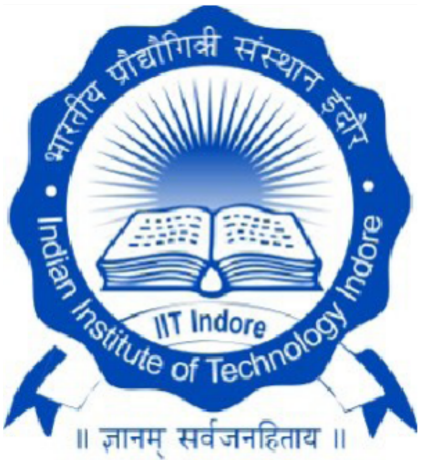
B.TECH PROJECT REPORT

ON

Formal Verification of Autonomous Systems

BY

AKASH ANAND KAMAT



DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2017

Formal Verification of Autonomous Systems

A PROJECT REPORT

Submitted in partial fulfillment of the
requirements for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

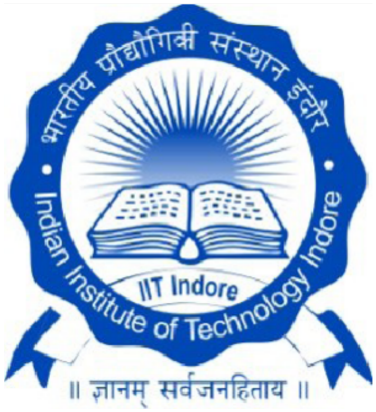
Submitted by :

AKASH ANAND KAMAT

Guided by :

DR. GOURINATH BANDA

(Associate Professor, IIT INDORE)



INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2017

CANDIDATE'S DECLARATION

I hereby declare that the project entitled **Formal Verification of Autonomous Systems** submitted in partial fulfilment for the award of the degree of Bachelor of Technology in **Computer Science and Engineering** completed under the supervision of **Dr. Gourinath Banda, Associate Professor, Computer Science and Engineering, IIT Indore** is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Akash Anand Kamat
140001003
Discipline of Computer Science and Engineering
IIT Indore

CERTIFICATE by BTP Guide

It is certified that the declaration made by the student is correct to the best of my knowledge and belief.

Dr.Gourinath Banda
Associate Professor
Discipline of Computer Science and Engineering
IIT Indore

PREFACE

This report on **Formal verification of Autonomous systems** is prepared under the supervision of Dr. Gourinath Banda, Associate Professor, Computer Science and Engineering, IIT Indore.

Through this report, I have tried to provide a detailed description of the technologies that have been used to design the proposed protocol and to produce an efficient novel algorithm to implement the same.

I have also tried to explain the proposed approach in detail and the results with each approach is also discussed.

ACKNOWLEDGEMENT

I would like to thank my BTP supervisor, **Dr. Gourinath Banda**, for his constant support in structuring the project and for his valuable feedback which helped me in the course of the project. He gave me the opportunity to discover and work in this domain. He guided me thoroughly and pulled me out of crates of failures I faced all through the period.

I am thankful to my BTP partner Utkarsh Saxena for his contribution and constant support throughout the project.

It is their help and support, due to which I was able to complete the design and technical report. Without their support, this report would not have been possible.

ABSTRACT

An autonomous system is a system that performs behaviours or tasks with a high degree of autonomy. These systems are used in robotics, human-robot teamwork, pervasive systems, autonomous road vehicles and in many more areas. Formal Verification is a powerful technique for finding software/hardware errors. It provides mathematical proof of absence of errors in implementations with respect to its specifications.

Model Checking is a verification technique that explores all possible system states. In software and hardware design of complex systems, more time and effort are spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time. Model-based verification techniques are based on models describing the possible system behaviour in a mathematically precise and unambiguous manner.

In our project, we explained the importance of scheduling in autonomous systems and how formal verification can be applied to analyse resource scheduling protocols. For case study, we considered the *Stack-based priority ceiling* (SBPC) protocol. This protocol involves integrated task and resource scheduling. We have modelled and implemented this protocol and verified it against multiple properties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Autonomous Systems

An autonomous system is a system that performs behaviours or tasks with a high degree of autonomy, which is particularly desirable in fields such as spaceflight, household maintenance (such as cleaning), waste water treatment, delivering goods and services, etc.

Some modern factory systems are "autonomous"within the strict confines of their direct environment. It may not be that every degree of freedom exists in their surrounding environment, but the factory system's workplace is challenging and can often contain chaotic, unpredicted variables. The exact orientation and position of the next object of work and (in the more advanced factories) even the type of object and the required task must be determined. This can vary unpredictably.

1.2 Significance of Autonomous Systems in modern times

Autonomous Systems is used in a lot of areas. Some of them are listed below:

- **Robotics and Robot Swarms:** As we move from the restricted manufacturing robots seen in factories towards robots in the home and robot helpers for the elderly, so the level of autonomy required increases.

- **Human-Robot Teamwork:** Once we move beyond just directing robots to undertake tasks, they become robotic companions. In the not too distant future, we can foresee teams of humans and robots working together but making their decisions individually and autonomously.
- **Pervasive Systems, Intelligent Monitoring, etc:** As sensors and communications are deployed throughout our physical environment and in many buildings, so the opportunity to bring together a multiplicity of sensor inputs has led to autonomous decision-making components that can, for instance, raise alarms and even take decisive action.
- **Autonomous Road Vehicles:** Also known as "driver-less cars", autonomous road vehicles have progressed beyond initial technology assessments (e.g. DARPA Grand Challenges) to the first government-licensed autonomous cars [3].

1.3 Challenges faced by Autonomous Systems in modern times

An autonomous system gains information about the environment. It must be deployed in remote environments where direct human control is infeasible or should be involved in activities that is too lengthy and/or repetitive to be conducted successfully by humans. It must be deployed in hostile environments where it is dangerous for humans to be nearby, and so difficult for humans to assess the possibilities. Also, it needs to react much more quickly than humans can. Last but most important, such autonomous systems need to be proved for their correctness before deployment.

1.4 Testing and Formal Verification

Testing

It can only detect presence of errors but it cannot find all errors, i.e.; error needs to be specified to check whether it exists. It is much cheaper than verification.

Formal Verification

It is a powerful method for finding software errors. It provides mathematical proof of absence of errors in implementations relative to specifications. Formal specification and analysis are often very expensive and requires highly qualified engineers. Automated techniques for formal verification are rather limited.

Formal Verification vs Testing

Verification is often used in early stages of development and testing in later stages but test cases can be developed during the specification phase. Verification presupposes formal program semantics whereas testing does not. Verification is often based on abstraction, thus it is also only a necessary correctness criterion.

System tests go beyond verification, since real environment is involved. Testing is strongly used in software engineering: up to 40% of software development efforts go into it. Formal verification is rarely used in practice.

Chapter 2

Modelling Formalisms for Autonomous Systems

2.1 Layered Architecture of Autonomous Systems

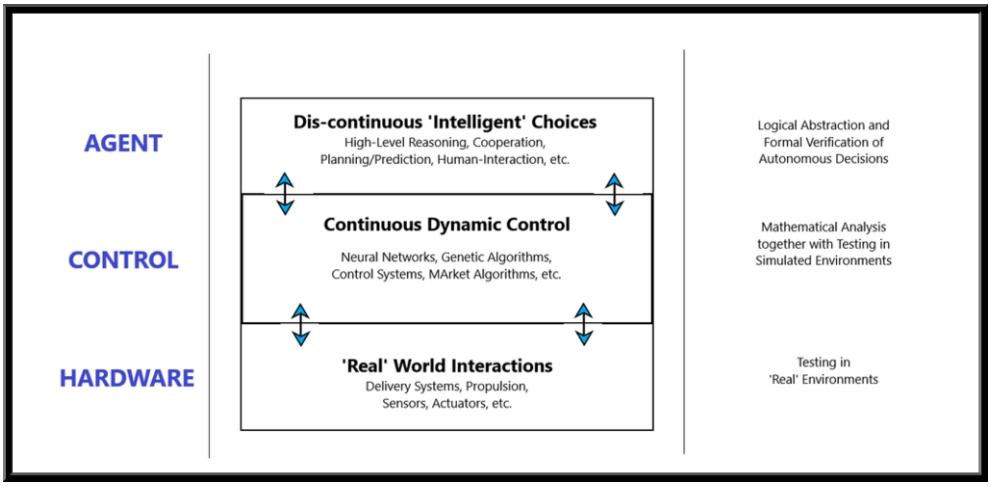


Figure 2.1: Layered Architecture of Hybrid Autonomous Systems

Many autonomous systems, ranging over unmanned aircraft, robotics, satellites and even purely software applications, have a similar internal structure, namely layered architectures as summarised in Figure 2.1. Although

purely connectionist/sub-symbolic architectures remain prevalent in some areas, such as robotics, there is a broad realisation that separating out the important/difficult choices into an identifiable entity can be very useful for development, debugging, and analysis. While such layered architectures have been investigated for many years they appear increasingly common in autonomous systems.

The system in Figure 1 is split into real-world interactions, continuous control systems, and discontinuous control. For example, a typical unmanned air system might incorporate an aircraft, a set of control systems encapsulated within an autopilot, and a high-level decision-maker that makes the key 'choices'. Once a destination has been decided, the continuous dynamic control, in the form of the autopilot, will be able to fly there. The 'intelligence' only becomes involved if either an alternative destination is chosen, or if some fault or unexpected situation occurs [3].

2.2 Importance of scheduling and resource allocation in autonomous systems

Scheduling and resource allocation is used in almost every autonomous system. In a critical operation the task must be processed in the time specified by the deadline.

A task in an autonomous system must be completed on time. A system is said to be not schedulable when tasks can not meet the specified deadlines.

2.3 Automatic Verification Techniques

Listed below are the most commonly used automatic verification techniques.

- Temporal Logics
- ω - Automata
- Model Checking
- Petri Nets
- Milner Calculus of Communicating Systems
- Hoare's Theory of communicating sequential processes

2.4 Model Checking

Model checking is a verification technique that explores all possible system states. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property [2].

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in online airline reservation systems, modern e-commerce protocols have been verified, and several studies of international IEEE standards for in-house communication of domestic appliances have led to significant improvements of the system specifications.

2.5 Timed Automata

Timed automata models the behaviour of time-critical systems. A timed automaton is in fact a program graph that is equipped with a finite set of real-valued clock variables.

During a run of a timed automaton, all clock values increase with the same speed. Along the transitions of the automaton, clock values can be compared to integers. These comparisons form guards that may enable or disable transitions and by doing so constrain the possible behaviours of the automaton. Further, clocks can be reset. Timed automata are a sub-class of a type hybrid automata.

Formal Definition

Formally, a timed automaton is a tuple $A = (Q, \Sigma, C, E, q_0)$ that consists of the following components:

- Q is a finite set. The elements of Q are called the states of A.
- Σ is a finite set called the alphabet or actions of A.
- C is a finite set called the clocks of A.
- $E \subseteq Q \times \Sigma \times B(C) \times P(C) \times Q$ is a set of edges, called transitions of A, where

  - B(C) is the set of boolean clock constraints involving clocks from C
  - P(C) is the powerset of C.

- q_0 is an element of Q, called the initial state.

An edge (q,a,g,r,q') from E is a transition from state q to q' with action a, guard g and clock resets r [4].

2.6 Property Specification Languages

Property Specification Language (PSL) is a temporal logic extending Linear temporal logic with a range of operators for both ease of expression and enhancement of expressive power. PSL makes an extensive use of regular expressions and syntactic sugaring.

It is widely used in the hardware design and verification industry, where formal verification tools (such as model checking) and/or logic simulation tools are used to prove or refute that a given PSL formula holds on a given design. Examples for such languages are CTL, LTL, CTL*, etc.

2.7 Temporal Logics

In logic, temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. In a temporal logic we can then express statements like "I am always hungry", "I will eventually be hungry", or "I will be hungry until I eat something". Temporal logic is sometimes also used to refer to tense logic, a particular modal logic-based system of temporal logic.

Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems. For instance, one may wish to say that whenever a request is made, access to a resource is eventually granted, but it is never granted to two requesters simultaneously. Such a statement can conveniently be expressed in a temporal logic.

2.8 Linear Temporal Logic (LTL)

In logic, linear temporal logic or linear-time temporal logic (LTL) is a modal temporal logic with modalities referring to time. In LTL, one can encode k-formulas about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc.

It is a fragment of the more complex CTL*, which additionally allows branching time and quantifiers. Subsequently LTL is sometimes called propositional temporal logic.

LTL is built up from a finite set of propositional variables AP, the logical operators $\neg$ and $\vee$, and the temporal modal operators X (some literature uses O or N) and U. Formally, the set of LTL formulas over AP is inductively defined as follows:

- if p $\in$ AP then p is an LTL formula
- if Ψ and φ are LTL formulas then $\neg\Psi$, $\varphi \vee \Psi$, $X\Psi$, and $\varphi \cup \Psi$ are LTL formulas.
- X is read as next and U is read as until. Other than these fundamental operators, there are additional logical and temporal operators defined in terms of the fundamental operators to write LTL formulas succinctly. The additional logical operators are $\wedge, \rightarrow, \leftrightarrow$, true and false. Following are the additional temporal operators.
  - G for always (globally)
  - F for eventually (in the future)
  - R for release
  - W for weakly until

Semantics

An LTL formula can be satisfied by an infinite sequence of truth evaluations of variables in AP. These sequences can be viewed as a word on a path of a Kripke structure (an ω - word over alphabet 2^{AP}). Let w = a_0, a_1, a_2,... be such an ω - word. Let w(i) = a_i. Let $w_i = a_i, a_{i+1}$,..., which is a suffix of w. Formally, the satisfaction relation $\vDash$ between a word and an LTL formula is defined as follows:

- w $\vDash$ p if p $\in$ w(0)
- w $\vDash \neg\Psi$ if w $\nvDash \Psi$
- w $\vDash \varphi \vee \Psi$ if w $\vDash \varphi$ or w $\vDash \Psi$
- w $\vDash X\Psi$ if $w_1 \vDash \Psi$ (in the next time step Ψ must be true)
- w $\vDash \varphi \cup \Psi$ if there exists i ≥ 0 such that $w_i \vDash \Psi$ and for all $0 \leq$ k $<$ i, $w_k \vDash \varphi$ (φ must remain true until Ψ becomes true)

2.9 CTL (Computation Tree Logic)

Computation tree logic (CTL) is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realised. It is used in formal verification of software or hardware artefacts, typically by software applications known as model checkers which determine if a given artefact possesses safety or liveness properties.

For example, CTL can specify that when some initial condition is satisfied (e.g., all program variables are positive or no cars on a highway straddle two lanes), then all possible executions of a program avoid some undesirable condition (e.g., dividing a number by zero or two cars colliding on a highway). In this example, the safety property could be verified by a model checker that explores all possible transitions out of program states satisfying the initial condition and ensures that all such executions satisfy the property.

The language of well formed formulas for CTL is generated by the following grammar:

$$\begin{aligned}\phi ::= {} & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \\ \mid {} & \mathbf{AX}\,\phi \mid \mathbf{EX}\,\phi \mid \mathbf{AF}\,\phi \mid \mathbf{EF}\,\phi \mid \mathbf{AG}\,\phi \mid \mathbf{EG}\,\phi \mid \mathbf{A}\,[\phi \mathbf{U}\ \phi] \mid \mathbf{E}\,[\phi \mathbf{U}\ \phi]\end{aligned}$$

where p ranges over a set of atomic formulas. It is not necessary to use all connectives - for example

$\{\neg, \wedge, \mathbf{AX}, \mathbf{AU}, \mathbf{EU}\}$

comprises a complete set of connectives, and the others can be defined using them.

- A means 'along All paths' (Inevitably)

- E means 'along at least (there Exists) one path' (possibly)

The logical operators are the usual ones: Along with these operators CTL formulas can also make use of the boolean constants true and false.

- Quantifiers over paths: Ψ
  - A Φ - All: Φ has to hold on all paths starting from the current state.
  - E Φ - Exists: there exists at least one path starting from the current state where Φ holds.
- Path - specific quantifiers:
  - X Φ - Next: Φ has to hold at the next state (this operator is sometimes noted N instead of X).
  - G Φ - Globally: Φ has to hold on the entire subsequent path.
  - F Φ - Finally: Φ eventually has to hold (somewhere on the subsequent path).
  - Φ U Ψ - Until: Φ has to hold at least until at some position Ψ holds. This implies that Ψ will be verified in the future.
  - Φ W Ψ - Weak until: Φ has to hold until Ψ holds. The difference with U is that there is no guarantee that Ψ will ever be verified. The W operator is sometimes called "unless".

2.10 Model Checking: CTL vs LTL

Computation tree logic (CTL) and Linear temporal logic (LTL) are both a subset of CTL*. CTL and LTL are not equivalent and they have a common subset, which is a proper subset of both CTL and LTL. Properties specified in CTL are verified in UPPAAL and properties specified in LTL are verified in SAL.

UPPAAL vs SAL:

Property	UPPAAL	SAL
Type of model checker	Real-time	Symbolic, bounded and infinite
Modelling language	Timed Automata, C subset	SAL
Properties Language	TCTL	LTL
Counter Example Generation	Yes	Yes
GUI	Yes	No
Graphical Specification	Yes	No
Counter Example Visualisation	Yes	No
Programming language used	C++, Java	Scheme

Table 2.1: CTL(UPPAAL) vs LTL(SAL)

2.11 Popular Model Checking Tools

- UPPAAL
- SAL
- SATABS
- RED
- SPIN
- APMC

- ARC
- DIVINE
- JAVA Pathfinder
- LLBMC
- Open MP C Analysis

Chapter 3

Stack-Based Priority Ceiling (SBPC) Protocol

3.1 Priority Ceiling Protocol

Considering a uniprocessor system, a real-time application is modelled as a finite set of tasks that run concurrently on a single processor. A task consists of a succession of jobs, each requiring a finite amount of computation. In other words, the computation associated with each job should terminate. Each task is characterised by its priority, period, and by the amount of processing it requires.

Other resources than the processor are shared by different jobs. For instance, the jobs may share common I/O channels or communicate with each other via shared variables. Access to these shared resources is controlled by binary semaphores which ensure mutual exclusion. The operations for requesting and releasing the lock on a semaphore S are denoted by lock(S) and unlock(S), respectively.

Each task has fixed priority, the priority of a job is the priority of the task to which it belongs. Jobs are executed in priority order: if two jobs of different priorities are ready to run at the same time, then the one with higher priority is allocated to the processor. It is wise to note that different tasks may have the same priority, although priority assignment techniques such as the rate monotonic or the deadline monotonic approaches avoid this situation by giving different priorities to different tasks.

When synchronisation primitives, such as semaphores, are used, there is

a problem called priority inversion which causes low priority jobs to prevent higher priority jobs from running. For instance, a job j can be blocked when trying to lock a semaphore S if a job k of lower priority has locked S before j was dispatched. As a result, a job j of top priority can be unable to execute and a job k of lower priority than j can become active. This phenomenon may block j for long periods of time, since other jobs, with priority greater than k, may prevent k to execute and consequently to unlock S. So, the high-priority job j can then be delayed by the low-priority job k that locks S but also by any job of intermediate priority that might preempt k. Since high-priority jobs are usually the most urgent and may have tight deadlines, such unrestricted priority inversion can be disastrous.

In the Priority Ceiling Protocol, the following approach is used to cope with this problem:

- Each semaphore S is assigned a fixed ceiling which is equal to the highest priority among the jobs that need access to S.
- A job j executing lock(S) is granted access to S if the priority of j is strictly higher than the ceiling of any semaphore locked by a job other than j. Otherwise, j becomes blocked and S is not allocated to j.

3.2 Definition of SBPC

Suppose all jobs share a single stack. When a job J is preempted by a job J', J continues to hold its stack space and J1 is allocated space immediately above it on the stack. The only special requirement is that if J is preempted it cannot resume execution until all the jobs that occupy stack space above it have completed. Since these jobs must have higher priority, this requirement is consistent with priority scheduling [5].

3.2.1 Rules of SBPC

Update of the Current Ceiling: Whenever all the resources are free, the ceiling of the system is Ω. The ceiling $\pi(t)$ is updated each time a resource is allocated or freed.

Scheduling Rule: After a job is released, it is blocked from starting execution until its assigned priority is higher than the current ceiling π(t) of the system. At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

Allocation Rule: Whenever a job requests a resource, it is allocated the resource.

3.2.2 Properties of SBPC

P1: When a job begins to execute, all the resources it will ever need during its execution are free. Otherwise, if one of the resources it will need is not free, the ceiling of the system is equal to or higher than its priority.

P2: No job is ever blocked once its execution begins.

P3: Deadlock Freedom: Deadlock is a state in which each member of a group is waiting for some other member to take action. SBPC protocol ensures no deadlock occurs.

3.3 Specification of SBPC Properties in CTL

P1: AG P_i.begin $\Rightarrow$ Q_{P_i}.free, where P_i denotes process 'i' and Q_{P_i} denotes that all resources used by P_i.

P2: AG P_i.begin $\Rightarrow$ (not P_i.blocked U P_i.end)

P3: AG not deadlock

Chapter 4

Modelling SBPC Protocol in UPPAAL

4.1 UPPAAL

UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays etc.).

The tool has been developed in collaboration between the Design and Analysis of Real-Time Systems group at Uppsala University, Sweden and Basic Research in Computer Science at Aalborg University, Denmark.

It is appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables.

Uppaal consists of three main parts: a description language, a simulator and a model-checker.

- The **description language** is a non-deterministic guarded command language with data types (e.g. bounded integers, arrays, etc.). It serves as a modelling or design language to describe system behaviour as networks of automata extended with clock and data variables.

- The **simulator** is a validation tool which enables examination of possible dynamic executions of a system during early design (or modelling) stages and thus provides an inexpensive mean of fault detection prior to

verification by the model-checker which covers the exhaustive dynamic behaviour of the system.

- The **model-checker** can check invariant and reachability properties by exploring the state-space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

4.2 Modelling SBPCP in UPPAAL

4.2.1 Deterministic Model:

Assumptions

- Uniprocessor system is used.
- Each task has a fixed priority, known before its execution.
- Each resource can be used by only one processor at a time.
- Scheduler has knowledge of which tasks will arrive beforehand, as well as the various resources used by them.
- The deadlines of resource lock/unlock event by each task is known beforehand.

Model Description

The model consists of three automata namely Task, Scheduler and Resource. Every Task and Resource is represented by its own system whereas there is only one scheduler system.

The model requires the following inputs:

- Number of tasks
- Number of resources
- For each task:
  - Arrival time
  - Computation time

  - Priority
  - Resource lock time for each resource it uses(relative to execution start time)
  - Resource unlock time for each resource it uses(relative to execution start time)

- Initial value of system ceiling = -1 (i.e. System ceiling when all resources are free).

In the model, two Priority Queues are used, one for tasks which are blocked and other for non-blocked tasks. Both queues are ordered using task priorities.

Every time a task arrives it is either added to blocked queue if it is blocked (i.e. its priority is less than system ceiling) or to the non-blocked queue if its not blocked.

Task Automaton

Each task requires finite amount of computation and has a fixed priority known beforehand by the scheduler. For each task, resource locking and unlocking are events, performed at specific time of its execution, i.e. the program counter is used for resource locking and unlocking.

A task is modelled to have 5 states namely:

- **Idle:** Task is not ready to execute.
- **Ready:** Task is ready to execute i.e. it has arrived.
- **Running:** The scheduler has scheduled the task to run. During this state the program counter increases and whenever it equals the timestamp of an event it transitions to a committed state where the type of event is evaluated i.e. resource locking, resource unlocking or task end. Upon resource locking or unlocking it transitions to another committed state locking or unlocking the resource synchronising with the resource system via allocate or release channel respectively, then returns back to the Running state.
- **Preempted:** The task has been preempted. The program counter is frozen in this state.

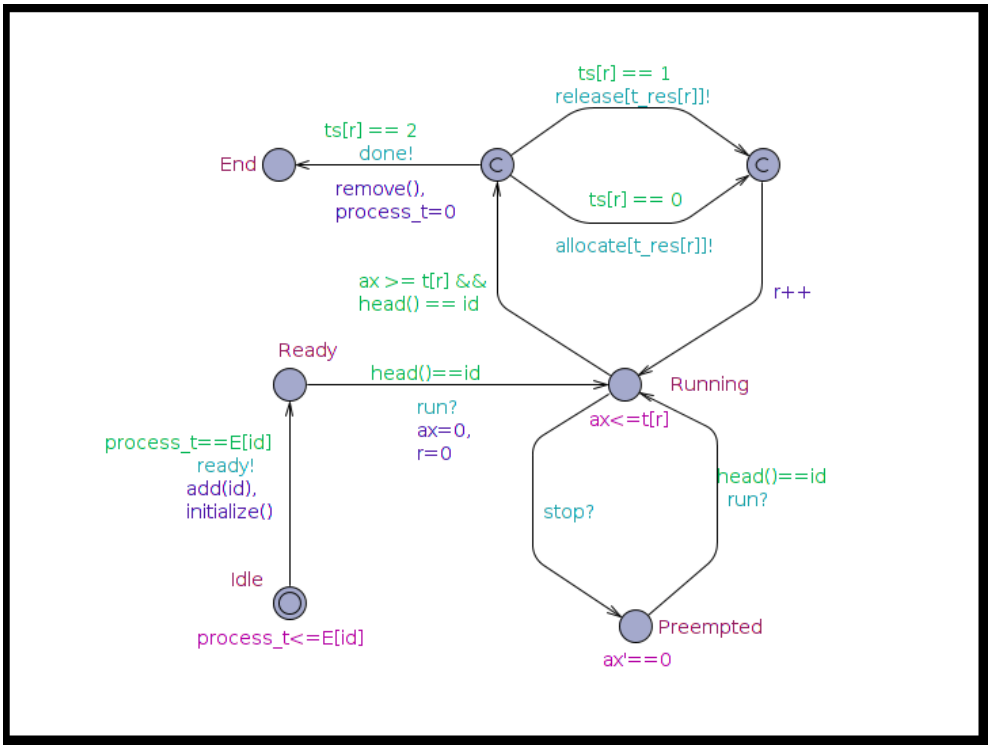


Figure 4.1: Task automaton (deterministic) in UPPAAL

- **End:** Task has finished execution.

Resource Semaphore Automaton

Resources are modelled as Binary semaphore. Each resource has the following states:

- **Free:** The resource is free, whenever a task locks a resource this transitions to 'Used' state while updating the system ceiling 'pi' and 'pi_stack' if necessary.
- **Used:** The resource is locked. When a task unlocks a resource this system transitions to a committed state, updates the system ceiling 'pi' and 'pi_stack' if necessary. If the system ceiling is updated the blocked-tasks and non-blocked-tasks queue are also updated, then the system transitions to 'Free' state synchronising with the scheduler via the 'resource_release' channel.

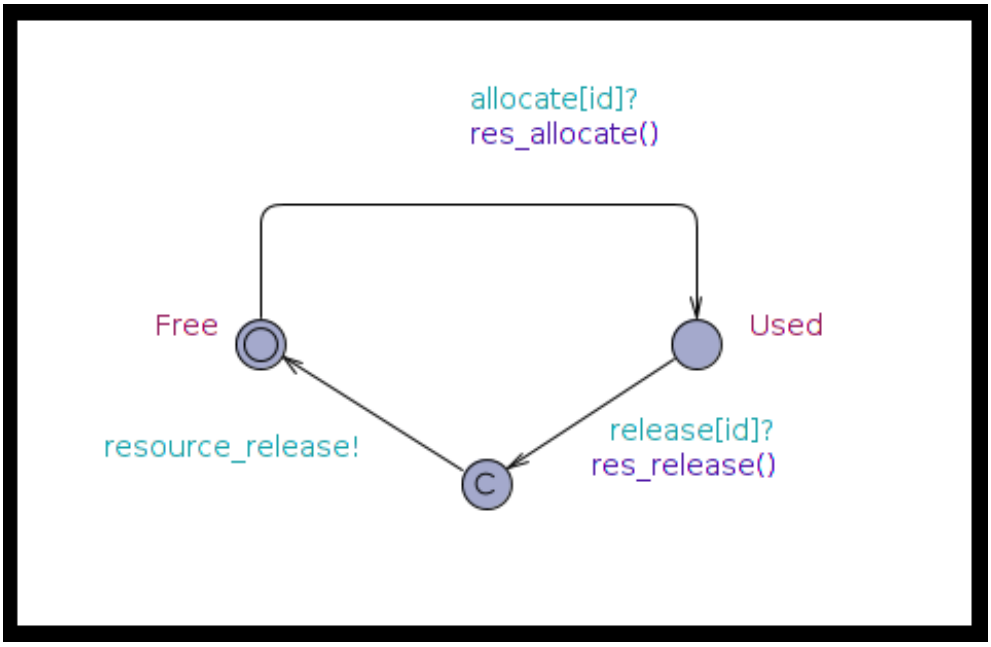


Figure 4.2: Resource semaphore automaton (deterministic) in UPPAAL

Scheduler Automaton

The scheduler initialises all the resource ceilings then transitions to 'Free' state.

- **Free:** The system has no tasks scheduled to run currently. If there are no ready to execute tasks, it transitions to 'Waiting' state otherwise it transitions to 'Select' state.
- **Waiting:** The system is idle and is waiting for a task to arrive. When a task is ready to be executed it (i.e. Task.Idle to Task.Ready) synchronises with the scheduler via 'ready' channel and the Scheduler transitions from 'Waiting' to 'Select' state.
- **Select:** The non-blocked-tasks queue is non empty, the scheduler selects the highest priority task from the non-blocked-task queue for running. The system transitions to 'Occupied' state synchronising with the selected task via the 'run' channel.
- **Occupied:** The system is occupied and a task is running. Three transitions are possible here:

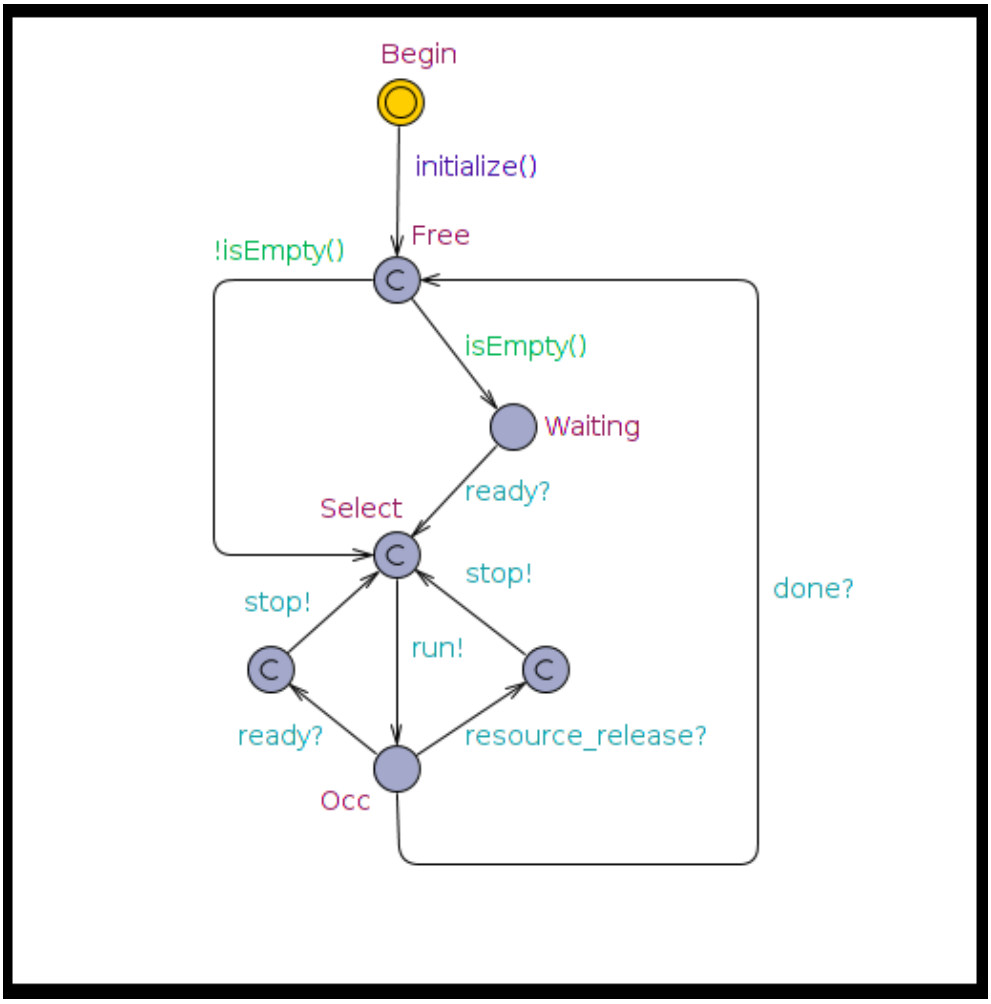


Figure 4.3: Scheduler automaton (deterministic) in UPPAAL

– If running task finishes execution the scheduler transitions to 'Free' state.

– When a resource is released the system ceiling may be updated so the scheduler preempts the current running task and transitions to 'Select' state.

– On arrival of a new task the scheduler preempts the current task and transitions to 'Select' state.

4.2.2 Non - deterministic Model:

Assumptions

- Uniprocessor system is used.
- Each task has a fixed priority, known before its execution.
- Tasks are modelled as periodic jobs.
- Each resource can be used by only one processor at a time.
- Scheduler has knowledge of which tasks will arrive beforehand, as well as the various resources used by them.
- Resource locking/unlocking event occurs randomly during the task's execution, scheduler has no prior knowledge of these events.

Model Description

The model consists of three automata namely Task, Scheduler and Resource. Every Task and Resource is represented by its own system whereas there is only one scheduler system.

The model requires the following inputs:

- Number of tasks
- Number of resources
- For each task:
  - Arrival time
  - Computation time
  - Priority
  - Resources used
- Initial value of system ceiling = -1 (i.e. System ceiling when all resources are free).

Two Priority Queues are used, one for tasks which are blocked and other for non-blocked tasks. Both queues are ordered using task priorities.

Every time a task arrives it is either added to blocked queue if it is blocked (i.e. its priority is less than system ceiling) or to the non-blocked queue if its not blocked.

Task Automaton

Each task requires finite amount of computation and has a fixed priority known beforehand by the scheduler. For each task with every integral increment of the program counter, following events randomly occur. When the task finishes execution, all resources locked by it are freed.

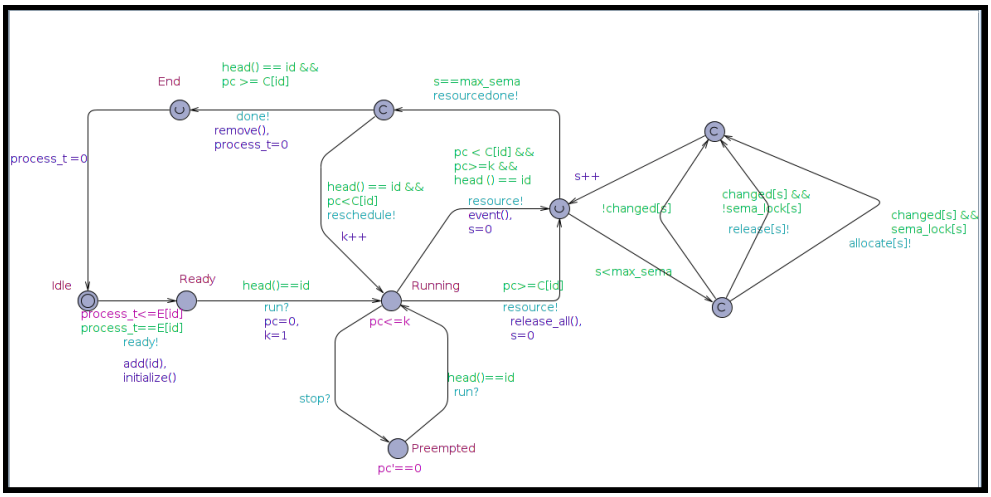


Figure 4.4: Task automaton (non - deterministic) in UPPAAL

- Locking of one or more resources used by the task.
- Unlocking of one or more resources already locked.

A task is modelled to have 6 states namely:

- Idle: Task is not ready to execute.
- Ready: Task is ready to execute i.e. it has arrived.
- Running: The scheduler has scheduled the task to run. During this state the program counter increases and after every integral increment the system transitions to 'Event' state. If the task is preempted the system transitions to 'Preempted' state.
- Event: If the task finishes execution, all resources it has locked are freed and the system transitions to 'End' state. In case the task hasn't finished execution, one of the following events randomly occur for every resource used by the task:

- Lock if not locked
- Unlock if locked
- Do nothing

Then the system transitions back to 'Running' state.

- Preempted: The task has been preempted. The program counter is frozen in this state. When the scheduler signals execution of this task the system transitions back to 'Running' state.
- End: Task has finished execution. From here, the system transitions to the 'Idle' state from where it executes again after its period.

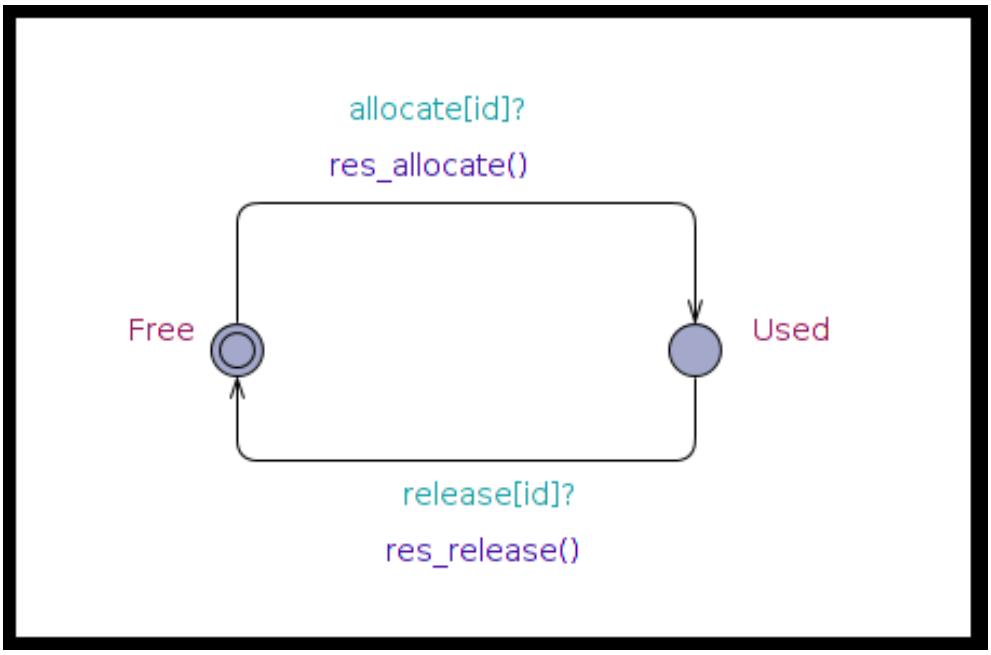


Figure 4.5: Resource Semaphore automaton (non - deterministic) in UPPAAL

Resource Semaphore Automaton

Resources are modelled as Binary semaphore. Each resource has the following states:

- Free: The resource is free, whenever a task locks a resource this transitions to 'Used' state while updating the system ceiling 'pi' and 'pi_stack' if necessary.

- Used: The resource is locked. When a task unlocks a resource this system transitions to a committed state, updates the system ceiling 'pi' and 'pi_stack' if necessary. If the system ceiling is updated the blocked-tasks and non-blocked-tasks queue are also updated, then the system transitions to 'Free' state.

Scheduler Automaton

The scheduler initialises all the resource ceilings then transitions to 'Free' state.

- Free: The system has no tasks scheduled to run currently If there are no ready to execute tasks it transitions to 'Waiting' state otherwise it transitions to 'Select' state.

- Waiting: The system is idle and is waiting for a task to arrive. When a task is ready to be executed it (i.e. Task.Idle to Task.Ready) synchronises with the scheduler via 'ready' channel and the Scheduler transitions from 'Waiting' to 'Select' state.

- Select: The non-blocked-tasks queue is non empty, the scheduler selects the highest priority task from the non-blocked-task queue for running. The system transitions to 'Occupied' state synchronising with the selected task via the 'run' channel.

- Occupied: The system is occupied and a task is running. Four transitions are possible here:

  - If runnning task finishes execution, the scheduler transitions to 'Free' state.
  - When a task transitions to the 'Event' state, the scheduler freezes by transitioning to a state where it waits for the resource locking/unlocking events to be randomly decided and then transitions back to 'Occupied' state.

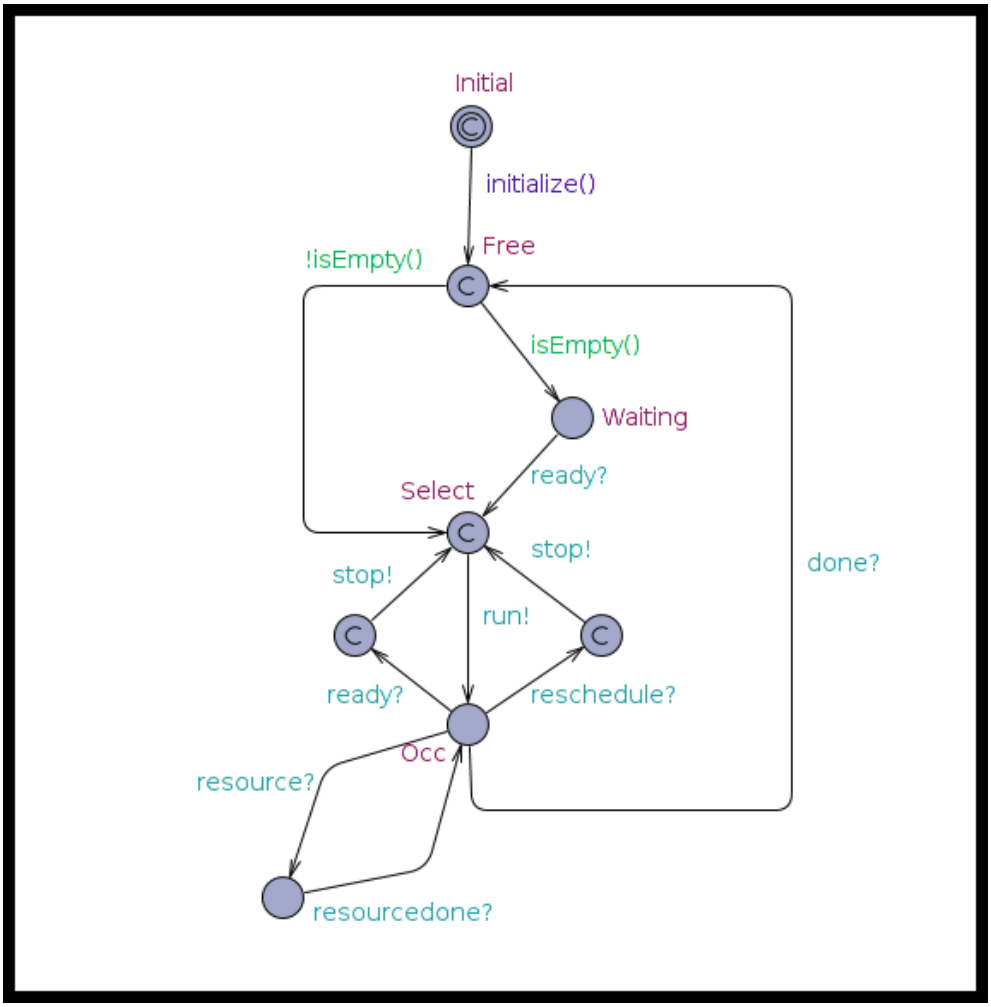


Figure 4.6: Scheduler automaton (non - deterministic) in UPPAAL

– After a task has finished resource locking or unlocking, the system ceiling may be updated. So the scheduler preempts the current task, and transitions to 'Select' state.

– On arrival of a new task the scheduler preempts the current task and transitions to 'Select' state.

4.3 Verification and Sanity Checks

For verifying properties in UPPAAL, we specify properties in CTL. The following properties were verified in SBPCP model:

- **Task finishes its event queue before terminating:**
  A[] forall (i: pid_t) (Task(i).End imply (Task(i).r == Task(i).t_len))
- **Deadlock condition:**
  A[] deadlock imply (forall (i: pid_t) Task(i).End)
- **Only one task is running at one instance:**
  A[] forall (i: pid_t)(forall (j: pid_t)(Task(i).Running && Task(j).Running imply i == j))
- **No job is ever blocked once its execution begins:**
  A[] forall (i: pid_t)(Task(i).Event1 && Task(i).ts[Task(i).r] == 0 imply Resource(Task(i).t_res[Task(i).r]).Free)
- **All tasks finish execution:**
  A⟨⟩ forall(i: pid_t) Task(i).End
- **Resources are free when no process is Running:**
  A[] Scheduler.Waiting imply forall (j: sid_t) Resource(j).Free

Chapter 5

Experimental Analysis and Results

The model for SBPC protocol in UPPAAL was compiled and run in a computer with specification as follows:

- **Operating System**: Arch Linux
- **OS Version**: 4.13.12
- **RAM**: 8GB DDR3 1.6 GHz
- **CPU**: Core i7 4710 HQ @ 2.5 GHz
- **Number of cores**: 4 / 8 (Physical / Virtual)

The results were recorded and are reported below:

5.1 Results for deterministic model

The model was run with 4 tasks and 3 resources. Each property specified was verified multiple times and the average of the results was taken and are recorded as follows:

Property	Time used (in s)			Memory usage peaks(in kB)	
	Verification	Kernel	Elapsed	Resident	Virtual
Deadlock freedom A[] deadlock imply (forall (i: pid_t) Task(i).End)	0	0.01	0.008	8832	43980
All tasks finish execution A⟨⟩ forall(i: pid_t) Task(i).End	0.01	0	0.007	8244	76200
Each task finishes its event queue before terminating A[] forall (i: pid_t) (Task(i).End imply (Task(i).r == Task(i).t_len))	0	0	0.008	7264	42420
All resources are freed when all processes finish execution A[] (forall(i: pid_t) Task(i).End) imply (forall (j: sid_t) Resource(j).Free)	0	0	0.009	6740	41840
Resources are free when no process is Running A[] Scheduler.Waiting imply forall (j: sid_t) Resource(j).Free	0.01	0	0.009	11412	45776
Only one task is running at a time A[] forall (i: pid_t)(forall (j: pid_t) Task(i).Running && Task(j).Running imply i==j)	0.01	0.01	0.01	7816	42972

Table 5.1: Complexity of verification properties of deterministic properties

5.2 Results for non - deterministic model

The model was run with 4 tasks and 3 resources. Each property specified was verified multiple times and the average of the results was taken and are recorded as follows:

Property	Time used (in s)			Memory usage peaks(in kB)	
	Verification	Kernel	Elapsed	Resident	Virtual
Deadlock freedom A[] deadlock imply (forall (i: pid_t) Task(i).End)	1.16	0.6	1.78	30000	58648
When all resources are free system ceiling equals -1 A[] (forall (i: sid_t) (Resource(i).Free)) imply pi == -1	0.25	0.01	0.245	26984	55708
A task is preempted only by a higher priority task A[] forall (i: pid_t)(forall (j: pid_t) Task(i).Running && Task(j).Preempted imply P[i] ⟩ P[j])	0.29	0	0.284	30804	59728
All resources used by a task are freed when it finishes its execution A[] forall (i: pid_t) (Task(i).End imply (forall(j: sid_t) (uses_sema[i][j] == 1 imply Resource(j).Free)))	0.8	0.68	1.496	29952	55576
Resources are free when no process is Running A[] Scheduler.Waiting imply forall (j: sid_t) Resource(j).Free	0.24	0	0.245	28264	56896
Only one task is running at a time A[] forall (i: pid_t)(forall (j: pid_t) Task(i).Running && Task(j).Running imply i==j)	0.28	0	0.287	29548	58216

Table 5.2: Complexity of verification properties of non - deterministic properties

Chapter 6

Future Work

Until now, SBPCP has been implemented in UPPAAL, a CTL model checker. Some properties can not be expressed in CTL but can be expressed in LTL. UPPAAL can only verify properties specified in CTL. SAL can verify both CTL and LTL properties and give counterexamples for properties specified in LTL.

The future work involves implementing SBPCP in SAL to verify those properties specified in LTL and to record complexity of verifying the properties.

Bibliography

[1] EM Clarke, O Grumberg, D Peled. *Model Checking.* MIT Press, 1999

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, Cambridge, Massachusetts

[3] Michael Fisher, Louise Dennis and Matt Webster. *Verifying Autonomous Systems.* Communications of the ACM, Vol. 56

[4] Rajeev Alur and David Dill. *Automata-theoretic Verification of Real-time systems.* 1995

[5] Jane W S Liu. *Typical Real-Time Applications.* 2000

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal.* http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf

[7] Kim G. Larsen, Paul Pettersson, and Wang Yi. *UPPAAL in a Nutshell.* http://www.it.uu.se/research/group/darts/papers/texts/lpw-sttt97.pdf