. The inter-task communication facilities allow these tasks to synchronize and communicate in order to coordinate their activity. In VxWorks, the inter-task communication facilities range from fast semaphores to message queues and pipes to network-transparent sockets. We concentrate on Vxworks, and develop UPPAAL models for the various inter-process communication mechanisms present in it, and specify how a system running on VxWorks platform can be converted to UPPAAL model.

VxWorks supplies the following set of inter-task communication mechanisms [21]:

- Shared memory, for simple sharing of data.
- Semaphores, for basic mutual exclusion and synchronization.
- Message queues and pipes for inter-task message passing within a CPU.
- Sockets and remote procedure calls, for network-transparent inter-processor communication.
- Signals, for exception handling.

4.1 Shared memory

Shared Memory as the name suggests is the most obvious mechanism for tasks to communicate by accessing shared data structures. All tasks in VxWorks exist in a single linear address space, hence sharing data structures between tasks is trivial. Shared memory includes variables or data structures that are declared in the global context. They can be referenced directly by code running in different contexts. When shared memory is utilized without any synchronization mechanisms it would result in a data race as in Figure 1.

4.2 Mutual Exclusion

Mutual Exclusion can be achieved using interrupt locks and preemptive locks.

- **Interrupt Locks** The most powerful method available for mutual exclusion is the disabling of interrupts. Such a lock guarantees exclusive access to the CPU. The interrupts can be disabled using *intLock()*, and enabled back using *intUnlock()* system routines. It is inappropriate as a general-purpose mutual-exclusion method for most real-time systems, because it prevents the system from responding to external events for the duration of these locks. Interrupt latency is unacceptable whenever an immediate response to an external event is required.

funcA { :; int lock = intLock(); critical section intUnlock(); : }	funcB { :; taskLock(); criticalSection taskUnlock(); : }
---	---

Figure 8: Program Segment illustrating Mutual Exclusion

- **Preemptive Locks** Using preemptive locks like *taskLock()* and *taskUnlock()* we can disable preemption of the current executing task but Interrupt Service Routines are able to execute. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region.

Mutual exclusion using the above primitives prevents data races, but cannot prevent general races. Figure 8 illustrates an example of mutual exclusion using the above mechanisms.

4.3 Semaphores

VxWorks semaphores are highly optimized, and provide the fastest inter-task communication mechanism. Semaphores are the primary means for addressing the requirements of both mutual exclusion and task synchronization:

- For mutual exclusion, semaphores interlock access to shared resources.
- For synchronization, semaphores coordinate a tasks execution with external events.

There are three types of semaphores, and each addresses a different class of problems:

- **Binary Semaphore** The fastest, most general-purpose semaphore, optimized for synchronization or mutual exclusion.

The general-purpose binary semaphore is capable of addressing the requirements of both forms of task coordination: mutual exclusion and synchronization. The binary semaphore has the least overhead associated with it, making it particularly applicable to high-performance requirements.

- **Mutual Exclusion** The mutual-exclusion semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources. The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

- It can be utilized only for mutual exclusion.
- It can be given only by the task that took it.

Call	Description
semBCreate()	Allocate and initialize a binary semaphore.
semMCreate()	Allocate and initialize a mutual-exclusion semaphore.
semCCreate()	Allocate and initialize a counting semaphore.
semDelete()	Terminate and free a semaphore.
semTake()	Take a semaphore.
semGive()	Give a semaphore.
semFlush()	Unblock all tasks that are waiting for a semaphore.

Figure 9: Semaphore Control Routines

- It cannot be given from an Interrupt Service Routine.
- **Counting Semaphore** A counting semaphore keeps track of the number of times a semaphore is given. It is optimized for guarding multiple instances of a resource. The counting semaphore works like the binary semaphore, except that it keeps track of the number of times a semaphore is given. Every time a semaphore is given, the count is incremented, every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given, when a task is blocked, it becomes unblocked. However, unlike the binary semaphore, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking.

Figure 9 briefly gives the semaphore control routines. The semBCreate(), semMCreate(), and semCCreate() routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by the other semaphore-control routines. When a semaphore is created, the queue type is specified. Tasks pending on a semaphore can be queued according to the task’s priority order SEM-Q-PRIORITY or in first-in first-out order SEM-Q-FIFO.

Improper use of semaphore for synchronization will lead to general races as shown in Figure 2. It may also lead to data races. However if proper synchronization is applied the race conditions may be eliminated.

4.4 Message Queues

Message Queues provide a higher-level mechanism for cooperating tasks to communicate with each other. In Vx-Works, the primary intertask communication mechanism within a single CPU is message queues [21]. Message queues allow a variable number of messages, each of variable length, to be queued. Any task or Interrupt Service Routine(ISR) can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

The messages are created and deleted with the routines shown in Figure 10. This library provides messages that are queued in FIFO order, with a single exception: there are two priority levels, and messages marked as high priority are attached to the head of the queue.

The message queue is created with the function msgQCreate(). The function’s parameters specify the maximum number of messages that can be queued in the message queue, and the maximum length in bytes of each message. msgQdelete() deletes the message queue and terminates it.

Call	Description
msgQCreate()	Allocate and initialize a message queue.
msgQDelete()	Terminate and free a message queue.
msgQSend()	Send a message to a message queue.
msgQReceive()	Receive a message from a message queue.

Figure 10: Message Queue Control Routines

A task receives its messages with msgQReceive(). If the message is already available in the queue when the task reaches msgQReceive() then the task reads the message. If no message is available, the task either blocks itself if the WAIT-FOR-EVER parameter is specified in the msgQReceive(). It goes ahead if the NO-WAIT option is specified. The tasks that are blocked can be ordered either by their priority or in the First-In-First-Out order of their arrival.

A tasks sends its messages with the msgQsend() function. If the message queue is empty when the task is ready to send the message with the msgQsend() the message is placed in the message queue. If the message queue has no empty slots to put the data then the task is blocked if the WAIT-FOR-EVER option is specified or it may go ahead if NO-WAIT option is specified just as in case of msgQsend().

A message queue is a high level communication mechanism which can be simulated via semaphores as shown in Figure 11. The declarations describe the setup needed for the message queue. We use two counting semaphores to do the task synchronization between the two sets of processes and the two binary semaphores to provide mutual exclusion for each set of processes that write or read. We assume that the tasks are blocking and semaphores are created using the WAIT-FOR-EVER option.

- The number of *QEmpty* semaphores available indicate the number of message queue locations that are available for the writer tasks to write. Initially *QEmpty* is Full, that indicates that all the locations in the message queue are free.
- The number of *QFilled* semaphores available indicate the number of message queue locations that are filled with messages. Initially *QFilled* is Empty, that indicates that none of the locations of the message queue are filled with messages.
- *ReaderBinary* and *WriterBinary* are utilized to provide mutual exclusion between the multiple readers and writers. Initially they are full. Full indicates that all the semaphores are available for the tasks to take.
- The writer who acquires the *WriterBinary* takes the *QEmpty* semaphore if it is available. It is available only if the message queue is not totally filled. It updates the queue with its message. Later, it gives a *QFilled* semaphore indicating that the message queue has got a data filled that can be taken by any reader waiting for it. It gives away the *WriterBinary* and exits.
- If the *QEmpty* is not available the writer is blocked on the semaphore. It can go ahead only when it is available, that is, when it is given by any reader who has read the message.
- The reader who acquires the *ReaderBinary* takes the *QFilled* semaphore if it is available(it is available only if there is a message present in the message queue). It reads the message, and gives a *QEmpty* semaphore indicating that the message queue has got an empty slot. It exits giving away the *ReaderBinary* semaphore that it has taken.

Declarations	Writer	Reader
CountingSemaphores QFilled[Empty, Max_Msgs]; QEmpty[Full, Max_Msgs]; BinarySemaphore ReaderBinary[Full]; WriterBinary[Full];	begin { : SemTake(WriterBinary); SemTake(QEmpty, WaitForEver); MsgQWrite; SemGive(QFilled); SemGive(WriterBinary); : }	begin { : SemTake(ReaderBinary); SemTake(QFilled, WaitForEver); MsgQRead; SemGive(QEmpty); SemGive(ReaderBinary); : }

Figure 11: Program Segment simulating Message Queues using semaphores

- If the *QFilled* is not available the reader is blocked on the semaphore. It can go ahead only when it is available that is when it is given by any writer who writes to the message queue.

4.5 Signals

VxWorks supports a software signal facility. Signals asynchronously alter the control flow of a task. Any task or ISR can raise a signal for a particular task. The task being signaled immediately suspends its current thread of execution, and executes the task-specified signal handler routine the next time it is scheduled to run.

The signal handler executes in the receiving tasks context and makes use of that tasks stack. Signals are more appropriate for error and exception handling than as a general purpose inter-task communication mechanism. In general, signal handlers should be treated like Interrupt Service Routines.

Figure 12 illustrates a simple signal handler using semaphores. A semaphore *signalSem* is declared as empty initially. It is utilized for the asynchronous communication between the process, and the signal handler routine. In the process if an error occurs the semaphore *signalSem* is given. The interrupt service routine that is waiting on the *signalSem* gets the semaphore, and starts executing its routine. Meanwhile the task that gives the semaphore does a *taskSuspend(0)* suspending itself.

In this report, we have developed models for the different types of inter-process communication mechanisms such as semaphores, message queues, interrupt service routines, and signal handlers. We have not addressed inter-processor communication mechanisms.

5 UPPAAL Models

We have designed and developed UPPAAL models for the various inter-process communication mechanisms present in VxWorks. This section discusses the models of the various inter-process communication mechanisms and their verification by timed automata conditions. We describe the modelling of a real-time application as a network of

Declarations	Signal Handler	Process
BinarySemaphore	begin { : : signalSem[Empty]; : : semTake(signalSem); : : }	begin { : : if (Error) { : semGive(signalSem); : taskSuspend(0); }

Figure 12: Program Segment simulating a simple Signal Handler utilizing semaphores

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.

5.1 Semaphores

Each semaphore has three models associated with it, namely, the initialization model, the enqueueing model, and the dequeuing model. Figure 13 specifies the required declarations necessary for a semaphore model. The Figures 14, 15, 16, 17, and 18 discuss the binary Semaphore (FIFO Option) that is used as a synchronization semaphore. Figures 14, and 15 specify the processes that give and take the semaphores respectively. Figures 16, 17, and 18 discuss the initialization, enqueueing, and the dequeuing models respectively.

The declarations section shown in Figure 13 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 inter-process communication 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 13 in the global declarations section, for each of the inter-process communication 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 (*pids*) 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 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

```

// 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 13: Declarations required for the FIFO binary synchronization semaphore



Figure 14: User ProcessB that gives the FIFO binary synchronization semaphore

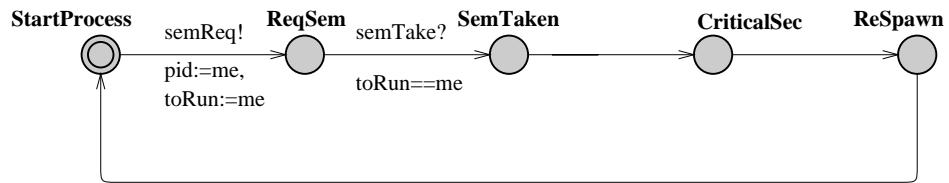


Figure 15: User ProcessA that takes the FIFO binary synchronization semaphore

WAIT-FOR-EVER option. The NO-WAIT is initially set to 0. The channel *noWait* is utilized in this connection.

5.1.1 Binary Semaphore

Figure 16 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 15 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 17.

- If a semaphore is available, it is signalled 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 enqueueing 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 14 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 inorder 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,
 - the tasks waiting(*tasksw*) is decremented by 1, and
 - the waiting task is signalled via the channel *semTake*.

Figure 19 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*, *processB* is the instantiation of the *processB*.

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 verified 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*.

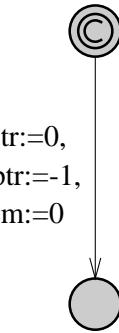


Figure 16: Initialization of a FIFO binary synchronization semaphore

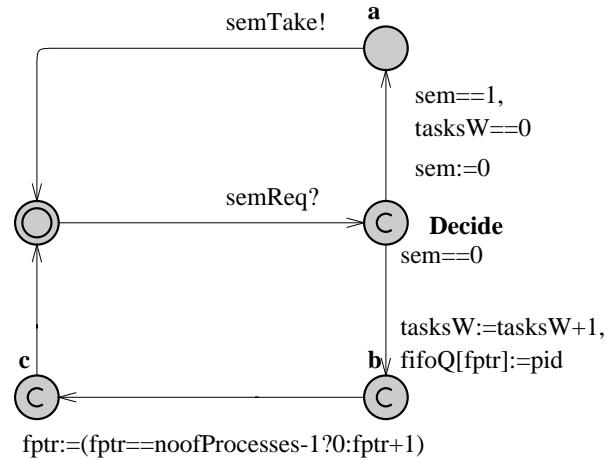


Figure 17: Enqueuing of tasks waiting for the FIFO binary synchronization semaphore

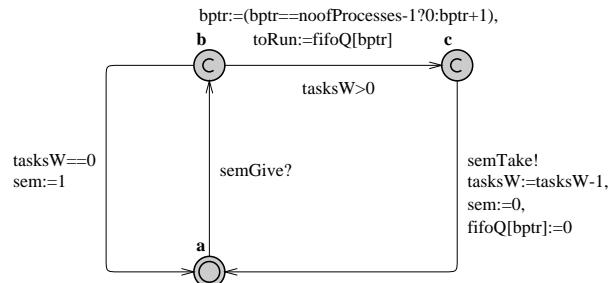


Figure 18: UPPAAL Model illustrating the dequeuing of tasks

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 19: TCTL verification conditions for the FIFO binary semaphore

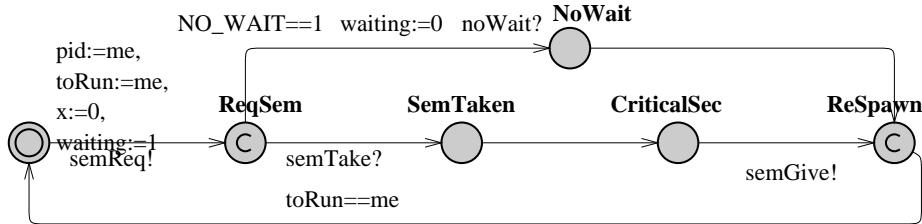


Figure 20: User application for FIFO Mutual Exclusion semaphore

4. On all the paths, if a process has reached the location *SemTaken*, it will eventually reach the location *CriticalSec*.
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 *pids* 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 enqueueing 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*.

5.1.2 Mutual-Exclusion Semaphore

The Mutual-Exclusion semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.

- A Mutual-Exclusion semaphore can only be given by the thread that takes it. This is modelled by making sure that only the process that takes the semaphore eventually gives it.

Figure 21 shows the initialization model for the mutual-exclusion semaphore, that is executed before any other model, as it has a committed start location. The corresponding enqueueing and dequeuing models are shown in Figures 22 and 23. The initialization model shown in Figure 21, initializes the front pointer $fptr$ to 0, the rear pointer $bptr$ to -1, and creates the semaphore by setting sem to 1.

The user application shown in Figure 20 starts with requesting a semaphore utilizing the channel $semReq$. The control is now transferred to Figure 22 to the location *Decide*. Here one of the three paths are taken.

- If the semaphore is not available, that is sem is equal to 0, and the NO-WAIT option is set, then the control is returned to the next executable user location in Figure 20, that is, to the location *ReSpawn*, via the channel $noWait$.
- If the semaphore is available, and none of the tasks are blocked on the semaphore, that is $tasksW$ is equal to 0, then
 - the semaphore is given to the requesting task via the channel $semTake$,
 - sem is set to 0 to make it unavailable, and
 - the control returns to the user process in Figure 20, to the location *SemTaken*.
- If the semaphore is not available, and the NO-WAIT option is set to 0, then the task has to wait.
 - The task's pid is now put in the $fifoQ$ in the location pointed to by the $fptr$.
 - The number of tasks that are waiting($tasksW$) is incremented by 1.
 - The $fptr$ is now made to point to the new location based on its current value.

Once the control has reached the location *SemTaken* in Figure 20, it implies that the process has taken the semaphore, and it can now execute the critical section. The location *CriticalSec* can be replaced by any user specific critical section. Once the user process has finished the execution of the critical section it has to give away the semaphore before it can quit. The user process now triggers a $semGive!$. The control is transferred to Figure 23 to the location *Decide*. Now,

- If $tasksW$ is equal to 0, that is none of the tasks are waiting for the semaphore, then the semaphore sem is made available by setting it to 1.
- If there are tasks that are waiting for the semaphore, that is $tasksW > 0$,
 - In Figure 23, the process that is to take the semaphore next is decided utilizing the $bptr$.
 - The $toRun$ variable is updated to contain the pid of the next process to run.
 - The tasks that are waiting are signalled via the channel $semTake$.
 - The $tasksW$ is decremented by 1, as one of the processes has been dequeued out of the queue.
 - The position of the $fifoQ$ from where the process has been dequeued is reset.

Figure 24 shows the timed automata verification conditions for the binary semaphore with FIFO option. The number on the left specifies the serial number of the timed automata condition that is shown in Figure 24. The conditions shown in italics are designed to fail the verifier test.

1. It checks to see that there are only 3 processes throughout the duration of the system. This makes sure that none of the processes modify the system parameters.

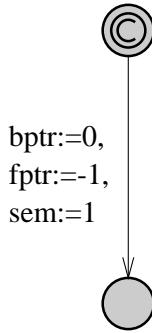


Figure 21: Initialization of FIFO Mutual Exclusion semaphore

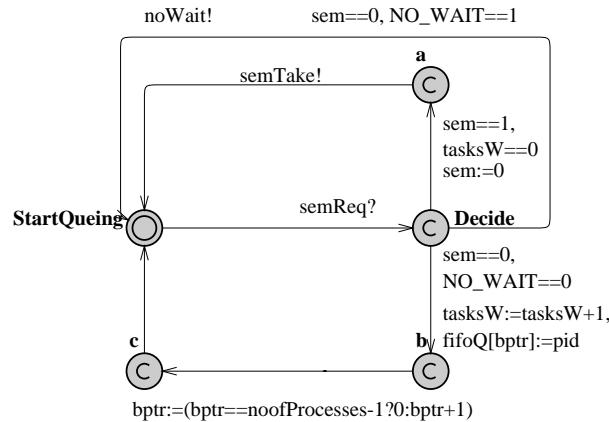


Figure 22: Enqueuing model for FIFO Mutual Exclusion semaphore

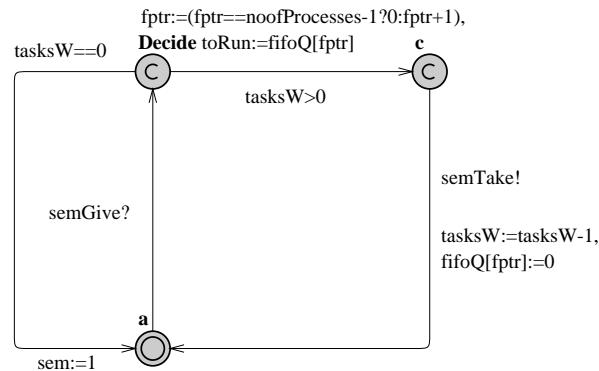


Figure 23: Dequeueing model for FIFO Mutual Exclusion semaphore

2. This condition checks 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*.
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 processes. They have been verified for all the other processes as well.

7. On all the possible paths, if a process has entered the critical section in the location *CriticalSec*, the semaphore is not available. A process can enter the critical section only if it gets the semaphore, in such a case the semaphore is not available to the other processes.
8. On all the possible paths, if a process reaches the location *SemTaken*, the semaphore is not available. A process model, once it reaches the location *SemTaken*, it would have taken the semaphore, so the semaphore is not available to any other process that is requesting it.
9. There exists a path, where both the *process1* and *process2*, or *process2* and *process3*, or *process3* and *process1* have entered the critical section, that is they are either in the locations *ReqSem* or *CriticalSec*. This condition fails as only one process can exist in the critical section at any point of time.
10. There exists a path, where all the three processes have requested the semaphores, and are waiting for it. This condition fails, as there cannot be a case where all the three processes are waiting for the semaphore, because at least one of them has to get the semaphore.
11. This condition verifies that the *fifoQ* is never holding any processes in it waiting for the semaphores. This is valid when NO_WAIT is set to 1.
12. This condition is valid only when NO_WAIT is set to 1. It implies that, if a process is in the critical section, any other process requesting the semaphore will reach the next valid executable location *ReSpawn*. A *waiting* variable is set when the request is made, and it is reset when the *noWait* channel is used. When the control reaches the location *ReSpawn* through the *noWait* channel, the *waiting* variable has to be set to 0.
13. If a process has entered the *NoWait* location, it implies that the semaphore is not available. The *ReqSem* is a committed location, so the process has to either get the semaphore, or it has to go to the *NoWait* location. The process can enter the location only if the semaphore is not available and the NO_WAIT option is set.
14. This condition verifies the dequeuing model. It is similar to the condition verified in the case of the binary semaphore.
15. This condition checks the enqueueing 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*.

-
- | | |
|---|------------------------------|
| 1. A[] noofProcesses==3 | (Verified for all processes) |
| 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. $E<> (((process1.CriticalSec \text{ or } process1.SemTaken) \text{ and } (process2.CriticalSec \text{ or } process2.SemTaken))$
$(((process1.CriticalSec \text{ or } process1.SemTaken) \text{ and } (process3.CriticalSec \text{ or } process3.SemTaken))$
$((process3.CriticalSec \text{ or } process3.SemTaken) \text{ and } (process2.CriticalSec \text{ or } process2.SemTaken)))$ | |
| 10. $E<> process1.ReqSem \text{ and } process2.ReqSem \text{ and } process3.ReqSem$ | |
| 11. A[](fifoQ[0] == 0 and fifoQ[1] == 0 and fifoQ[2] == 0). | (Valid for NO_WAIT = 1) |
| 12. A[](((process1.SemTaken or process1.CriticalSec) and process2.ReSpawn)
imply process2.waiting == 0) | (Valid for NO_WAIT = 1) |
| 13. A[] (process1.NoWait or process2.NoWait or process3.NoWait imply sem==0) | (Valid for NO_WAIT = 1) |
| 14. 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) |
| 15. 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 24: TCTL verification conditions for the FIFO Mutual Exclusion semaphore

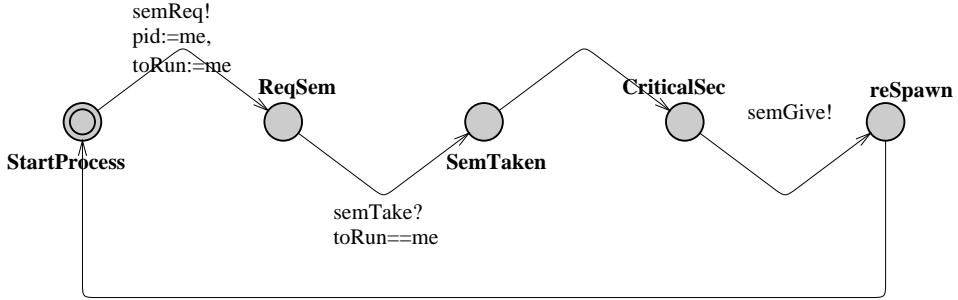


Figure 25: User application utilizing the Priority semaphore

The UPPAAL model for semaphores with the priority option is similar to that of the mutual exclusion semaphore with the FIFO option. The *pid* of the process is utilized as the priority of the process. The lower the *pid* is, the higher the priority. Figure 25 shows the UPPAAL model of the user application that is requesting the priority semaphore. Figure 26 shows the initialization routine where,

- the *fptr* is initialized to 0,
- the semaphore *sem* is initialized to 1,
- we do not need a *bptr* here, as the dequeuing is done utilizing the priority of the processes, and
- the *fifoQ* is filled with a very large number.

The enqueueing model shown in Figure 27 is similar to the one for the mutual exclusion semaphore with FIFO option. In the dequeuing model in Figure 28,

- if the *tasksW* is equal to 0, that is none of the tasks are waiting, the semaphore *sem* is set to 1, and the model exits.
- if not,
 - The next process to run is identified based on the priority of the processes that are in the queue.
 - At the location *Decide*, all the process *pids* are checked to decide the process with the highest priority, and the *toRun* variable is initialized with it.
 - The corresponding location in the *fifoQ* from where the process is dequeued is set to a very large number.
 - As a task is dequeued, the *tasksW* variable is decremented by 1.
 - A signal is sent to the waiting user model via the channel *semTake* allowing it to grab the semaphore.

Figure 25 illustrates the user model that utilizes the semaphore with Priority option. It is similar to the model of the thread utilizing a mutual exclusion semaphore as shown in Figure 20.

Figure 29 shows the timed automata verification conditions that are relevant for the semaphore with priority option. The number on the left specifies the serial number of the timed automata condition that is shown in Figure 29. All the conditions, unless shown in italics, are designed to pass the verifier test for satisfiability.

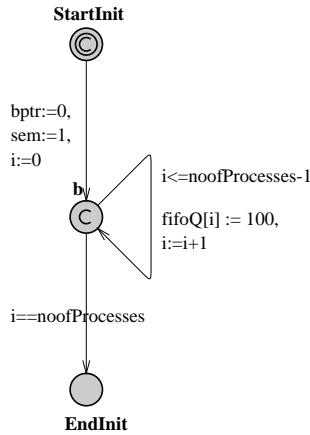


Figure 26: Initialization of a Priority semaphore

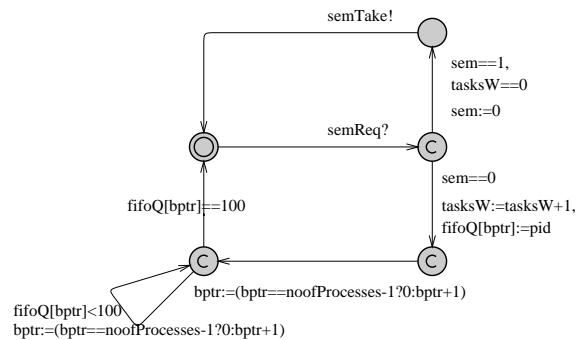


Figure 27: Enqueuing of tasks waiting for the Priority semaphore

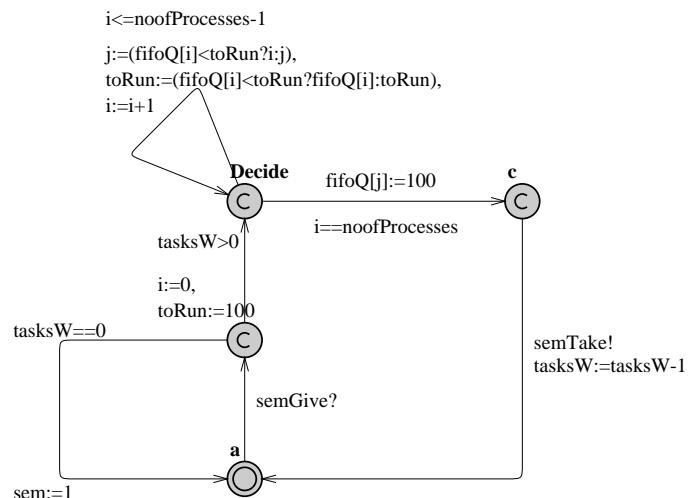


Figure 28: Dequeueing of tasks, Priority semaphore

-
1. $A[] \text{ noofProcesses} == 3$
 2. $\text{process1.ReqSem} \rightarrow \text{process1.SemTaken}$ (Verified for process1 and process2)
 3. $\text{process3.ReqSem} \rightarrow \text{process3.SemTaken}$
 4. $\text{process1.StartProcess} \rightarrow \text{process1.ReqSem}$ (Verified for all processes)
 5. $\text{process1.SemTaken} \rightarrow \text{process1.CriticalSec}$ (Verified for all processes)
 6. $\text{process1.CriticalSec} \rightarrow \text{process1.ReSpawn}$ (Verified for all processes)
 7. $\text{process1.ReSpawn} \rightarrow \text{process1.StartProcess}$ (Verified for all processes)
 8. $A[] (\text{process1.CriticalSec} \text{ imply } \text{sem} == 0)$ (Verified for all processes)
 9. $E<> (((\text{process1.SemTaken} \text{ or } \text{process1.CriticalSec}) \text{ and } (\text{process2.SemTaken} \text{ or } \text{process2.CriticalSec})) \text{ or } ((\text{process1.SemTaken} \text{ or } \text{process1.CriticalSec}) \text{ and } (\text{process3.SemTaken} \text{ or } \text{process3.CriticalSec})) \text{ or } ((\text{process2.SemTaken} \text{ or } \text{process2.CriticalSec}) \text{ and } (\text{process3.SemTaken} \text{ or } \text{process3.CriticalSec})))$
 11. $A[] ((\text{process1.ReqSem} \text{ and } (\text{process3.reSpawn} \text{ or } \text{process3.Startprocess}) \text{ and } \text{DequeSems.c}) \text{ imply } (\text{toRun}==1))$
 10. $A[] ((\text{process1.ReqSem} \text{ and } \text{process2.ReqSem} \text{ and } (\text{process3.reSpawn} \text{ or } \text{process3.StartProcess}) \text{ and } \text{DequeSems.c}) \text{ imply } (\text{toRun}==1))$
 12. $A[] ((\text{process2.ReqSem} \text{ and } \text{process3.ReqSem} \text{ and } (\text{process1.reSpawn} \text{ or } \text{process1.StartProcess}) \text{ and } \text{DequeSems.c}) \text{ imply } (\text{toRun}==2))$
 13. $A[] ((\text{process1.ReqSem} \text{ and } \text{process3.ReqSem} \text{ and } (\text{process2.reSpawn} \text{ or } \text{process2.StartProcess}) \text{ and } \text{DequeSems.c}) \text{ imply } (\text{toRun}==1))$
 14. $A[] ((\text{process1.SemTaken} \text{ or } \text{process1.CriticalSec}) \text{ and } \text{process2.ReqSem} \text{ and } \text{EnqueSems.b} \text{ and } \text{fifoQ}[0] == 3 \text{ imply } (\text{fifoQ}[0] == 3 \text{ and } \text{fifoQ}[1]==2)) \text{ or } ((\text{process1.SemTaken} \text{ or } \text{process1.CriticalSec}) \text{ and } \text{process2.ReqSem} \text{ and } \text{EnqueSems.b} \text{ and } \text{fifoQ}[1] == 3 \text{ imply } (\text{fifoQ}[1] == 3 \text{ and } \text{fifoQ}[2]==2)) \text{ or } ((\text{process1.SemTaken} \text{ or } \text{process1.CriticalSec}) \text{ and } \text{process2.ReqSem} \text{ and } \text{EnqueSems.b} \text{ and } \text{fifoQ}[2] == 3 \text{ imply } (\text{fifoQ}[2] == 3 \text{ and } \text{fifoQ}[0]==2)) \text{ or }$
-

Figure 29: TCTL conditions for the priority semaphore

1. It checks to see that there are only 3 processes throughout the duration of the system. This makes sure that none of the processes modify the system parameters.
2. This condition checks 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 the *process2*.
3. If *process3* has reached the location *ReqSem*, it will not always get the semaphore, as it is the lowest priority process. It may happen that, even before it gets the semaphore, the higher priority processes may request the semaphore, hence it gets starved. So, this condition fails.
4. On all the paths, if a process has reached the location *StartProcess*, it will eventually reach the location *ReqSem*.
5. On all the paths, if a process has reached the location *SemTaken*, it will eventually reach the location *CriticalSec*.
6. On all the paths, if a process has reached the location *CriticalSec*, it will eventually reach the location *ReSpawn*.
7. On all the paths, if a process has reached the location *ReSpawn*, it will eventually reach the location *StartProcess*.

The conditions 2, 4, 5, 6, and 7 check for the absence of deadlocks in *process1*. They have been verified for *process2*.

8. On all the paths, if a process has entered the location *CriticalSec*, it implies that the semaphore is not available.
9. There cannot exist a path where more than one process has entered the location *SemTaken* or the location *CriticalSec*, as, it means that more than one process has taken the semaphore.
10. If the *process1* is requesting the semaphore, the next one to run is *process1*.
11. If the *process1* and *process2* are requesting the semaphores, the next one to run is *process1*, as *process1* has got higher priority than the other waiting process(*process2*).
12. If the *process2* and *process3* are requesting the semaphores, and *process1* is not requesting the semaphore, the next one to run is *process2*, as *process2* has got higher priority than the other waiting process(*process3*).
13. If the *process1* and *process3* are requesting the semaphores, and *process2* not requesting the semaphore, the next one to run is *process1*, as *process1* has got higher priority than the other waiting process(*process3*).
14. This condition verifies the enqueueing model of the semaphore. It is similar to the condition that is shown for the binary semaphore.

The Figures 30, 31, and 32 specify the initialization routine, the user application, and the mechanism utilized to process the timeout for a semaphore that is initialized with a TIMEOUT option. The timeout value *timeOut* specifies how long the user process can wait in the location *ReqSem* for the semaphore, before it can proceed to the next executable location. Each process model is got a clock associated with it, in order to specify time. The process that requests the semaphore gets blocked at the location *ReqSem*. It can wait at the location as long as its clock variable *x* is less than the *timeOutVal* as set by the user in the initialization routine in Figure 30. Once the clock variable *x* reaches the timeout value, the *timingOut* channel is triggered, and the control reaches Figure 32. Here, the process

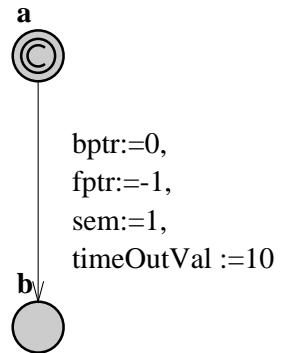


Figure 30: Initialization of a semaphore with TIMEOUT option)

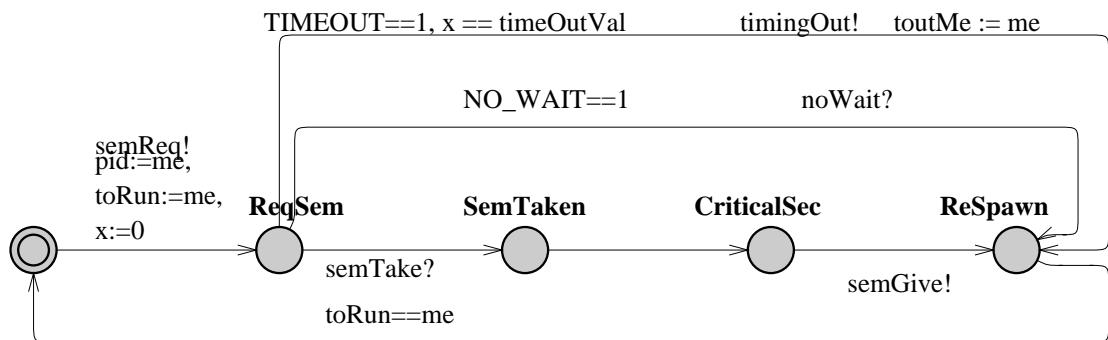


Figure 31: User Application with TIMEOUT option

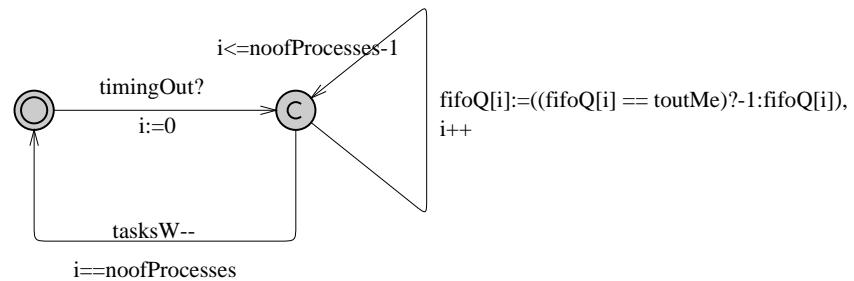


Figure 32: Figure Illustrating TIMEOUT Mechanism

-
1. $A[] \text{ noofProcesses} == 3$
 2. $\text{process1.ReqSem} \rightarrow \text{process1.SemTaken}$ (Verified for all processes)
 3. $\text{process1.ReqSem} \rightarrow \text{process1.ReSpawn}$ (Verified for all processes)
 4. $\text{process1.StartProcess} \rightarrow \text{process1.ReqSem}$ (Verified for all processes)
 5. $\text{process1.SemTaken} \rightarrow \text{process1.CriticalSec}$ (Verified for all processes)
 6. $\text{process1.CriticalSec} \rightarrow \text{process1.ReSpawn}$ (Verified for all processes)
 7. $\text{process1.ReSpawn} \rightarrow \text{process1.StartProcess}$ (Verified for all processes)
 8. $A[] (\text{process1.CriticalSec} \text{ imply } \text{sem} == 0)$ (Verified for all processes)
 9. $A[] (\text{process1.SemTaken} \text{ imply } \text{sem} == 0)$ (Verified for all processes)

 10. $E <> \text{process1.ReqSem} \text{ and } \text{process2.ReqSem} \text{ and } \text{process3.ReqSem}$

 11. $A[] (\text{sem} == 0 \text{ and } \text{process1.ReqSem} \text{ and } \text{process1.x} < \text{timeOutval} \text{ imply } (\text{fifoQ}[0] == 1 \text{ or } \text{fifoQ}[1] == 1))$ (Verified for all processes)
 12. $A[] (\text{process1.x} > \text{timeOutVal} \text{ imply not } \text{process1.ReqSem})$
(Verified for all processes)
-

Figure 33: Verification conditions for semaphore with TIMEOUT option

issuing the *timingOut* is dequeued from the *fifoQ*, and the *fptr* and *tasksW* are updated. All the other models are similar to the other semaphore models discussed previously.

The Figure 33 specifies the verification mechanisms for the semaphore with the TIMEOUT option. The conditions are designed to pass the verifier test for satisfiability, except those shown in italics.

1. This condition checks to see 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 checks to see that if the process has requested the semaphore, it will eventually get it. But due to the timeout mechanism this is not possible. Hence, this condition fails.
3. On all paths, if a condition has reached the location *ReqSem*, it eventually reaches the location *ReSpawn*. The process can go either through the critical section, or if its timeout clock expires, it will go through the *timingOut* channel.
4. On all the paths, if a process has reached the location *StartProcess*, it will eventually reach the location *ReqSem*.
5. On all the paths, if a process has reached the location *SemTaken*, it will eventually reach the location *CriticalSec*.
6. On all the paths, if a process has reached the location *CriticalSec*, it will eventually reach the location *ReSpawn*.
7. On all the paths, if a process has reached the location *ReSpawn*, it will eventually reach the location *StartProcess*.

The conditions 3, 4, 5, 6, and 7 check for the absence of deadlocks in each process model.

8. This condition implies that, if a process has entered the critical section, the semaphore is not available.
9. This condition verifies that, if the semaphore is taken by a process, then it is not available to any other process.
10. This condition checks to see that, all the processes cannot be waiting in the location *ReqSem* for the semaphore, as atleast one of them should have got it.
11. For all the paths, if the semaphore is not available, and the *process1* has requested the semaphore, and the value of the clock variable is less than the *timeOutval*, in such a case the user process is present in the *fifoQ*.
12. For all the paths, if the clock variable *process1.x* has reached a value that is greater than the *timeOutVal* specified, the process has to leave the *ReqSem*. This condition is verified for all the processes.

5.1.3 Counting Semaphore

A Counting Semaphore keeps track of the number of times a semaphore is given. It is optimized for guarding multiple instances of a resource. It implies that multiple instances of the same semaphore exist at any point of time. This can be achieved in the current context by modifying the *sem* in Figure 13 declaration to contain values ranging from 0 to the number of counting semaphores that are available. Counting Semaphores with FIFO/Priority and NO-WAIT/WAIT-FOR-EVER/Timeout options have been modelled on similar lines as the above mechanisms.

Figure 34 shows the user application requesting the counting semaphore. Figure 35 shows the initialization model for the counting semaphore. Here,

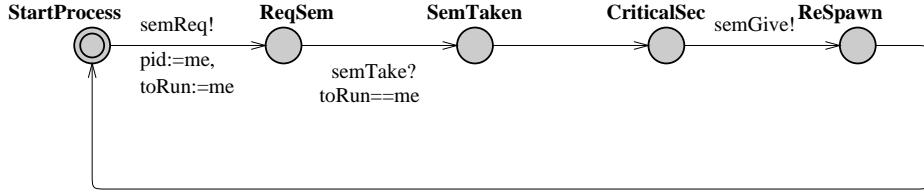


Figure 34: User application which utilizes the Counting semaphore

- The *fptr* and the *bptr* are initialized to 0 and -1 respectively.
- The *sem* variable is initialized to 2 indicating the number of counting semaphores that are available.

Figure 36 shows the enqueueing model for the counting semaphore. It is quite similar to the other enqueueing models.

- Whenever a semaphore is allocated, the *sem* variable is decremented by 1, unlike the previous cases, where it was reset.

Figure 37 shows the dequeuing model for the counting semaphore. It is quite similar to the previous models, the difference being,

- Whenever a semaphore is given, and there are no tasks waiting for the semaphore, the *sem* is incremented by 1.
- Whenever a semaphore is given, and there are tasks waiting for it, the most eligible task is signalled to take it, just as in the previous dequeuing models.

Figure 38 shows the timed automata verification conditions for the counting semaphore with FIFO option. The following paragraph briefly describes the timed automata conditions shown in Figure 38. The number on the left specifies the serial number of the timed automata condition that is shown in Figure 38. The conditions shown in *italics* are designed to fail the verifier test. All the other properties are designed to satisfy the verifier test.

1. It checks to see that there are only 4 processes throughout the duration of the system. This make sure that none of the processes modify the system parameters.
2. This condition checks 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*.
5. On all the paths, if a process has reached the location *CriticalSec*, it will eventually reach the location *ReSpawn*.

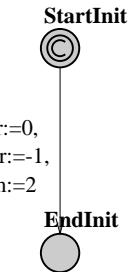


Figure 35: Initialization of a Counting semaphore

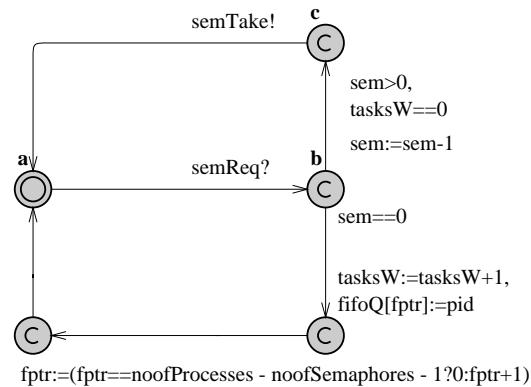


Figure 36: Enqueuing of tasks waiting for the Counting semaphore

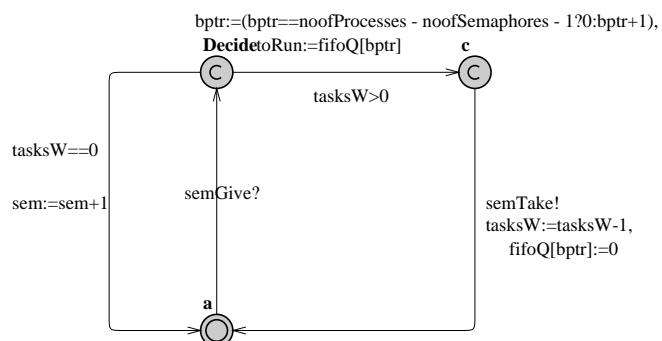


Figure 37: Dequeueing of tasks, Counting semaphore

1. A[] noofProcesses == 4
 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.SemTaken or process1.CriticalSec imply process1.takenSem==1)
(Verified for all processes)
 8. A[](process1.takenSem + process2.takenSem + process3.takenSem + process4.takenSem
 <= noofSemaphores)

Figure 38: TCTL conditions for the counting semaphore

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 the system. They can be extended to all the other processes.
 7. On all the paths, if the process has entered the critical section, that is, it is currently in the location *SemTaken* or in *CriticalSec*, it implies that the variable *takenSem* is set to 1.
 8. On all the paths, the number of processes in the critical section, is always less or equal to the number of semaphores available.

5.2 Message Queues

Message queues have been developed 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 atleast 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 39. 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 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

```

//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[bufSize];                                            //the msgQ that holds the message
int fifoQG[noofProcesses];
int fifoQP[noofProcesses];                                         //FIFO Queue to hold the waiting processes Getting Msgs
int gptr, bptr;                                                     //Front Pointer and Back Pointer of the FIFOQ
int mptr, fptr;                                                     //Front and Back Pointer of the msgQ
int pptr, bptr;                                                     //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 39: UPPAAL Model illustrating the declarations necessary for a Message Queue

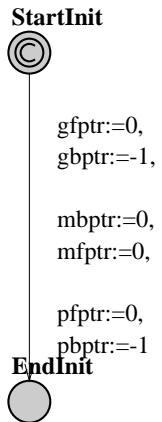


Figure 40: Initialization of a Message Queue

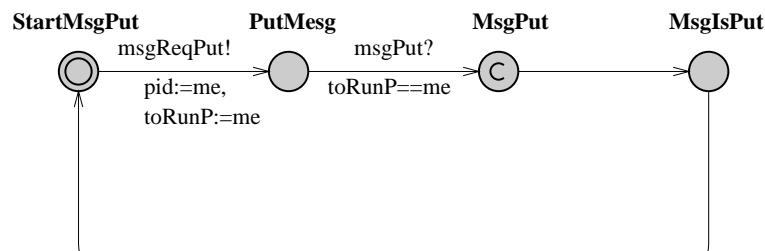


Figure 41: UPPAAL Model illustrating a writer task in the Message Queue Models

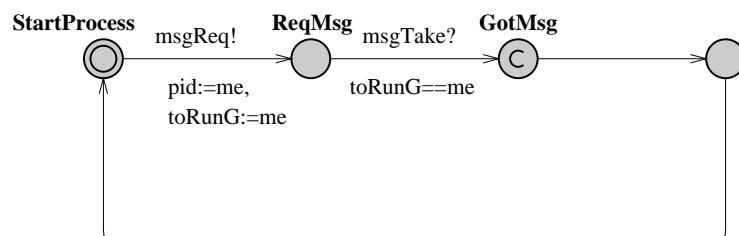


Figure 42: UPPAAL Model illustrating a reader task in the Message Queue Models

message queue. $pbptra$ 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 40. 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 42. The user process that writes a message to the message queue is shown in Figure 41. The process in Figure 42 initially makes a request for the message via the channel $msReq$. The control is now transferred to the model in Figure 44 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 42.
- When at the location *Decide* in Figure 44, 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 signalled, 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 45. 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 be signalled. This is done via the channel $msgPut$.
 - * The next thread to run is identified by the $pfptra$, 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 $mfptra$ after which, $mfptra$ 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 *Updategptr*.
- The reader process is queued in the $fifoQG$ utilizing its pid , and the $gfptra$ is updated.
- The $tasksWg$ is incremented by 1, to denote that there exists a reader process that is blocked in the queue $fifoQG$.

The user process that writes a message to the message queue $msgQ$ is shown in Figure 41. It initially makes a request to place a message in the message queue via the channel $msgReqPut$. The control is now transferred to Figure 43 to the location *Decide*. Now,

- if any of the slots in the message queue are free,
 - Control is transferred to the location *Updatemfptr*.

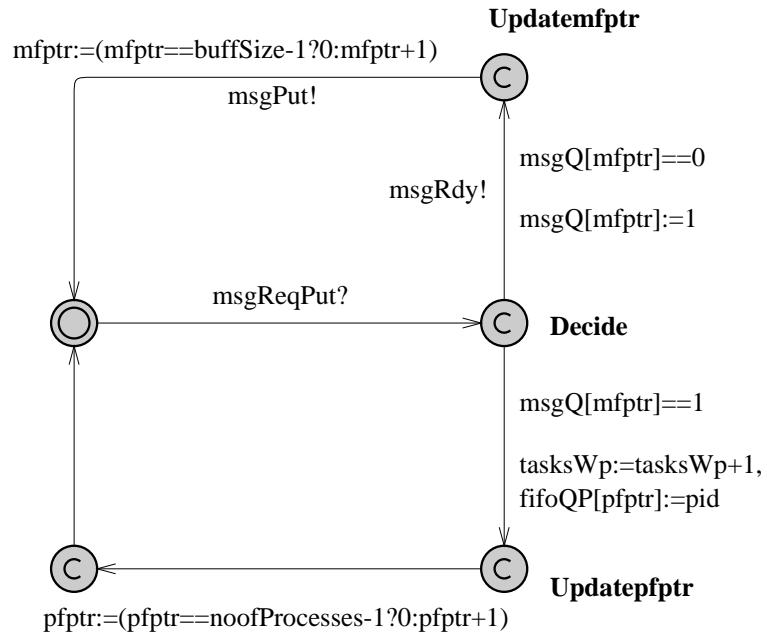


Figure 43: Enqueing model for writer tasks

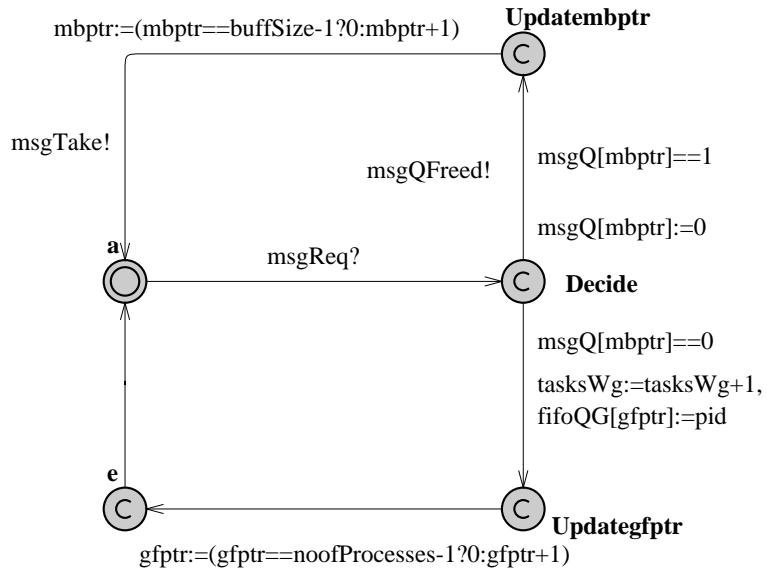


Figure 44: Enqueing model for reader tasks

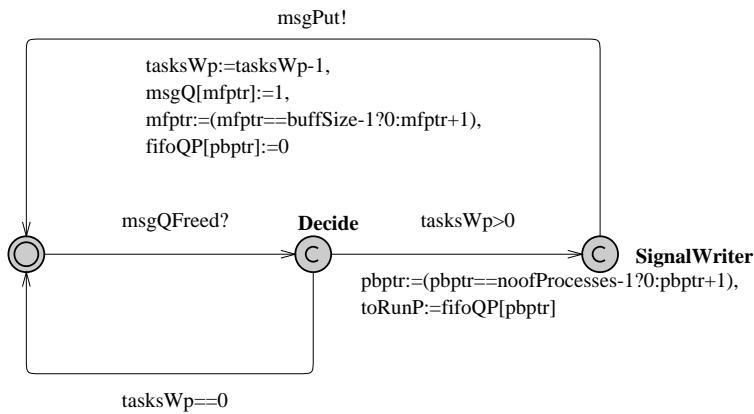


Figure 45: Dequeuing model for writer tasks

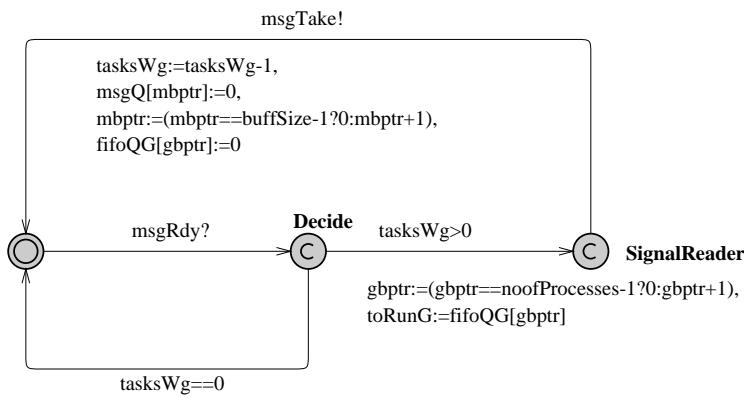


Figure 46: Dequeuing model for reader tasks

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

Figure 47: Timed Automata verification conditions for the Message Queue

- 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 signalled via $msgRdy$.
- When signalled via the channel $msgRdy$, the control is transferred to Figure 46.
- Utilizing the $gbptr$ and the $fifoQG$, the next reader process is decided, and signalled via $msgTake$.
- if the message queue is filled up, the writer process is queued up in the $fifoQP$, and the $pfptr$ is updated.

Figure 47 shows the timed automata verification conditions that are relevant for the Message Queue. The following paragraph describes the timed automata conditions shown in Figure 47. The number on the left specifies the serial number of the timed automata condition that is shown in Figure 47. 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.
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 $Updategptr$ in Figure 44, 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.

5.3 Modelling a User Program in UPPAAL

The given real-time system is mapped into a finite state machine with the help of timed automata to verify its correctness, and to detect the presence of race conditions. UPPAAL is an integrated tool environment for modelling,

validation and verification of real-time systems modelled as networks of timed automata. Hence, it facilitates the development of a real-time system in terms of its timed automata network models.

Utilizing our UPPAAL templates the user can model his tasks as described by the modelling of the example in Figure 48. Figure 48 shows two threads Thread-A and Thread-B each with two local variables *var-A* and *var-B*, two shared variables *Buf-A* and *Buf-B*. It is a simple code where the two threads communicate with each other via the global variables *Buf-A* and *Buf-B*. The code utilizes a semaphore to build the critical section.

The following steps need to be taken initially:

- Identify the shared data items between the different threads.
- Identify the inter process communication mechanisms that are utilized by the threads for communication.
- Identify the synchronization mechanisms utilized.
- Based on the type of communication mechanism utilized, the user can import any of our templates into his model to do the modelling of his task.

For example consider Figure 48, it consists of two threads thread-A and thread-B. The two threads share the common variables *Buf-A* and *Buf-B*. The critical section for each of the threads is modelled utilizing a semaphore. Each thread takes the semaphore before entering the critical section, and returns it when exiting it. In order to model this task on UPPAAL one has to initially model the semaphores to build the critical section. But since the semaphores are already modelled, the user can import our templates along with their declarations and reuse them. Figures 49 and 50 illustrate how the user threads shown in Figure 48 can be modelled.

Whenever a process requests a semaphore the UPPAAL model utilized will be similar to that shown in Figure 20 with the region in between the locations *SemTaken* and *CriticalSec* being the critical section. The location *CriticalSec* can be replaced by the user level code of the critical section. In Figure 48, thread-A and thread-B contain critical sections that are protected by the semaphore *sem*. This part is included on the edge in between the locations *SemTaken* and *CriticalSec* as shown in Figures 49 and 50.

In this example we have concentrated on modelling the critical section. But the model is extendable in other directions. If there exists code other than the critical section, in such a case, it can be inserted on the edges before the location *ReqSem* or after the location *ReSpawn*.

5.4 Race Condition Verification by utilizing UPPAAL

This section describes how TCTL can be utilized to detect race conditions in the UPPAAL models. Figure 48 shows an example of two threads communicating via a shared location utilizing a binary semaphore. Thread-A writes into the variable *Buf-A* and thread-B has to read it before thread-A writes another value into the shared variable. But the synchronization mechanism utilized cannot guarantee this, and it results in a race condition. This could be verified by utilizing the verification tool UPPAAL. Once the model is instrumented with the code, we can frame timed automata conditions to detect the occurrence of race conditions.

In the example above, we introduce a shared variable *written* that is utilized in the code of the critical section. Thread-A sets the variable whenever it writes to *Buf-A*, and Thread-B resets the variable whenever it reads the *Buf-A*. Once Thread-A sets the variable, and if it comes around to find that the variable *written* has been set then it implies that, it is trying to write to the location even before it is read by the other thread. This indicates the occurrence of a race condition. This needs to be verified by utilizing the model checker engine of the UPPAAL. Once the design is modelled as shown in Figures 49 and 50 the TCTL verification formula can be modelled as

Global	Thread_A	Thread_B
int Buf_A, Buf_B;	begin	begin
BinarySemaphore	{	{
Sem;	int Var_A;	int Var_B;
	while(1)	while(1)
	{	{
	:	:
	SemTake(Sem);	SemTake(Sem);
	Buf_A = Var_A;	Buf_B = Var_B;
	Var_A = Buf_B;	Var_B = Buf_A;
	SemGive(Sem);	SemGive(Sem);
	:	:
	}	}
	}	}

Figure 48: User Application to be modelled utilizing UPPAAL

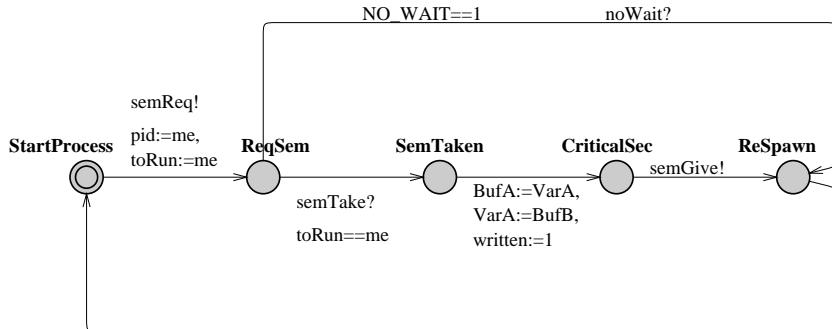


Figure 49: User Application thread A modelled utilizing UPPAAL

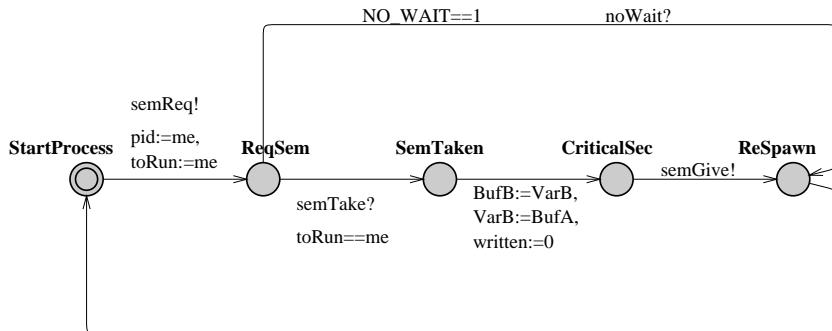


Figure 50: User Application thread B modelled utilizing UPPAAL

E<> (process1.SemTaken and written == 1).

Here *written* is utilized as the instrumentation variable which is set whenever the value is written to the *Buf-A*, and reset whenever the value is read. This TCTL condition tests if there is a path where the Thread-A has obtained the semaphore, and is about to write to the shared location. If the variable *written* is still set that implies that the process is attempting to write to the shared location even before it is read. This is a race condition. So if the formula shown above is satisfied then it implies the existence of a race condition.

6 A Case Study

This section presents a case study of a real-time application that reads a potentiometer. It illustrates the modelling of the systems in UPPAAL, and explains the verification of the race conditions in them.

Many embedded sensor applications are dependent on external environment for input. In many embedded applications, we need to know the position of the mechanical devices. Many sensors exploit a potentiometer to detect the position of the moving part. As the sensor moves, the resistance of the potentiometer changes (linearly) which, in turn, leads to variable voltage or current. A good example is the analog joystick. For each axis, a potentiometer is connected to a capacitor and a mono-stable multi-vibrator. Based upon the resistance, a pulse of variable width is generated, that determines the position of the joystick .

The potentiometer can be accessed from the user space by utilizing a driver routine. A driver provides all the system calls that can be utilized to open the device from the user space, and read the position of the sensors that are connected to it. This section presents the case study of a real-application. It illustrates the development of the application into a network of finite state automata models. It then describes the race condition verification in the model so developed.

The application is a driver application, that consists of the routines required to read a joystick from the user space. The joystick is connected to the game port of the Creative Labs SoundBlaster 16PCI card on a SA1100 board loaded with VxWorks operating system. The routines modelled include

- A user application (Figure 51) that spawns the periodic tasks that open the potentiometer from the user space and read the values.
- A driver install (Figure 52) routine that installs the driver, and allocates the semaphores that are required.
- An interrupt service routine (Figure 53) that is utilized to generate interrupts to read the position of the potentiometer.
- The re-entrant read (Figure 54) routine that is utilized to read the value, by utilizing the interrupt service routine and the timer.
- A user routine (Figure 55) that utilizes the driver routine to read the joy stick.
- The inter-task communication mechanisms (semaphores, message queues) that were modelled earlier by utilizing the tool UPPAAL are utilized to implement the inter-process communication, and are shown in Figures 59, 60, 61, 62, 63, 64, 56, 57, and 58 respectively.

In order to access a hardware device that is attached to the system, the following steps have to be taken.

- The driver code has to be written, and complied. It contains all the functions like open, close, read, write, and control routines.

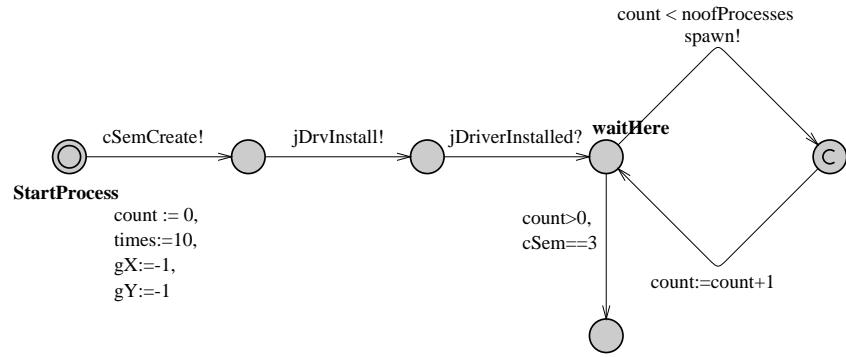


Figure 51: UPPAAL Model illustrating the User Level Application

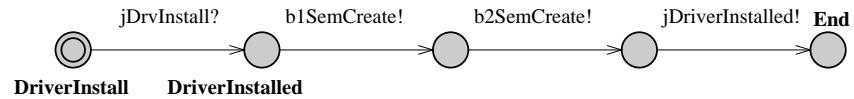


Figure 52: UPPAAL Model for the driver installation routine

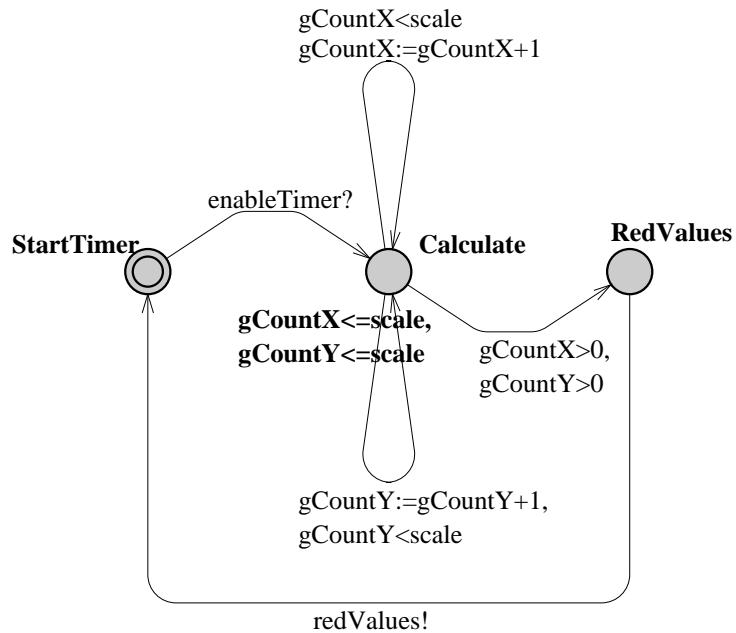


Figure 53: The Interrupt Service routine

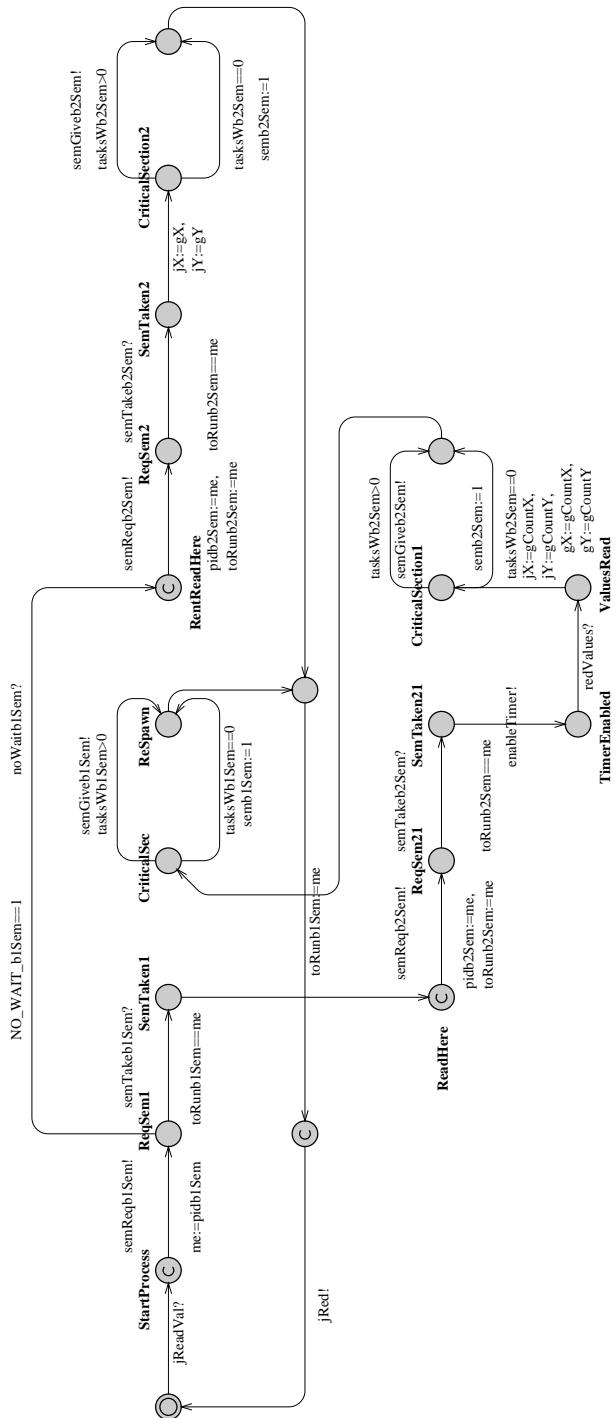


Figure 54: UPPAAL Model illustrating the reentrant read routine

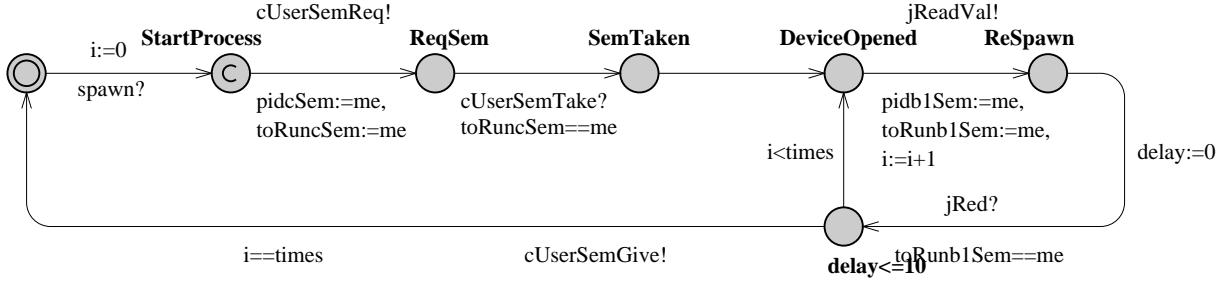


Figure 55: Task that uses the driver routine to read the device

- The driver has to be installed on the system. The operating system creates an entry for the driver in the driver table, and updates certain other data structures.
- The device has to be installed, and the drivers for it have to be specified. The operating system now inserts the device in the device table, and updates the data structures.
- To access the device from the user space, it has to be opened initially by utilizing the open system call. Once opened, the system call returns a file descriptor to the device that is just opened. This file descriptor can be utilized in all subsequent system calls to access the device.

6.1 UPPAAL Model of the Application

6.1.1 User application spawning tasks

- Figure 51 illustrates the user level application that spawns three user tasks that open the device, and test the re-entrant **jRead** routine.
- The *cSemCreate!* signals Figure 56 to initialize the semaphores that are utilized by the device driver.
- The *jDrvInstall!* installs the driver by invoking the driver installation routine in Figure 52.
- The *b1SemCreate!* and *b2SemCreate!* initialize the *b1* and *b2* semaphores as shown in Figures 59 and 62.
- Once the driver is installed, the driver installation routine (Figure 52) signals the current routine that the driver has been installed by utilizing the channel *jDriverInstalled?*.
- Once the driver is installed, the tasks that read the device are spawned by utilizing the channel *spawn!*, which, invokes the function *spawn* as in Figure 55 to spawn the periodic tasks.
- This routine waits in the location *waitHere* for all the tasks to give their corresponding *cUserSems* semaphores, after which it will terminate.

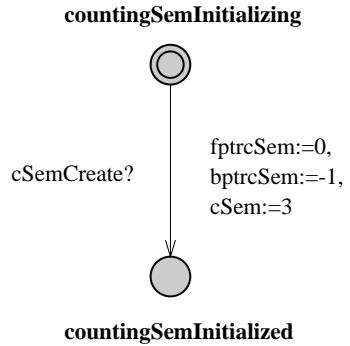


Figure 56: Initialization of the counting semaphore to control the user application

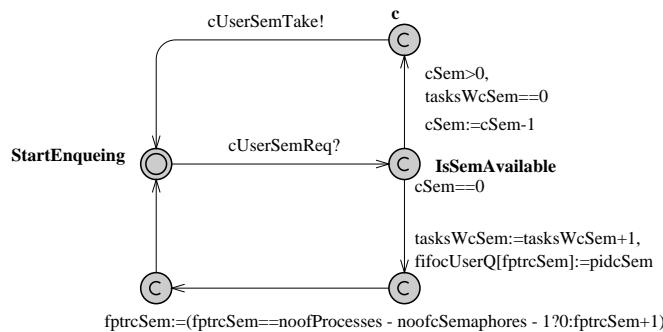


Figure 57: Enqueuing of tasks that are waiting for the counting semaphore

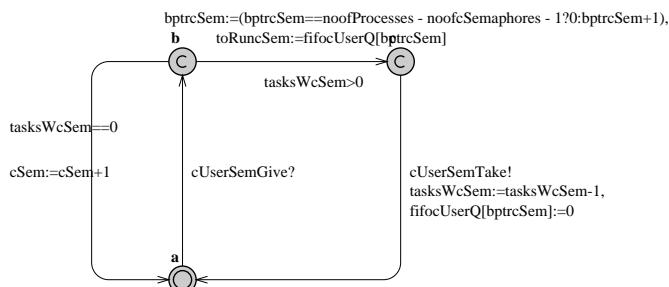


Figure 58: Dequeueing of tasks once a counting semaphore is available

6.1.2 Driver installation routine

- The *jDrvInstall* routine installs the driver as already mentioned.
- The control returns to the user application in Figure 51 via *jDriverInstalled?*.

6.1.3 User task that reads the potentiometer

- The task is created via the *spawn?* channel.
- The task now requests the semaphore *cUserSem* by utilizing the channel *cUserSemReq!*. The control is transferred to Figure 57. If the semaphore is available, the task can proceed forward. If the semaphore is not available, the task is queued.
- The task takes the semaphore via *cUserSemTake?*, and proceeds to open the device.
- Once the device is opened, it is read by utilizing the *jReadVal!*, which transfers the control to Figure 54.
- The value that is read from the *jRead* routine is updated in the global locations, from where the user application can access it. The *jRead* routine signals the current routine by utilizing *jRed?*.
- As the task is periodic, it waits in the current location for a specific time. The waiting task models a process that is sleeping, and can be awakened by utilizing a timer interrupt. Once the timer expires, the task can proceed to read the potentiometer again.
- This process continues for a specified amount of time, after which the task expires by returning the counting semaphore *cUserSem*.

6.1.4 Read function invoked by the kernel

- This function is invoked by the user application when it does the *read(...)* from the user space. Control is passed to this routine via the *jReadVal* channel.
- This function utilizes two semaphores, namely *b1*, *b2* to ensure re-entrancy.
- It requests the first semaphore by utilizing the *semReqb1Sem!*. If it is available the semaphore is taken, and the control proceeds to the location *SemTaken1* to enter the critical section that is guarded by the semaphore *b1*. If the semaphore is not available, it proceeds to the location *RentReadHere* where it is waiting for the semaphore *b2*, so that it can read the values set by the task that is currently present in the critical section that is guarded by the semaphore *b1*.
- If the task enters the location *SemTaken1*, it requests the semaphore *b2*, and gets blocked if it is not available. The semaphore is specified with the FIFO option so the tasks get queued into it in first-in-first-out fashion.
- If the semaphore is available, the task enters the critical section of *b2*.
- In the critical section that is guarded by both *b1* and *b2*, the task enables the timer by utilizing *enableTimer!* which passes the control to the timer module shown in Figure 53.
- Once the timer sets the values, the control returns to the *jRead* function by utilizing the channel *redValues*.

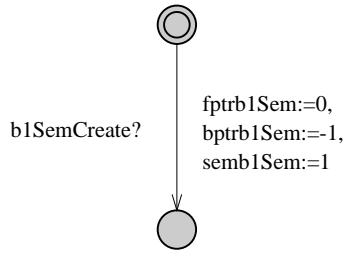


Figure 59: Initialization of the b1 semaphore utilized by the jRead function

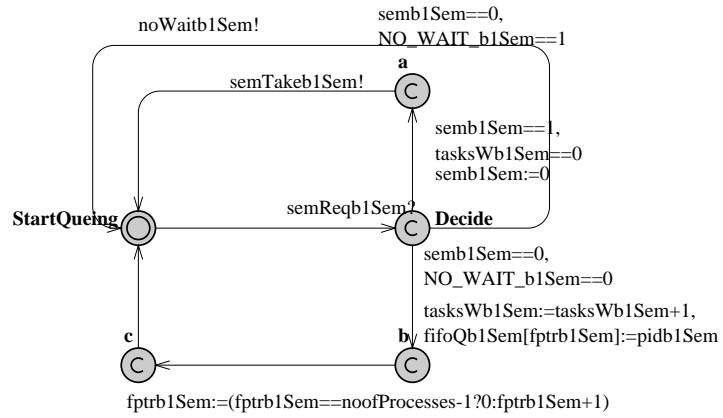


Figure 60: Enqueuing of tasks that are waiting for the semaphore b1

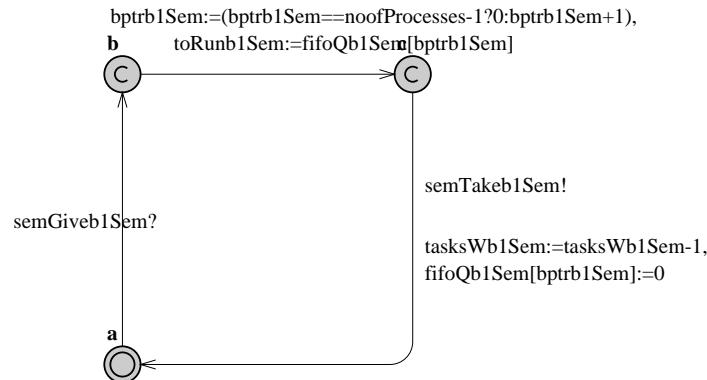


Figure 61: Dequeueing of tasks once a b1 semaphore is available

- The global values gX and gY are updated so that any task that is queued for the semaphore at the location *RentReadHere* can access the most recent values. It now gives away the semaphores $b2$ and then $b1$.
- Any task that is waiting for semaphore $b2$ can take it and update its local variables jX and jY with the global values, and then give away the semaphore $b2$.
- Each task, once it updates the jX and jY , gives back the semaphores that it has taken. Later, it signals the User Task that invoked the *jRead* by utilizing *jRed!*.

6.1.5 Timer and the corresponding Interrupt Service Routine

- The timer is enabled by the *jRead* function by utilizing the channel *enableTimer*.
- The timer is supposed to call an interrupt service routine for every tick. It is simulated in the same module in the location *Calculate*.
- In this location a certain non-determinism is introduced by utilizing variables $gCountX$ and $gCountY$. It is done in order to simulate the values that are set by the sensor.
- Once the values are set, the *jRead* routine is signaled by utilizing the channel *redValues*, and the control now transfers to the *jRead* function.

6.1.6 Inter-Process Communication Mechanisms Utilized

Three semaphores (see Section 4 and Section 5) namely *cUserSem*, $b1$ and $b2$ are utilized. Each semaphore has 3 modules associated with it namely, the initialization routine, the enqueueing module, and the dequeuing module. The initialization routines for the three semaphores are shown in Figures 56, 59, and 62 respectively. The enqueueing modules are shown in Figures 57, 60, and 63. The dequeuing modules are shown in Figures 58, 61, and 64.

6.1.7 Race conditions in the modelled application

In the application modelled above, when two or more processes are executing the code, the first process may take the semaphore $b1$, and even before it takes the semaphore $b2$, the second process may go ahead and take it. Now, the second process will read the values even before they are set by the process in the critical section guarded by $b1$. This is a typical general race condition, and can be tested by the following TCTL formulae.

E <> (jRead1.SemTaken2 and gX == -1)

This formulae tests to see if there exists a path where the process has reached the location *SemTaken2*, but the value of the gX is not yet set.

7 Examples

This section illustrates a number of examples that have been modelled 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.

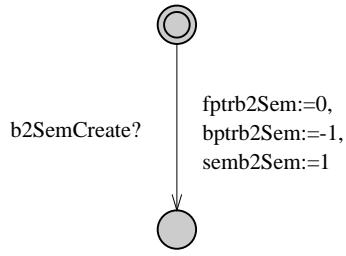


Figure 62: Initialization of the b2 semaphore utilized by the jRead function

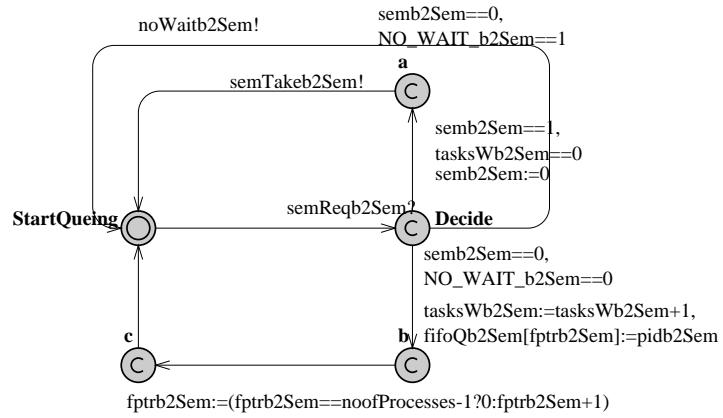


Figure 63: Enqueuing of tasks that are waiting for the semaphore b2

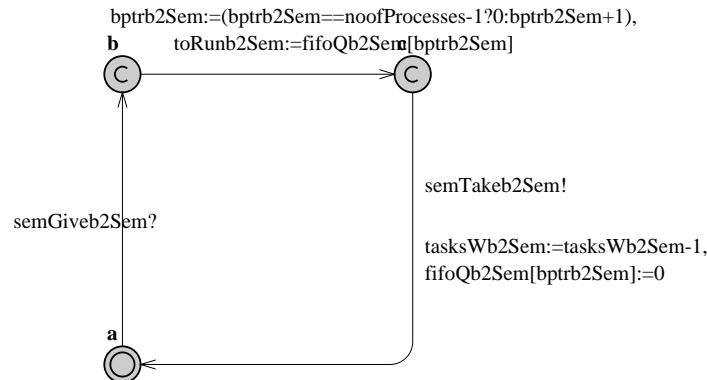


Figure 64: Dequeueing of tasks once a b2 semaphore is available

7.1 Example 1

Figure 65 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 signalling 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 65. 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 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 66 shows the UPPAAL model of the *thread-A* along with the code inserted into the model. Figure 67 shows the set of race conditions expressed in timed automata. The first condition in Figure 67 checks all the possible paths to see if the model is free from deadlocks. Figure 66 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*.

7.2 Example 2

In the example in Figure 65, 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 68, utilizes two different semaphores to force a thread in group A to wait until a thread in group B completes its task.

It utilizes four binary semaphores, two for each thread group, of which two are initially full, and two are empty. Initially the semaphores *Already* 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

```

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

Thread_A(...)

{
1      int Var_A;
2      while(1)
{
.....  

3      Var_A = ...;  

SemGive (B);  

4      SemTake (A);  

5      Buf_A = Var_A;  

6      Var_A = Buf_B;
.....  

7      }
}

Thread_B(...)

{
int Var_B;
while(1)
{
.....  

Var_B = ...;  

SemGive (A);  

SemTake (B);  

Buf_B = Var_B;  

Var_B = Buf_A;
....}
}

```

Figure 65: Example 1 illustrating Data Races

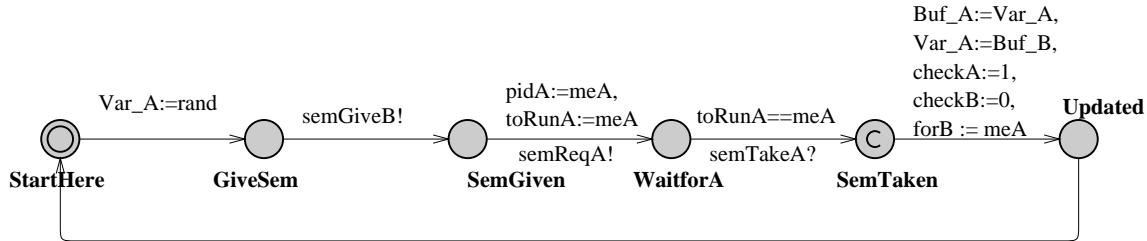


Figure 66: UPPAAL model for Thread-A, Example 1

-
1. A[] not deadlock
 2. E<> (Thread_A1.SemTaken and checkA == 1)
 3. E<> (Thread_A2.SemTaken and forA == 1)
-

Figure 67: UPPAAL verifier model Thread-A, Example 1

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 69 illustrates the UPPAAL model of the example in Figure 68. It utilizes two variables *forA* and *forB* to do the instrumentation similar to the example in Figure 69. The race conditions are illustrated in Figure 70.

7.3 Example 3

In this example shown in Figure 71, we utilize a mutual-exclusion semaphore to protect the shared variable. This makes sure that access to *Buf-A* and *Buf-B* is mutually exclusive. Before a thread can exchange a message it follows a handshaking protocol to update its own buffer at line 5 in Figure 71. It then performs a second hand-shaking protocol to receive a message from a thread in the other group. The utilization of mutual-exclusion semaphore prevents two threads in group A from accessing *Buf-A*, and *Buf-B* at the same time. This protection is not adequate to prevent general races.

There are two different identical handshaking mechanisms involved in the processes. The first handshaking mechanism is to update the shared data item and the second handshaking mechanism is to read the shared data item. Once each of the threads have updated their shared data items, and reach the line 6, the binary semaphores are empty, and the Mutual-exclusion semaphore is full, just as the case where the processes initially started at line 1. Now, either the two processes can proceed to completion, in which case, there is no race condition, or, two new processes one each of the two thread groups can get started, in which case, the shared buffers are over written by the new threads leading to a general race or, a thread in the old group may communicate with a thread in the new group leading to another general race condition. These race conditions can be alleviated by protecting the whole execution by mutual exclusion semaphores rather than each individual statement.

Figure 73 shows the UPPAAL model of the process, and the timed automata race condition detection mechanism is shown in Figure 72. The first condition in Figure 72 checks for deadlocks. The second condition checks to see if there exists a path where in *Thread-B* is writing to a location *Buf-B* even before *Thread-A* has read it. The third condition checks to see if there exists a path, wherein *Thread-A* is writing to a location *Buf-A*, even before *Thread-B* has read it. This is possible when there are multiple threads of the same sort, like the case where there exist A1, A2 belonging to thread group A and B1, B2 belonging to thread group B. If they were not to be duplicated then the race condition would not have occurred.

7.4 Example 4

In the above example, a thread in the thread group may ruin the message even before the previous message exchange completes. This can be prevented by expanding the critical section to include the complete message exchange sequence as in Figure 74.

This example utilizes a mutual exclusion semaphore, to make sure that the threads in the same group do not interfere

```

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 68: Example 2 illustrating Data Races

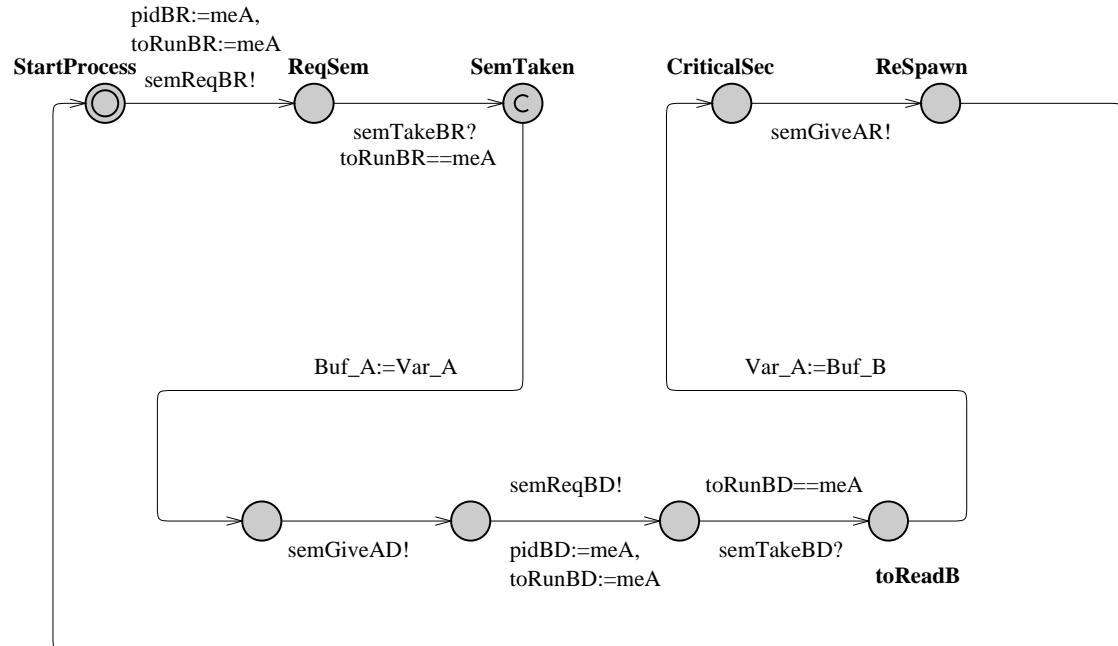


Figure 69: UPPAAL model for Thread-A, Example 2

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

Figure 70: UPPAAL verifier model Thread-A, Example 2

with each other while in progress. But this does not prevent general races. The *thread-A1* may write to the buffer *buf-A*, and take the semaphore given by *thread-B1*, and read the value set by it and exit the critical section guarded by *Already*. Meanwhile *thread-B1* is still blocked at line 6. Now, the semaphore *Already* is available, so *thread-A2* may take it and over-write the *buf-A*. This is a general race condition. The UPPAAL model for the system is shown in Figure 75. The timed automata conditions to detect the race conditions are similar to the above examples.

7.5 Example 5

One of the problems in the previous examples was that the buffer was being over-written even before it was read. This can be prevented by having multiple slots in the buffer. The reader process blocks till it finds a slot with a message and the writer process blocks itself till it finds a free slot in the buffer. This can be implemented by message queues. If implemented this would prevent general races.

The pseudo code is shown in Figure 76. The UPPAAL model is shown in Figure 77.

8 Conclusion

We have devised a methodology to map the various inter-process communication mechanisms in VxWorks to timed-automata models in the UPPAAL tool suite. The correctness of these models was verified using timed computational tree logic. We have presented a set of guidelines to convert any real-time system to finite-state concurrent systems with a time domain using the UPPAAL inter-process communication mechanism templates that we have developed. Timed Computational Tree Logic was utilized to specify and verify the properties and constraints of these real-time system models. We have also given an exhaustive 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 modelling 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] J. A. Stankovic, K. Ramamritham. "Tutorial: Hard Real-Time Systems". *IEEE Computer Society Press*, 1988.
- [2] 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.

```

Binary Semaphore A = 0, B = 0;
Mutex Semaphore Mutex = 1;
int       Buf_A, Buf_B;

Thread_A(...)
{
    int Var_A;

1     while(1)
    {
        ....
        SemGive (B);
        SemTake (A);
        SemTake(Mutex);
        Buf_A = Var_A;
        SemGive (Mutex);
        SemGive (B);
        SemTake (A);
        SemTake (Mutex);
        Var_A = Buf_B;
        SemGive (Mutex);
   12    ....
    }
}

Thread_B(...)
{
    int Var_A;

    while(1)
    {
        ....
        SemGive (A);
        SemTake (B);
        SemTake (Mutex);
        Buf_B = Var_B;
        SemGive (Mutex);
        SemGive (A);
        SemTake (B);
        SemTake (Mutex);
        Var_B = Buf_A;
        SemGive (Mutex);
    }
}

```

Figure 71: Example 3 illustrating general Races

-
1. A[] not deadlock
 2. E<> (Thread_A1.Check and checkA == 1)
 3. E<> (Thread_B.SemTaken and checkB == 1)
-

Figure 72: UPPAAL verifier model Thread-A, Example 3

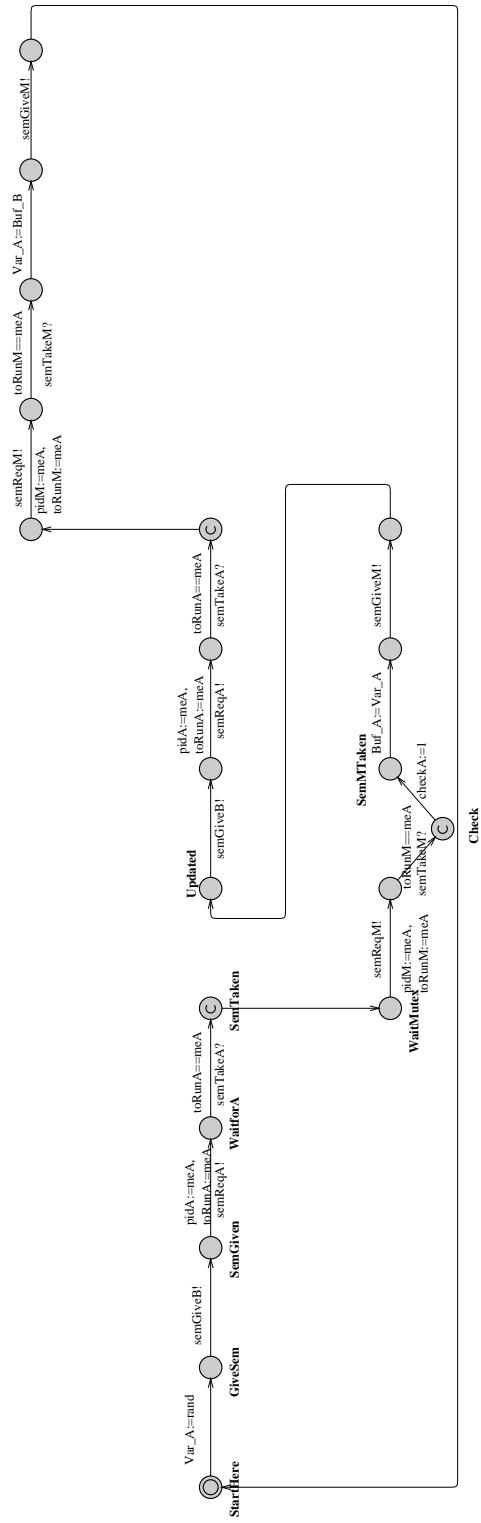


Figure 73: UPPAAL model Thread-A, Example 3

```

Mutex 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 (Aready);
3       Buf_A = Var_A;
4       SemGive (Adone);
5       SemTake (Bdone);
6       Var_A = Buf_B;
7       SemGive (Aready);
8       ...
9    }
}

Thread_B(...)

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

```

Figure 74: Example 4

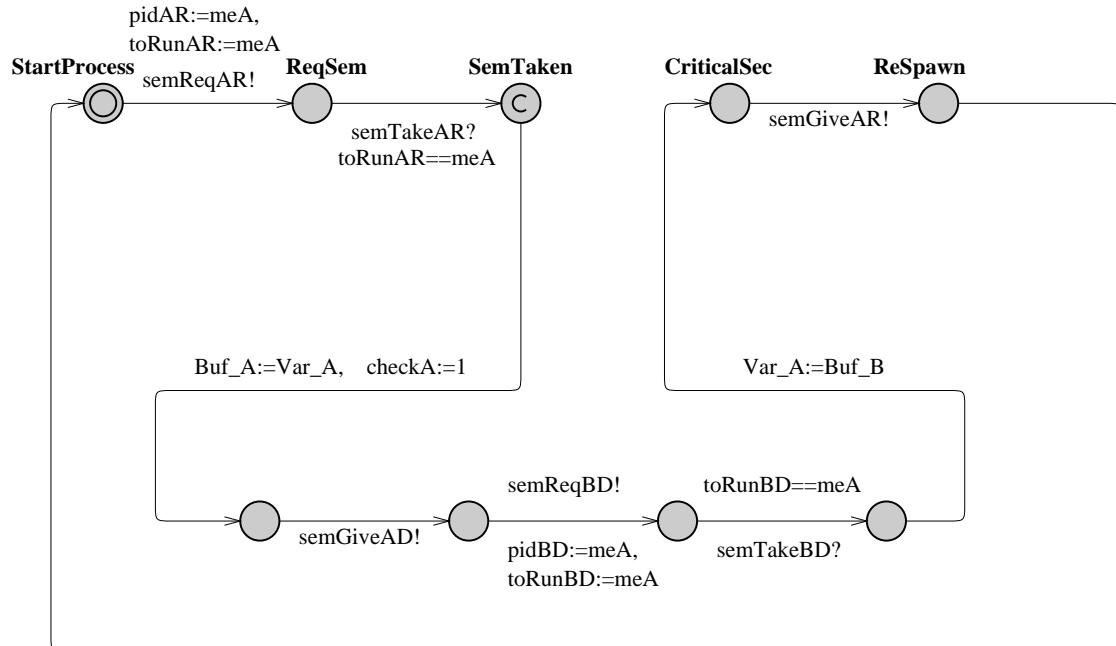


Figure 75: UPPAAL model for Thread-A, Example 4

```

Semaphore Aready = 1, Bready = 1;
int           Buf_A, Buf_B;

Thread_A(...)

{
    int Var_A;
    1   while(1)
    {
        ....
        2   SemTake (Amutex);
        3       PUT(Var_A, Buf_A);
        4       GET(Var_A, buf_B);
        5   SemGive (Amutex);
        ...
    }
}

Thread_B(...)

{
    int Var_B;
    while(1)
    {
        ....
        SemTake (Bmutex);
        PUT(Var_B, Buf_B);
        GET(Var_B, buf_A);
        SemGive (Bmutex);
        ...
    }
}

```

Figure 76: Example 5, free from races

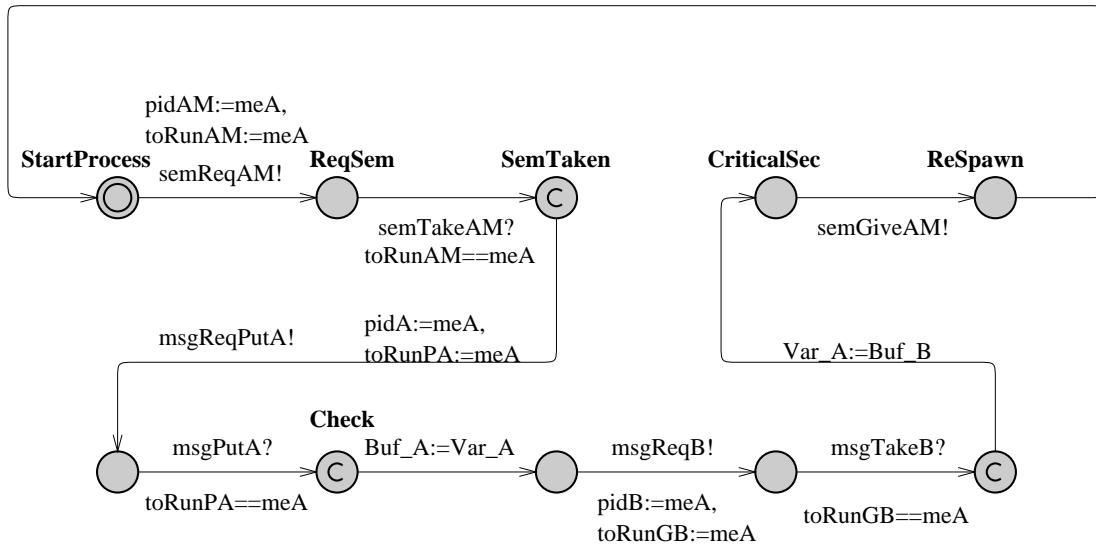


Figure 77: UPPAAL model for Thread-A, Example 5

- [3] 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.
- [4] Dejan Perković 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 1996.
- [5] A. J. Bernstein. "Analysis of programs for parallel processing, Vol. 15". *IEEE Transactions on Electronic Computers*, pages 757–762, 1966.
- [6] 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.
- [7] 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.
- [8] Alur, R. and Dill, D. L. "A Theory of Timed Automata". *Theoretical Computer Science*, 126(2), pages 183–235, 1994.
- [9] Kim G. Larsen, Paul Pettersson, Wang Yi. "UPPAAL in a Nutshell". *Springer International Journal of Software Tools for Technology Transfer* 1(1+2), 1997.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.

- [19] 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.
- [20] 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.
- [21] "VxWorks 5.4 Programmers Guide Version 1".