Helsinki University of Technology
Department of Electrical and Communications Engineering

Model checking a client-server system with a scalable level of concurrency

Topi Pohjolainen

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology

Espoo, May 6, 2005

Supervisor: Prof. Ilkka Niemelä
Instructor: D.Sc. (Tech.) Keijo Heljanko

Author: Topi Pohjolainen

Title: Model checking a client-server system with a scalable level of concurrency

Finnish title: Rinnakkaisen asiakas-palvelin järjestelmän verifiointi mallintarkastusta käyttäen

Date: 6. May 2005 **Pages:** vii + 42

Department: Department of Electrical and Communications Engineering

Professorship: Theoretical Computer Science **Code:** T-119

Supervisor: Prof. Ilkka Niemelä **Instructor:** D.Sc. (Tech.) Keijo Heljanko

Abstract:

During the last few decades software has become more common in an increasingly wide range of products. The tools and methodology regarding software development have improved but the improvements have rather addressed the quantity than the quality. While one is given the ability of producing more software in less time, the paradigms needed for validating the results have lacked behind. This thesis studies a field of formal verification called *model checking* providing a robust and exhaustive approach for software validation. The work examines a client-server system applying concurrency in its dynamic structure and aims to verify the absence of often encountered faults such as deadlocking and starvation in the design. The system of interest is modeled and verified with two state-of-the-art model checkers, Maria and Spin, to demonstrate in practice the advantage of having formal verification as part of the software development process. In addition, the performance of the tools is studied by gradually increasing the level of concurrency in the models and by examining the effect on the processing times needed for the checking. It is shown that both Maria and Spin apply well to the task in hand while having their unique strengths and weaknesses.

Keywords: Verification, Model checking, Linear Temporal Logic, Concurrent Programming, Mutual Exclusion, Starvation, Deadlocking, Weak and Strong Fairness

Tekijä: Topi Pohjolainen

Työn nimi: Rinnakkaisen asiakas-palvelin järjestelmän verifiointi mallintarkastusta käyttäen

English title: Model checking a client-server system with a scalable level of concurrency

Päivämäärä: 6. toukokuuta 2005 **Sivut:** vii + 42

Osasto: Sähkö- ja tietoliikennetekniikan osasto

Professuuri: Tietojenkäsittelyteoria **Koodi:** T-119

Valvoja: Prof. Ilkka Niemelä **Ohjaaja:** TkT Keijo Heljanko

Tiivistelmä:

Muutamien viimeisien vuosikymmenien aikana on ohjelmistoja hyödynnetty yhä useammassa tuotteessa. Ohjelmistokehitykseen käytetyt työkalut ja menetelmät ovat parantuneet, mutta kehitys on tapahtunut enemmän määrää kuin laatua silmälläpitäen. Vaikka ohjelmistoja voidaankin tuottaa yhä nopeammin, eivät menetelmät laadun tarkistamiseen ole parantuneet samassa määrin. Tämä työ käsittelee formaalin verifioinnin osa-aluetta nimeltään *mallintarkastus*, jolla ohjelmistojen oikeellisuus voidaan todentaa tyhjentävästi. Työssä tarkastellaan asiakas-palvelin-pohjaista järjestelmää, joka hyödyntää dynaamisessa rakenteessaan rinnakkaisuutta. Tarkoituksena on varmistaa, että yleiset rinnakkaisuuten liittyvät viat kuten lukkiuma ja nälkiintyminen, eivät esiinny järjestelmässä. Työssä tarkasteltu järjestelmä mallinnetaan ja verifioidaan käyttäen kahta eturivin mallintarkastinta, Mariaa ja Spiniä, pyrkien samalla tuomaan ilmi formaalin verifioinnin tarjoamia etuja ohjelmistokehityksessä. Edelleen työssä tarkastellaan käytettyjen työkalujen suorituskykyä tutkimalla mallin läpikäyntiin käytetyn ajan suhdetta mallin rinnakkaisuuteen. Käy ilmi, että sekä Maria että Spin soveltuvat annettuun tehtävään hyvin kuitenkin omaten yksilöllisiä eroja vahvuuksissaan ja heikkouksissaan.

Avainsanat: Verifiointi, mallintarkastus, lineaarinen aikalogiikka, rinnakkaisohjelmointi, keskinäinen poissulkevuus, nälkiintyminen, lukkiuma, heikko ja vahva reiluus

Contents

List of Figures

List of Tables

Terms and Abbreviations

LTL	Linear Temporal Logic
MARIA	The Modular Reachability Analyzer
SPIN	An LTL Model Checker
PROMELA	Process Meta Language, the Input Formalism for Spin
MUTEX	A Device for Implementing Mutual Exclusion
ANSI	American National Standards Institute
ANSI C	The C Programming Language Specified by the ANSI
C++	An Object Oriented High-level Programming Language
$\models$	Satisfaction relation
$\mathcal{X}$	Temporal logic operator: next time
$\mathcal{U}$	Temporal logic operator: until
$\Box$	Temporal logic operator: always
$\Diamond$	Temporal logic operator: eventually, exists
$\rightarrow, \leftarrow$	Predicate logic connectives: implications
$\wedge, \vee$	Predicate logic connectives: and, or
$\neg$	Negation in predicate logic
$S \subseteq T$	The set S is a subset of the set T
$S \times T$	The cartesian product of two sets S and T
2^S	The set of all subsets of set S
$T \rightarrow 2^S$	A function mapping the members of set T into 2^S
$\langle E_1, E_2, ..., E_n \rangle$	A tuple with n elements E_i

1 Introduction

Industrial level software development has been in steady increase for some time. Especially the introduction of the high level languages such as C++ [13] and Java [23] have greatly increased the applicability of software. While there has been a lot of progress in the basic tools such as the compilers, the capability of pinpointing programming errors has not improved in the same phase. In most cases the sole methodology for finding faults has been simple testing. In addition, design decisions are rarely validated until the completion of the implementation, if not even then. One has to remember that testing is an incomplete paradigm which at its best may give only vague guarantees about correctness. *Verification* [3] in turn is a complete method considering all the possible behaviors of a system where testing considers only a fraction of them.

Contrary to common belief, proper verification can be applied to various types of systems, both before and after implementation while the importance of the former cannot be emphasized enough. Most of the times implementation is time consuming activity which one should not engage in before one has a valid design to work with. In other words, one should not waste time implementing designs that are invalid to start with. Furthermore, using a proper formalism it is possible to validate designs in a fraction of time it would take to implement them.

This work studies verification in the domain of concurrent programming. The aim is to illustrate how properties related to concurrency may be expressed formally and how these properties may be verified automatically. This thesis introduces an abstract system design involving a scalable level of concurrency. By verifying the design, the methodology and tools are studied in practice.

The verification paradigm considered in this thesis is called *model checking* [4]. In principle, it is about formalisms enabling one to represent designs as *models* and checking the models for the properties of interest. There are various methods and tools to choose from, among which this thesis concentrates on two tools, namely *Maria* [1] and *Spin* [8, 9]. These are compared both with respect to the modeling power and to the efficiency of the model checking itself.

Special emphasis is placed on *the process scheduling* [7], on its properties such as *fairness* [11], and modeling techniques. Regarding the tools, the aim is to study their response to the level of concurrency in the model. Model checking is known to be a demanding computing task[1]. There are, however, several ways the tools may fight against this. The purpose of this thesis is to find out how the chosen tools manage in practice.

The work is organized as follows. First, verification is placed into the context of a development cycle. This is followed by the description of the design to be examined. The subsequent two sections discuss concurrent programming, first in general and then with respect to model checking. Before actually modeling the design, the underlying theory and the tools are studied in the scope of this work. Section 7 finally builds the models and formalizes the requirements of interest. These are immediately followed by the results and related discussion. It will be shown that while both Spin and Maria have their strengths and weaknesses, both are highly applicable to the problem in hand and that verification can be a valuable part of a development cycle.

[1] Model checking is known to be PSPACE-complete with respect to the input, i.e., the size and complexity of the model [5].

2 Verification

The development work of any kind of a system usually starts with a specification outlining the purpose and the requirements of the system. The actual amount of details varies case by case but in general the specification gives at least a rough overview what the system should be capable of once implemented. Assuming one is presented with a meaningful specification, the first task is to produce a plan, called *the design*, telling how the system described by the specification is going to be implemented. Regardless of the level of abstraction in the specification or the design, one eventually wants to know if the design is correct, i.e., if the design leads to an implementation that matches the specification. A common approach is to implement the system according to the design and try to answer the question in practice. But both, implementation and testing, are often time consuming tasks and ill side-effects of a faulty design might not be apparent even then.

It is argued that it is possible to *validate* designs formally without needing to implement them first [1, 4]. Furthermore, it is often possible to do validation in only a fragment of the time it would take to implement the system. The task of checking if a design matches its specification is referred to as *verification* and unlike testing, is *an exhaustive method* capable of telling for sure if given requirements apply in the design or not. Regardless of the specific paradigm applied, verification essentially provides

- highly expressive formalism for compact representation of designs,
- semantically well-defined and expressive way of stating requirements and
- enables one to detect defects in designs as early as possible.

2.1 An automated paradigm, model checking

While in theory verification can be done as a mathematical proof applying techniques such as induction, it is often impractical due to the level of skill required not to mention the amount of work needed to be done by hand. Therefore one often opts for another paradigm called *model checking* [3, 4] which is *an automated* approach. In model checking the core of the verification task is done by a computer as an *exhaustive search*.

In principle one represents the design and its specification formally and applies a tool for checking the former against the latter. Compared to implementation, instead of representing the design in *the implementation language*, one translates the design into a *model* using a more expressive *modeling language*. Whereas a compiler of an implementation language expresses the design for a computing platform to execute, *a model checker* produces a representation of all the possible executions of the design.

Sounding as good as it does, model checking, however, can in practice be very demanding with respect to the amount of random access memory and processor time. An increase in the complexity of the models may greatly effect the time and memory needed for checking them. While this may not always be the case, one should nevertheless be aware of it while building complex models. The problem is referred to as the *state-space explosion problem* [5] and is examined in more detail in Section 9.

2.2 Model checking in the development cycle

Embedding model checking into the conventional development cycle is ideally done as depicted in Fig. 1. There the model checking simply becomes an additional step between

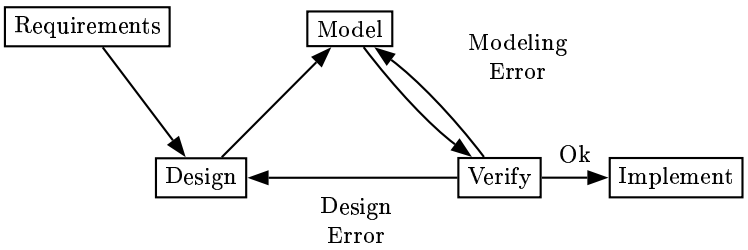


Figure 1: Verification and refinement procedure.

the design and the implementation phases. The idea is to start with high-level abstractions of both the design and the model and to refine them gradually. In terms of the diagram, the three phases *Design*, *Model* and *Verify*, are to be iterated until the desired level of detail is achieved.

While the errors in the design need to be corrected both in the design and in the model, the errors caused by inconsistencies between the model and the design require the model to be adjusted until it correctly reflects the design. One repeats the iteration until the design matches all the given requirements and only then begins the implementation phase.

3 The design to be verified

This thesis aims to elaborate the phases *Requirements*, *Design*, *Model* and *Verify* depicted in Fig. 1 by performing each of them in practice. The focus is on verification and especially on model checking. While this section discusses the requirements and the design, the latter two cannot be studied before the proper background is in place. The aim, however, is to apply model checking on a design containing concurrency and examine how two top of the line model checkers, *Maria* [1] and *Spin* [8, 9], cope with the state-space explosion in practice.

Irrespective of the specific development methods and tools being used, a common approach is to work with an abstraction of the system. Therefore, instead of considering the entire specification with all its details, only those parts that have relevance to the behavior of interest are usually examined. This is indeed the case also in this work. While the high level of abstraction is primarily needed for model checking efficiency, it should also make it easier to adopt this work elsewhere.

3.1 The specification

The system of interest can be thought as a periodical transmitter of information. The information is partitioned into items which are transmitted in parallel using one or more *Asynchronous Layered Coding*, ALC [19] channels, i.e., *Internet Protocol* IP [20, 21] streams. The order of the items is not considered here but the items are thought to be sent repeatedly over time. From the analysis point of view the items are thought to be grouped into non-overlapping sets, each set regarded as self-contained unit independent of the others. A set is associated with an output time window and an output frequency. The output time window specifies the interval within which the system should output the items in the set. The frequency in turn specifies the time interval between the output operations of any two subsequent items belonging to the same set. And each of the sets may be transmitted using one or more channels. From here on the channels are simply referred to as *outputs*.

The system has to provide an interface for adding and removing sets, modifying their parameters, the time window and the frequency, as well as the contents of the sets.

The accuracy concerning the output intervals and frequencies is relaxed to match the capabilities of the surrounding environment. At the abstraction level of this thesis, all operations are only required to take finitely long time forcing the delays in periodic operations to be finite also. All *real-time scheduling* [7, Pages 394-406] issues are out of the scope of this work as the requirements concerning them are relaxed to match the invidual transmission environments.

3.2 The static structure of the design

The given specification does little to restrict the *structure* of the design. Specifically, the specification just considers what the behavior of the system should look like from the outside while the internal breakdown of the system is left totally open. From the number of design possibilities, this work studies a straightforward approach utilizing concurrent programming [7, Pages 56-60].

The system is partitioned into separate processes based on the sets of items. Each set is given its own process responsible for maintaining and outputting the items of the set.

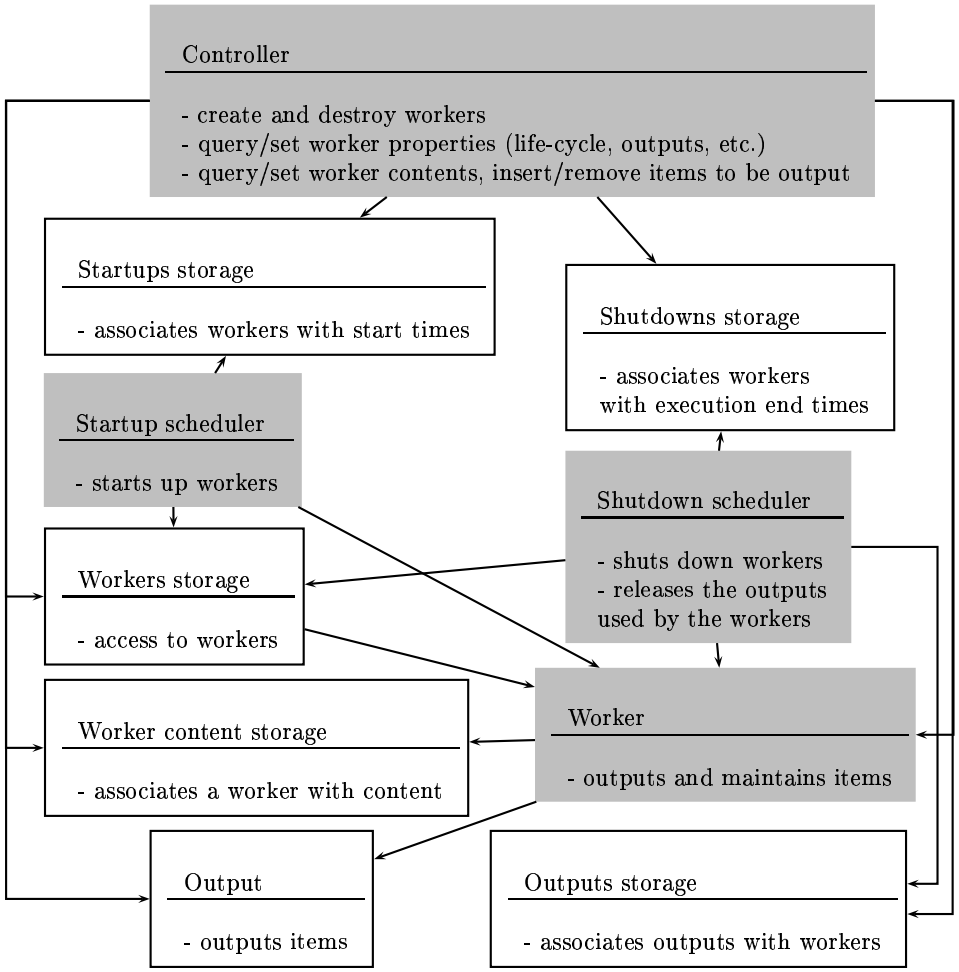


Figure 2: Entities of the system.
Light gray boxes depict processes while white boxes depict pure data entities.

These processes, called *workers*, are in turn controlled by three utility processes, called the startup scheduler, the shutdown scheduler and the controller. While the controller takes care of additions and removals of sets as well as their modifications, the two schedulers are responsible of starting up and shutting down worker processes according to their output time windows. All entities and their relationships are depicted in Fig. 2.

Each worker is associated with a *content storage* for holding the items the worker should output. The workers are not aware of the content storages of the other workers.

The startup scheduler in turn is associated with a data structure called *startups storage* which tells the scheduler when to kick-off the executions of the workers. Similarly, the shutdown scheduler is associated with a storage called *shutdowns* which tells the scheduler when to shutdown the worker processes. The starts and ends of the worker time windows are therefore maintained separately.

The controller is thought to accept commands and queries from the clients of the system. Via the controller, clients of the system can create and destroy workers as well as modify their contents (the sets of items). The controller keeps a data structure called the *workers storage* providing access to the workers.

```
0. while (¬terminate)
1.     Lock "my content storage s";
2.     Lock "an output o";
3.     Output an item ∈ s;
4.     Release o;
5.     Release s;
6. endwhile
```

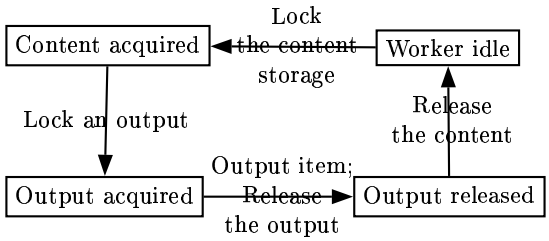


Figure 3: The body of the worker process and the simplified state machine.

The controller is also responsible for providing workers with outputs. One worker may be associated with more than one output and one output may be associated with more than one worker also. This association is done with the means of a storage called *outputs*.

Workers are first created as pure data entities into the workers storage by the controller. Upon creation each worker is associated with one or more outputs and a life-cycle. The controller updates corresponding storages, the outputs, the startups, the shutdowns and the workers accordingly. The startup and shutdown schedulers are then relied on to kick-off and to shutdown executions of workers.

Workers may be associated with contents before and during their execution life-cycle. The controller does this when requested by the client(s) of the system.

3.3 The dynamic structure of the design

The concurrency in the design introduces special needs for the data structures of the system. The data entities are *guarded* restricting several processes from using them concurrently but instead only one process at a time. Without such a construct two processes oblivious of each other could modify the same memory area simultaneously causing the behavior of the system being unpredictable. Section 4 discusses the need for the guarding and the precise mechanism in more detail. For now, processes desiring access to a storage are just considered to be *blocked*, i.e., suspended from executing, whenever the storage is occupied. A storage becomes occupied when it is locked and unoccupied upon release.

As the name implies, the workers are the ones performing the core processing of the system, the outputting. The process body listed in Fig. 3 shows that a worker is simply a loop that repeatedly examines its content storage and outputs the items within. The boolean flag *terminate* controls if the worker should keep on going or cease executing. These flags are set by the controller and the shutdown scheduler whenever a worker process is needed to be terminated. The precise mechanism is discussed in the context of the controller and the shutdown scheduler in turn.

The state machine associated with the worker process listing provides simplified view of the body. The precise mechanism for termination is ignored and the state machine therefore depicts how the workers act whenever they are alive. The reasoning behind the abstraction becomes apparent when examining the process body and the corresponding state machine of the controller.

The controller is designed as listed in Fig. 4. It serves as the control interface for the system by processing the requests from the clients. The controller supplies the information contained in these requests into the various storages of the system to be "consumed" by the other processes. The controller process has the life-cycle of the entire system. In other words the controller process is created in the startup of the system and destroyed along the shutdown of the system.

```
0. while (true)
1.    Read client request r;
2.    if (r = "query or modify content of a worker w") ∨ (r = "add worker w")
3.       Lock "the workers" storage;
4.       Lock "the content storage" s_w of w;
5.       Query or modify s_w;
6.       Release s_w;
7.       Release "the workers";
8.    fi
9.    if (r = "add or remove worker w")
10.      Lock "the startups" storage;
11.      Add or remove start time of w in/from "startups";
12.      Release "the startups";
13.      Lock "the shutdowns" storage;
14.      Add or remove end time of w in/from "shutdowns";
15.      Release "the shutdowns";
16.      Lock "the workers" storage;
17.      if (r = "remove w") ∧ w is running
18.         Set the termination flag of w to true;
19.         Wait until w terminated;
20.      fi
21.      Lock "the outputs" storage;
22.      if (r = "remove w")
23.         Destroy output(s) used solely by w;
24.      else
25.         Create the output(s) needed by w that do not exist yet;
26.      fi
27.      Release "the outputs";
28.      if (r = "remove w")
29.         Remove w in "the workers";
30.      fi
31.      Release "the workers";
32.   fi
33.endwhile
```

Figure 4: The body of the controller process.

In the initial model constructed in Section 7, the level of detail found in Fig. 4 is reduced. The model concentrates on the concurrency of the design and therefore ignores all the local processing.

Consider the simplified state machine in Fig. 5. The state machine depicts only the locking and releasing of the shared storages. The local processing detailed in Fig. 4 is thought to happen between these actions and taking an arbitrary but finite amount of time.

The worker process termination depicted on lines 18 and 19 is taken into account simply by making the controller to wait until the content storage of the corresponding worker becomes available. Recall that a worker terminates only from the idle state and that the last thing a worker performs before returning to its idle state is to release its content storage.

Observe also that the lines from 22 to 26 are ignored. On line 23 the controller examines outputs that are not used by any other process, and on line 25 the controller just creates new outputs.

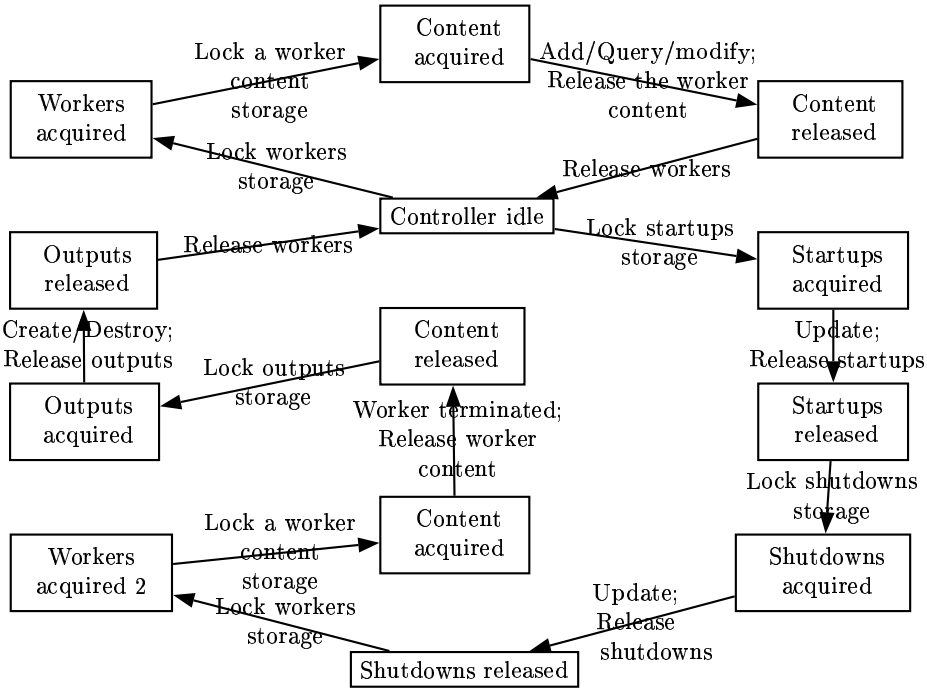


Figure 5: Simplified state machine of the controller process.

The startup scheduler computes in "looping" fashion also. It repeatedly checks the startup storage to determine if one or more of the worker processes need to be started. It uses the workers storage for accessing the workers themselves. The process body and the corresponding state machine are depicted in Fig. 6.

Like in the case of the workers and the controller, the state machine of the startup scheduler ignores the details of local processing. The design relies on the operating system to startup processes and to provide an interface for the startup scheduler to wait for the completion of the operation. In normal conditions the startup operation does not fail and is computed quickly. The failure cases in turn are due to anomalies in the surrounding environment such as excess load. In the chosen abstraction level success and failure look

```
0. while (true)
1.     Lock "the startups storage S";
2.     Lock "the workers storage";
3.     for every (time, worker) ∈ S
4.         if time ≤ now
5.             Kick off worker;
6.             Wait until running;
7.         fi
8.     end
9.     Release "the workers";
10.    Release "the startups";
11. endwhile
```

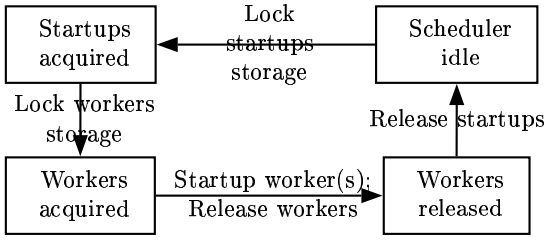
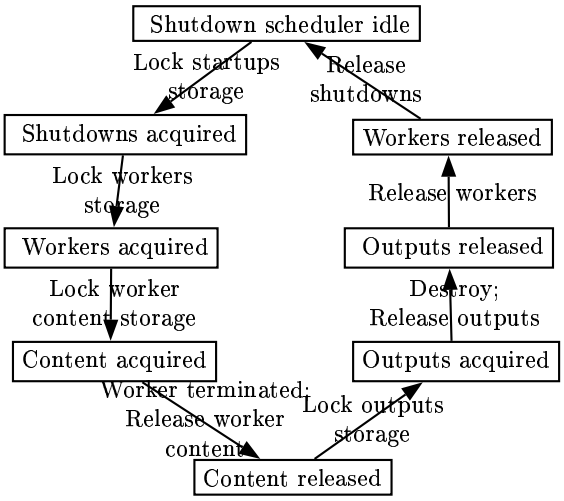


Figure 6: The process body and the state machine of the startup scheduler. The current time acquired from the operating system is represented by the variable *now*.




```
0.while (true)
1.   Lock "the shutdowns storage S";
2.   Lock "the workers storage";
3.    for every (time, worker) ∈ S
4.       if time ≤ now
5.           Set the termination flag of
                worker to true;
6.           Wait until worker terminated;
7.           Lock "the outputs storage";
8.           Destroy output(s) used solely
                by worker;
9.       fi
10.    end
11.  Release "the workers";
12.  Release "the shutdowns";
13.endwhile
```

Figure 7: The process body and the state machine of the shutdown scheduler. Similarly to the startup scheduler, the current time is represented by the variable *now*.

alike. The control simply flows through the state machine and result is ignored in the idle state. The startup operation is therefore simply assumed always to be computed in finite amount of time regardless of the result.

The shutdown scheduler examines its *shutdowns* storage in a similar fashion to the startup scheduler checking if one or more of the workers need to be terminated. The shutdown operation mimics the one defined for the controller and results in an equivalent abstraction in the state machine of the scheduler. The process body and the state machine are depicted in Fig. 7.

3.4 Properties to be verified

Describing the properties to be checked can be seen as a task interpreting the requirements stated in the specification to the context of the design. Furthermore, the mechanisms and algorithms applied in the design itself usually impose additional properties to be added. Nevertheless, regardless of the source the properties need to be described in such a manner that makes the actual verification feasible to perform.

Like the design phase, verification can also be performed using top-down approach. That is, starting with an abstraction of the requirements and refining them gradually. The verification phase performed in this thesis can be thought as the first round dealing with the most abstract level of the given requirements. The aim is to verify

1. the data integrity in the storages,
2. the finiteness of operations performed by the processes and
3. the continuity of the processes.

The first is due to the need imposed by the concurrency of the design. The second in turn is directly stated by the specification itself. The third is again imposed by the design

as the design expects the processes to compute *infinitely* unless terminated explicitly by the logic of the design.

All the properties fall into the domain of concurrent computing where one deals with

- *process synchronization* [3, 4],
- runtime defects caused by synchronization called *deadlocking* and *starvation* [7, 3, 4],
- *process scheduling* [7] and
- *fairness* [11] related to starvation.

These issues are studied in turn and only then the properties can be stated formally. In addition, instead of stating the detailed properties in general they are described directly in the contexts of the *models* of the design in Section 7.

4 Synchronizing computations

Concurrency in programming arises whenever a computing task is decided to be performed using several simultaneous processes[2]. Furthermore, processes executing in parallel and performing parts of a common task rarely execute entirely oblivious of one and other but rather tend to communicate with each other. This in turn raises the need for representing *synchronization* [3, 4] between processes.

Where synchronization is essentially about restricting the executions of concurrent processes with respect to one and other, the process communication can roughly be thought as *the semantic* level of synchronization giving *meaning* for the restrictions. The precise mechanism is often categorized into *message passing* [7, 3] and constructs based on *shared variables* [3, 4]. However, in practice these two often overlap and specific implementations could be, at least in theory, realized using either of them. This section considers a widely used technique called *mutual exclusion* [7] that can be thought as an application of either.

4.1 Shared resources, need for synchronization

A common situation requiring synchronization is one where processes *share* a common resource. The need for sharing in turn could be due to *limited* amount of resources in the environment or a specific arrangement *structuring* the concurrent execution of a computing task. Examples of the former are the physical devices of the computing platform, the *random access memory* and the processor time itself. A classic example of the latter is a database that handles several parallel requests using a separate process for each request, each process in turn targeting its operations on the shared database storage. Where the sharing due to limitations mostly falls to the domain of operating systems, the structuring by sharing is in turn applied by the designers of concurrent systems. Nevertheless, the basic need is the same, controlling *the access* to shared resources.

Consider a write operation targeting a shared chunk of memory. In case the operation spans over several instructions, it is possible for this operation to be suspended while some other process is given a chance to execute in turn. Assume that the chunk represents a storage and that the executed instructions deleted some entries in the storage, but where the remainder of the operation is not allowed to delete corresponding references. Consequently, there might be another process accessing the same storage and the references to the just deleted and therefore non-existing entries could result to undefined behavior. If one had means of preventing any process from accessing a storage while the storage is been updated, one could prevent such situations from happening.

4.2 Critical sections and mutual exclusion

Sequences of instructions accessing a shared resource are often referred to as *critical sections* [7, 4], emphasizing that only one process should perform such sequences at a time. A common mechanism forcing this behavior is called *mutual exclusion* [7, 4] which can be thought as an *agreement* between two or more processes not to enter their critical sections simultaneously. In practice, the specifics of the mechanism vary. This section defines a device called *mutex* for implementing the semantics used in this thesis.

[2]In the context of a single program, one often refers to concurrently executing entities as threads [7] rather than as processes. For the sake of simplicity, this thesis does not make this distinction.

Definition 1 *Given a set P of processes, a mutex Mx is represented with a state variable Mx_s over the values $\{reserved, free\}$ and a set $Mx_{sus} \subseteq P$ of suspended processes. The value of Mx_s and the contents of the set Mx_{sus} are altered by the processes using the mutex with two atomic operations, lock and release. While the operation lock depends on the value of Mx_s, the operation release depends on the contents of Mx_{sus} as follows:*

- *if $Mx_s = reserved$, the operation lock suspends the calling process and places the process into Mx_{sus},*
- *if $Mx_s = free$, the operation lock assigns $Mx_s := reserved$ and lets the calling process to continue its execution,*
- *if $Mx_{sus} = \{\}$, the operation release assigns $Mx_s := free$ and*
- *if $Mx_{sus} \neq \{\}$, the operation release selects one process $p \in Mx_{sus}$, removes p from Mx_{sus} and enables p to continue its execution.*

Assuming a process P performs the lock operation on a mutex Mx such that $Mx_s = free$, subsequently resulting the Mx_s to be assigned *reserved*, P effectively suspends the execution of any other process performing the lock operation on Mx. Once Mx is released by P, one of the suspended processes is resumed to execute and allowed to acquire the mutex in turn. The suspension of processes, i.e., blocking, and the releasing of the processes is usually handled by the operating system and is often part of the process scheduling studied in the next section.

Mutual exclusion on a shared resource can now be forced by associating a mutex Mx_{res} with the resource. Each entry to a critical section representing access to the resource is preceded by locking Mx_{res} and each exit in turn is followed by releasing Mx_{res}. The releasing is vital for allowing other processes to access their critical sections in some point also.

4.3 Defects caused by mutual exclusion

While mutual exclusion can be a powerful mechanism in concurrent designs, it can also be a source for multitude of runtime defects. The types of defects are often categorized into *deadlocking* and *starvation* [3, 4, 7]. Both are of special interest as the aim of the thesis is to show the absence of both in the design of interest.

Deadlocking

For deadlocking due to resource allocation to occur, there has to be at least two processes and at least two shared resource used by the processes. In addition, the access to each of the shared resources has to be restricted in such away that all of the processes cannot access the resource the same time. Consider the basic case where there are two processes competing for two shared resources such that the access to both of the resources is restricted to one process at a time. Deadlocking happens when both of the processes are holding one of the resources while the resource is needed by the other process. In order either of them to proceed, the resource held by the other process needs to be released. But as neither of them are able to release the resource they are holding before they have access to the other resource, the executions of both them are suspended infinitely.

Starvation

Unlike deadlocking, starvation usually manifests itself by only one of the processes of the system being *stuck*. The process is not, however, suspended explicitly, but rather implicitly by the other processes. Consider three processes P_1, P_2 and P_3 competing for an access into a mutually excluded resource. Now consider a computation where P_1 and P_2 take turns accessing the resource infinitely but where P_3 is not given a turn at all. One says that P_3 is *starved*.

It should be noted that while deadlocking may be detected in a system in a relatively short time, starvation often requires long runs and special conditions to occur. This makes it difficult, if impossible, to spot without exhaustive analysis. Thus in addition to detecting deadlocking in the early phases of development, formal verification can be regarded as the only tool for really finding the defects causing starvation.

4.4 Process scheduling and fairness

Starvation relates closely to the process selection procedure performed when a mutex is used. One process at a time is given a turn to acquire the mutex but nothing is said if some of the process are favored more than the others. For instance, if some of the processes are always neglected, they are bound to starve. The problem could be solved by imposing special mechanisms to the design itself. But the operating system scheduler [7, Pages 345-414] responsible for the selection can also be argued to be responsible for treating processes *fairly*. The design of interest relies on the latter and for the purposes of the verification, the notion of fair selection needs to be defined more precisely. While this section gives formal basis for representing the selection procedure, *fairness* [11] itself is discussed in Section 5.

The dilemma concerning the selection in the context of a mutex can be seen just as a part of the processor time distribution to the processes. Whenever the process scheduler is about to select the next process to be executed, the scheduler considers only those processes that are not blocked, i.e., not in the set of suspended processes of any mutex. A process that is not blocked is said to being *enabled* to execute. Blocked processes in turn become enabled when the corresponding mutex is freed and the selection procedure is performed again. The processes remaining in the suspension set become disabled while the one chosen to lock the mutex remains enabled until it is possibly blocked by some other mutex.

The aim is to express fairness constraints on the process scheduling stating that none of the processes should not be ignored infinitely. Using the notions of a process being enabled and selected, this could informally be re-stated by saying that if a process is enabled often enough, it must eventually be also selected.

In practice, there are as many approaches to scheduling as there are computing platforms and operating systems. And in general, giving the possible priorities of the processes, the goal of treating the processes as fairly as possible, i.e., to distribute the resources of the system as evenly among the processes as possible, is next to impossible to achieve and systems just aim to approximate such behavior. Nevertheless, assuming such behavior can serve as a starting point for studying starvation in designs.

5 Representing computations

Verification in the context of programming is essentially about observing how a program behaves. Specifically, one considers every possible way the program may be executed. Thus, there is a need for representing both a single execution of the program as well as the set of all possible executions. Additionally one may want to divide the set of all executions into subsets sharing some properties of interest. A common approach to examining a single execution is to consider it in terms of *a computation path* [11, page 10] while the set of all executions then becomes a set of computation paths.

In principle, a computation path is a sequence of states or configurations of the program. This makes it is possible to represent a property of an execution as a sub-sequence or a pattern in the sequence, and further divide the set of all computation paths into subsets with respect to the sub-sequence or the pattern.

It should be pointed out that while in theory it may be possible to construct computation paths of programs by hand, in practice this is very rarely the case due to the length and amount of the paths. Not to mention how tedious it would be to divide them into meaningful subsets. This thesis aims to elaborate how computation paths can be generated automatically and how their analysis can be automated also.

Definition 2 *A state of a program is a fixed valuation of the programs variables, the program counter, etc. From here on, a state is referred to as a configuration to make it clearer in other contexts.*

Definition 3 *An atomic instruction of a program is an instruction that cannot be suspended during its execution. In other words, an atomic instruction is always executed in one step.*

The concept of atomic instruction is needed in the context of multiprogramming environments where executions of programs can be *pre-empted* [7], i.e., suspended without a request from the program itself. The pre-emption can only occur before and after atomic instructions giving the granularity for the concurrency and therefore for the representations of the computations also.

Definition 4 *Given a set C of configurations a computation path π is defined to be a sequence $\pi = c_1c_2c_3...$ where $c_i \in C$ such that any two subsequent configurations c_j and c_{j+1} exist in the sequence if and only if there exists an atomic instruction that produces c_{j+1} from c_j. In this work all computation paths are additionally defined to be of infinite length. By writing π_j one refers to the configuration c_j at the index j and a suffix of π starting at the index j is in turn denoted by $\pi^j = \pi_j\pi_{j+1}\pi_{j+2}....$*

It should be noted that some instructions may not alter the configuration of the program and therefore two subsequent configurations may remain the same. Whether one needs to represent such instructions depends on the context. Sequences of instructions in turn may result in an already encountered configuration. Therefore, a computation path may contain a single configuration multiple times.

5.1 Computation trees and branching

Sometimes it is useful to visualize sets of computation paths at least partially in order to study some properties of the paths. One way to visualize several computations is to

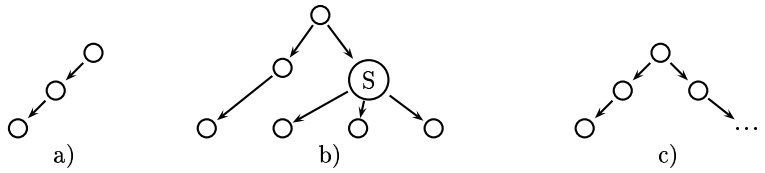


Figure 8: Examples of computation trees.

depict the *prefixes* of the corresponding computation paths using *a computation tree* [11, page 10]. In addition to sharing common prefixes of computation paths, computation trees can be used to visualize where the computation paths branch.

Each node of a computation tree represents a configuration of a program while an arc between two configurations represents an atomic instruction leading from one configuration to another. A computation path in a computation tree becomes simply a path starting from the root node of the tree, the root representing the initial configuration of the program.

Some examples of computation trees are depicted in Fig. 8. Tree *a)* is an example of a program with one execution path. Tree *b)* in turn has four paths. The node denoted *S* branches the incoming path into three and thus represents a selection in the program. The outcome of the selection has three possibilities and it may depend on various things such as the history of the program, the current state of the surrounding environment and possibly on some input. Nevertheless, depicted as a tree, all the three choices seem equally acceptable as one does not have any notion of *how* the selection is actually performed. For instance, when the program is run, the left-most path may in practice be taken with probability close to one making the two others virtually impossible to be chosen. But this certainly does not mean that the other two choices do not exist.

Trees *a)* and *b)* are also examples of always terminating programs. That is, all the paths in their computation trees are of finite length. Tree *c)* in turn has one path of infinite length. The computation along that path never terminates meaning that the execution of the corresponding program cannot be guaranteed to terminate either. It should be noted that a computation tree may have infinitely many paths of both finite and infinite length. For instance, an infinitely often repeated selection with several choices produces infinitely many paths.

5.2 Non-determinism and guarded choices

Similarly to the computation trees, one often wants to ignore the details of a selection procedure and simply just concentrate on the fact that the selection has several outcomes. Such a representation of a selection is called *non-deterministic*. The program listed in Fig. 9 illustrates how using a *guarded command language* [14, 11] non-determinism may be expressed using programming language like notation. The example also shows how the number of choices in the selection may be controlled over time using a technique called *guarding*.

The program is effectively an infinite loop that makes a selection over and over again. The first four choices are conditional, the last one being unconditional. The guards restrict which of the choices may be taken meaning that the conditional choices may be considered only when their guards are enabled. This results a computation tree depicted in Fig. 10. Note that every non-terminating node has three of the five choices enabled.

```
x := 0; y := 0
do
    :: (x == 0) →   y := 1; x := 1
    :: (x == 0) →   y := 2; x := 1
    :: (x == 1) →   y := 3; x := 0
    :: (x == 1) →   y := 4; x := 0
    ::              y := 5; break
od
```

Figure 9: A non-deterministic selection.
Lines beginning with :: represent choices followed by guards representing conditions for the choices. The last choice with empty guard represents a choice that can be taken unconditionally.

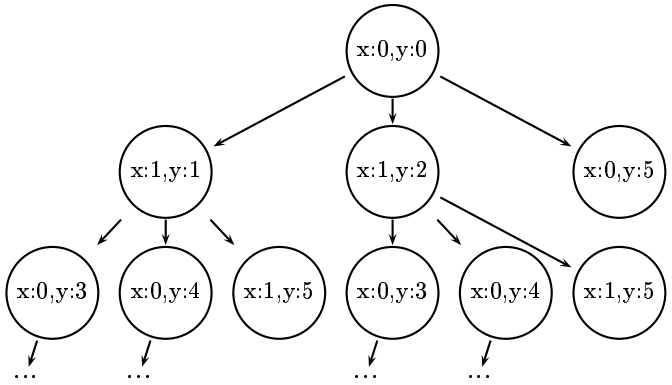


Figure 10: Computation tree representing non-deterministic guarded selection.

The computation tree in Fig. 10 also serves as an example on how a program may have *infinitely* many computation paths. The program has non-terminating executions which correspond to infinitely long paths in the computation tree. Along each of these paths there are in turn infinitely many nodes where the execution is branched over and over again yielding infinitely many computation paths.

5.3 Fairness in selections

An interesting property related to non-deterministic selections and guarded choices is *fairness* [11] observing *how* the choices are taken when the selection is repeated *infinitely* many times. This property is needed later on when *process scheduling* is studied as a non-deterministic selection. In general, fairness is a wide topic and an in-depth discussion can be found in the book by Francez [11] while this section just builds the required background for the rest of this study.

Before going into details, it should be pointed out that fairness is meaningfully defined only for infinite computations.

Example 1 *Consider a program that repeats a non-deterministic selection S infinitely many times. Let computation path π represent an arbitrary, but infinite, execution of the program. In case every choice of S that has its guard enabled continuously is taken infinitely often along π, one says that π is weakly fair. If in turn every choice of S that*

has its guard enabled infinitely often is taken infinitely often along π, one says that π is strongly fair.

The two notions of fairness give meaningful way of dividing computations into subsets based on how choices are taken. They are later on used as *filters* to the computations of *the process scheduler* restricting the way processes are selected over time.

5.4 Finite representations, Kripke structures

Computation trees serve well for elaborating things such as branching and prefixing of computation paths. They are, however, often structures of infinite size due to possibly infinite number of paths and possibly infinite lengths of the paths. This infiniteness makes the computation trees impractical for algorithmic uses and one needs to have some other means for representing infinite computations. For instance, the tools automating the analysis of programs require finite structures.

In principle, in order to keep the representation finite one needs to avoid duplicating configurations infinitely many times. One widely used construct forcing this is *a Kripke structure* [4, Page 14] which needs to include each configuration of a program only once. If the total number of configurations is finite, the size of the structure can be kept finite also.

Where a computation tree uses nodes for representing configurations, a Kripke structure refers to the configurations using *states*. But unlike the computation tree, the Kripke structure does not depict each path explicitly but rather folds them "on top of each other".

Definition 5 *Given a set of atomic propositions AP, a Kripke structure is defined as a four tuple $\langle S, S_0, R, L\rangle$ where*

- *S is a finite set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is a successor relation such that each state $s \in S$ must have at least one successor state (which can be the state itself),*
- *$L : S \to 2^{AP}$ is a function associating each state $s \in S$ with a set $L(s) \subseteq AP$ of atomic propositions that hold in the state s.*

The set of atomic propositions is used to represent configurations. A straightforward approach would be to give each valuation of each variable its own proposition. For instance, the set of propositions for the program listed in Fig. 9 becomes $AP_{ex} = \{X_0, X_1, Y_1, Y_2, Y_3, Y_4, Y_5\}$, where $X_0 : (x = 0)$, $X_1 : (x = 1)$ and so on. The successor relation R in turn is constructed by examining which configurations can follow each other in any of the possible computations of the program.

Definition 6 *A Kripke structure corresponding to a program P is defined as a tuple $\langle S_P, S_{P0}, R_P, L_P\rangle$ where*

- *S_P is the set of all possible configurations of P,*
- *$S_{P0} \subseteq S_P$ is the set of initial configurations,*

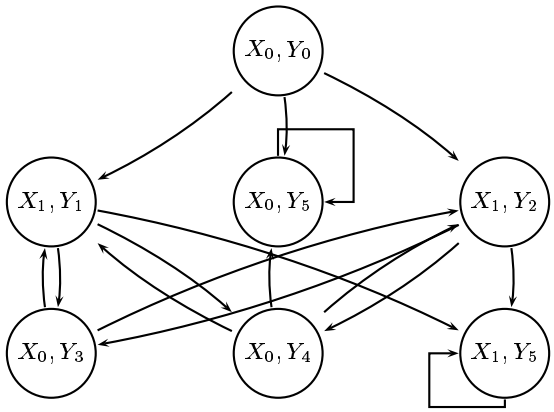


Figure 11: A Kripke structure for program in Fig. 9.

- *R_P is the successor relation such that $(c1, c2) \in R_P$ if and only if along some computation path π of P there exists an index i such that $\pi_i = c_1$ and $\pi_{i+1} = c_2$ and*
- *L_P is a labeling associating each $c \in S_P$ with a subset of A_P of atomic propositions.*

A computation path in a Kripke structure $\langle S_P, S_{P0}, R_P, L_P \rangle$ then becomes an *infinite* sequence $c_1 c_2 c_3 ...$ of configurations such that $c_1 \in S_{P0}$ and that for every two subsequent configurations c_j and c_{j+1} there exists corresponding relation $(c_j, c_{j+1}) \in R_P$. The infiniteness is due to the fact that every state has at least one successor. Finite computations are represented simply by defining each of the states representing final configurations to have the state itself as its successor.

A Kripke structure representing the program in Fig. 9 is depicted in Fig. 11. It should be noted that only those configurations are depicted that can be reached by one or more computations of the program.

6 Modeling designs and their requirements

The verification technique, model checking, studied in this thesis is based on the ability to represent the computations of a given program. This means essentially the capability of generating the set of all possible configurations of the program and given a configuration to deduce the next possible configuration(s). All the computations paths of the program can then be produced by starting from the initial configuration and recursively exploring the configurations that can immediately follow.

In model checking the configurations and their successors are deduced by *a model checker* from *a model* representing the design of the program. The model is essentially a representation of the variables of the program and the control logic expressed in *the modeling language* of the model checker. Whereas the implementation language of a program represents the program for a compiler, the modeling language represents the program to the model checker. This section studies the languages, *Petri nets* [2] and *Promela* [9], accepted by the model checkers *Maria* [1] and *Spin* [8, 9], respectively. The Promela language resembles closely conventional programming languages while a Petri net can be seen as *transition system* dealing with transitions between configurations already in the syntactic level. Nevertheless at least in theory, both can be used to represent any kind of programs, they are especially practical for representing concurrency due to their built-in support for synchronization between processes.

In addition to being capable of generating the computations of a program, one also needs to have corresponding means for representing the requirements to be checked. One widely used formalism, called *the linear temporal logic*, LTL [4, Pages 27-32], is of special interest. First, it can easily be utilized for expressing claims with preconditions and claims about events happening in sequence and/or periodically. And second, LTL is well-defined on Kripke structures making it applicable in variety of model checkers including Maria and Spin.

The modeling languages, Petri nets and Promela, are studied first followed by a discussion of LTL in the scope of this work.

6.1 Petri nets

Transition systems provide an intuitive way of representing processes. Instead of describing the control logic of a process in terms of instructions, one represents directly the moves, called *transitions*, from configurations to others. A transition system therefore simply describes the behavior of a program as discrete changes in the status of the process(es) of the program.

The fact that transition systems resemble traditional state machines, makes them an appealing approach as in addition to being capable of verifying a system, one readily has a well defined specification also. After-all, a transition may be associated with a desired level of abstraction.

At the time Petri nets were introduced [10], transition systems lacked elegant constructs for representing synchronization between processes. Compared to the other formalisms in those days, Petri nets specifically introduced the means for constraining concurrency between several transition systems. In principle, this gave one tools for representing process communication including mechanisms based on shared variables and message passing.

In Petri nets the variables of a program are represented using *places*. At any point

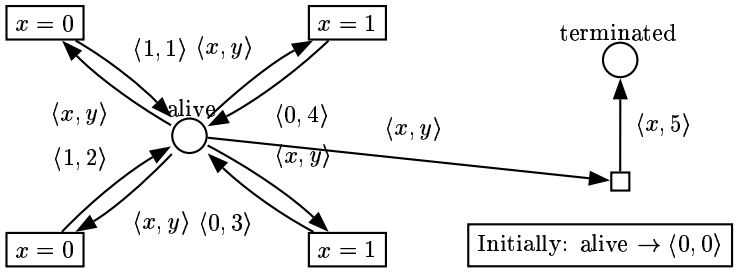


Figure 12: Petri net depicting the program listed in Fig. 10.
Each transition represents one choice in the selection.

of time, the contents of the places describe the current configuration. The places hold *tokens* which can be in turn thought as representing the variables of programs. Depending on the type of the Petri net, the tokens can be anything from "abstract dots" to simple variables to aggregated objects.

Transitions in Petri nets move tokens to and from places, an activity referred to as *firing* a transition emphasizing the basic principle that only one transition is fired at a time. A transition is thought as an instruction altering the valuation(s) of one or more variables of the system. Starting from the initial configuration one can construct the corresponding Kripke structure by firing enabled transitions one by one.

One may restrict the transitions to be fired until certain pre-conditions are met. This is called *constraining* the transitions forming the basis for the synchronization constructs.

Consider the Petri net depicted in Fig. 12 drawn according to common practice. Circles represent places and boxes represent transitions. Arcs connecting places with transitions are labeled by specifiers describing the tokens involved in the transition. The transitions in turn are labeled by constraints on the tokens. For instance, the transition in the low left-hand corner labeled "$x = 0$" requires one token of the type pair from place *"alive"* such that the first element of the pair equals zero. Therefore, whenever there is a pair available in place *"alive"* with its first element equaling zero, the transition labeled "$x = 0$" is enabled. Firing the transition causes the pair to be removed from place *"alive"*, the first element to be assigned value one and the second element value two, and finally the pair to be returned to place *"alive"*.

Representing the program introduced in Section 5.2, the net aggregates the variables of the program into a tuple, a pair. The values of the variables are altered by the transitions of the net representing the instructions of the program. The guards of the instructions are simply introduced as constraints in the transitions. The pair aggregating the two variables, x and y, is initially situated in place *"alive"* and the program can be considered terminated once the pair reaches the place *"terminated"* as there is no transition out from the place anymore.

The net in Fig. 12 is an example of *high-level* Petri nets where tokens may have structure and identity. High-level nets can often represent models in more compact form compared to those treating tokens without structure or identity [1, page 13]. The models considered in thesis are based on the ability of treating tokens as tuples.

Synchronizing processes

Petri net representing a process can be thought to consist of one particular token that represents the current state of the process. At any time, the token is in one of the places in the Petri net and associates the point of time with the state represented by the place. Transitions in turn can be thought to depict processing, i.e., by firing a transition the process can be thought to be given processing time and the processor running the instructions of the processes. Transitions thus effectively move programs from a configuration to another.

Recall the definition for synchronized resources based on mutexes. Remember also that a transition can be triggered only if all its incoming places contain the tokens required by the s constraints of the transition. Keeping the semantics of the Petri net as described above, we can now model a shared resource by a place and a token. Observe Fig. 15 depicting part of the model of the design of interest. Assume that a token t representing a process P is situated in place *Workers acquired*. By firing transition *Lock content storage* t would be moved into *Content acquired* P being thought to got hold of the corresponding content storage. But for the transition to be enabled in the first place, the token c representing the content storage would need to be in *Worker content storages*. Firing the transition simply removes c from *Worker content storages* disabling the transition *Lock content storages* for any other token in *Workers acquired*.

6.2 Promela

In the early 1980s, Holzmann introduced a protocol analyzer called *Pan* which later involved into the model checker *Spin* [8, 9]. The modeling language for Spin was in turn named *Promela*. When defining Promela, Holzmann emphasized interactions between concurrent processes while keeping the notation program-like. For instance, programmers familiar with ANSI-C [12] or C++ [13] are likely to find Promela intuitive to work with. Its syntax resembles closely that of ANSI C. Nevertheless, the foundations of the language lie in Dijkstra's *guarded command language* [14] and Hoare's *communicating sequential processes*, CSP [15]. The former contributes the syntax and semantics of conditional execution of statements while the latter introduces non-determinism into selections and communication between processes.

Roughly, Promela can be thought as ANSI C extended with a few communication primitives. In practice, however, Promela as a language is defined by what the parser of Spin accepts. This is a union of subset of ANSI C and the statements expressing communication between processes [3]. For instance, while one-dimensional arrays are supported, ANSI C structures (struct) or enumeration (enum) are not part of Promela "specification". Also the native data types (integers, floats, etc.) are limited. However, basic arithmetic and boolean comparison operators are supported and they have ANSI C syntax and semantics. As Promela is intended for modeling concurrency, it cannot be used to model recursion. In fact Promela does not even have the concept of function. The closest it can offer is macro definition and expansion, noted *inline*.

For modeling processes Promela introduces the concept of *process template* denoted a *proctype*. In ANSI C terms this can be viewed as a function definition describing a process body. Starting execution of a process means instantiation of a process template. Each

[3] Since version four it is been possible to embed fragments of ANSI C directly into the model, but as such they are not considered to be part of Promela itself.

template may be instantiated multiple times both in the startup and during the analysis. Therefore Promela allows one to model systems with a growing and shrinking amount of processes.

While the process template describes the behavior of a process, the statements within may describe both actions local to the process itself and actions effecting other processes. All variables declared in a Promela model are global no matter where they are declared in the source code of the model. Therefore process communication using shared variables is implicitly supported. Processes may also interact through special message channels offering easy way for modeling rendezvous. In Promela the basic concept of synchronous message passing of CSP is extended with polling mechanism and message buffering. The third mechanism can be seen as a special application of shared variables. Depending on a boolean condition a process may be blocked or may be given permission to execute statements guarded by the condition. The process remains blocked until the condition becomes enabled. Through the variables of a model a process may then effect the outcome of a boolean condition controlling whether another process is blocked or not. For instance, one can model mutexes using a single boolean variable and appropriate condition statements.

A configuration in a Promela model is a single valuation of all the variables of the model including the program counters of the processes. A corresponding Kripke structure $\langle S, S_0, R, L \rangle$ simply consists of the configurations and the successor relation between the configurations. This relation represents the effects of the atomic Promela statements on the configurations.

In Promela there are explicit mechanisms for reducing the state-space of the Kripke structure. Each statement in Promela is regarded as atomic step but one may declare sets of statements to be treated as atomic steps also. This can be done by placing a sequence of statements under *d_step* or *atomic* constructs. Both constructs declare that once a process starts to execute the sequence, the process may not be preempted until the last statement of the sequence is executed. This mechanism alone restricts any branching of the execution path until the end of the sequence. The former, *d_step,* also instructs the state-space generator to treat the entire construct as one transition in the Kripke structure. For instance, using *d_step*, one may add local processing details without affecting the Kripke structure.

6.3 Expressing requirements in LTL

Programs can vary greatly with respect to things ranging from the initial specification to the actual runtime behavior. Where the specifications differ in the level of detail, the level of correlation between the runtime behavior of a program and the state of the surrounding environment varies also. For instance, some programs can be sensitive to the load and timing of events in the system. Other programs may in turn depend heavily on the initial input, but be very tolerant to changes in the environment later on. Nevertheless, the properties to be verified often share common structure regardless of the level of detail or the dependence on the external state of fairs.

Berard et. al. [3, Pages 77-107] categorized claims about the correctness of a design or a program into

- claims of *reachability* saying if certain configuration can be reached,
- claims of *safety* saying that something never happens,

- claims of *liveness* saying that something will eventually happen and
- claims of *fairness* saying that something will happen infinitely often.

In practice, most of the properties of interest can be captured by combining the latter three. And one can even view claims of safety as negations of claims of reachability and fairness in turn as a special case of liveness.

Using *linear temporal logic* [4, Pages 27-32] one can combine claims of safety, liveness and fairness in various ways. In addition to expressing simple claims and their conjunctions and disjunctions (and/or respectively), LTL also enables one to use some properties as preconditions and some others as effects to form more complex claims such as

> *"if event A never happens and event B happens eventually, then event C will happen infinitely often".*

Using *the proportional logic* as a basis, the syntax of linear temporal logic is defined as follows.

Definition 7 *Linear temporal logic is the language of LTL formulas. Given a set AP of atomic propositions, an LTL formula is of the form:*

- *true, false, or p for any* $p \in AP$,
- $\neg f$, $f \vee g$, $f \wedge g$, $f \rightarrow g$, $f \leftrightarrow g$, Xf *or* $f \; \mathcal{U} \; g$, *where f and g are LTL formulas.*
- *Nothing else is an LTL formula.*

The semantics of LTL formulas are defined recursively starting from the atomic propositions. It should be noted that only atomic propositions are required to hold in specific *configurations* whereas formulas are required to hold in *the suffices* of the paths of *a Kripke structure*. Recall Section 5 for definitions.

Definition 8 *Let p be an atomic proposition, g and f any LTL formulas and* π *a path in some Kripke structure. The semantics is defined as follows:*

- $\pi \models p$ *holds if and only if p holds in the first configuration* π_0 *of the sequence* π *while* $\pi \models true$ *and* $\pi \models \neg false$ *always hold explicitly,*
- $\pi \models g \wedge f$ *holds if and only if* $\pi \models g$ *and* $\pi \models f$,
- $\pi \models g \vee f$ *holds if and only if* $\pi \models g$ *or* $\pi \models f$,
- $\pi \models \neg f$ *holds if and only if f does not hold in the path* π,
- $\pi \models f \; \mathcal{U} \; g$ *(read: f until g) holds if and only if there exists a suffix* π^j *of* π *such that* $\pi^j \models g$ *and that for every suffix* π^i *such that* $i < j$ *of* π, $\pi^i \models f$ *holds and*
- $\pi \models \mathcal{X} f$ *(read: next time f) in turn, holds if and only if* $\pi^1 \models f$ *holds for the suffix* π^1 *of* π.

For example, for *any* path π in the Kripke structure in Fig. 11 it holds that $\pi \models X_0 \wedge Y_0 \wedge \neg Y_5$. Atomic propositions X_0 and Y_0 hold in the configuration π_0, but Y_5 does not.

Connectives $\rightarrow$ and $\leftrightarrow$ are defined as semantic shorthands for *implications* just as for conventional proportional logic. That is, $\pi \models f \rightarrow g \equiv \pi \models \neg f \vee g$ and $\pi \models f \leftrightarrow g \equiv \pi \models (f \wedge g) \vee (\neg f \wedge \neg g)$.

In practice, one often utilizes the following short-hands for expressing the semantics of *eventually* and *always* instead of constructing them using $\mathcal{U}$ and X explicitly.

Definition 9 *Operator $\Diamond$ (read eventually) is defined as $\Diamond f_1 = true\ \mathcal{U} f_1$ and operator $\Box$ (read always) in turn becomes $\Box f_1 = \neg\Diamond\neg f_1$ for any LTL formula f_1.*

Semantically for any path π in some Kripke structure, $\pi \models \Diamond f_1$ says that formula f_1 must eventually be *true* on π. In other words there must exist an index i of π such that $\pi^i \models f_1$. Equivalently $\pi \models \Box f_1 \equiv \pi \models \neg\Diamond\neg f_1$ says formula f_1 cannot be false in any of the suffices $\pi^i, i \geq 0$. In other words, formula f_1 must hold in every suffix of π.

Keeping on referring to the Kripke structure in Fig. 11 it holds for any of its path π that $\pi \models \Box Y_5 \rightarrow \Box Y_5$, saying that once y gets assigned five, it will stay so infinitely. It should be noted that there are infinite paths that never assign five to y and the formula therefore holds as the *precondition* is not met. By applying formula $\Box\neg Y_5$ as precondition, one restricts to examine only the infinite computation paths as y is assigned five only when the program terminates. One can write for instance that for any path $\pi \models \Box\ \neg Y_5 \rightarrow \Box(X_0 \rightarrow \mathcal{X}\ X_1)$, saying that as long as the program does not terminate the assignment $x := 0$ is always followed by assignment $x := 1$. One could relax this by saying that as long as the program does not terminate every assignment $x := 0$ is eventually followed by assignment $x := 1$. Using LTL this would be $\pi \models \Box\ \neg Y_5 \rightarrow \Box(X_0 \rightarrow \Diamond\ X_1)$.

Definition 10 *Consider a Kripke structure K and some LTL formula f. The formula f is defined to hold in K, $K \models f$, if and only if for every computation path π in K it holds that $\pi \models f$.*

6.4 Strong and weak fairness in LTL

Using linear temporal logic, the fairness definitions in Section 5.3 can be re-stated. Consider an infinite computation path π representing execution of a program that performs a non-deterministic guarded selection S over and over again. Using proportional logic one can express a choice $c \in S$ being enabled by writing $enabled(c)$. The fact the choice was taken, can be expressed by writing $selected(c)$.

Definition 11 *Let there be a program that performs non-deterministic selection S infinitely often. Consider an (infinitely long) computation path π of the program. By writing $\pi \models \Box\Diamond\ enabled(c) \rightarrow \Box\Diamond selected(c)$ one says that the choice c is strongly fair on the path π. Respectively, $\pi \models \Diamond\Box\ enabled(c) \rightarrow \Box\Diamond selected(c)$ states that the choice is weakly fair on π.*

7 Models

This section constructs the models for the design introduced in Section 3.2 using Petri nets and Promela in turn. The construction of the two models should elaborate the fact that while the two languages differ greatly, they can both be easily applied to the modeling task in hand. Both provide expressive structures for representing synchronization between processes including mutual exclusion applied in the design. The major difference in practice is that weak and strong fairness are supported only in Maria, but not in Spin. Therefore one additionally needs to construct the support in the Promela model.

Section 2 emphasizes the importance of abstractions in specifications and designs. The same can be argued to apply for models also. In addition it being easier to start with an abstraction, simpler models can often be checked in less time and using smaller amount of memory. Mostly due to the latter, the models in this section make use of the abstractions introduced in Section 2. In addition,

- the outputs of the system are modeled using a single one and
- instead of having a variable number of worker processes, the number of processes is fixed.

The former abstraction is needed for keeping the sizes of the Kripke structures at bay. It forces every worker to use the same output and therefore making it more likely for the workers to be blocked. While this limits the behavior, it can still be argued to produce meaningful results by observing that the workers are the only ones using the individual outputs and that the executions of the workers do not branch in any point.

The latter abstraction is modeled by creating the fixed number of worker processes in the startup along the schedulers and the controller processes. The startup operations performed by the startup scheduler are simple considered as finite delays without kicking off any processes. The shutdowns operations in turn target the fixed amount workers over and over again only waiting for the corresponding workers to return their idle state. The worker processes are not terminated, but exist from the beginning of the analysis until the end.

In the next section the number of the worker processes is increased gradually in order to determine how the model checkers can cope with the corresponding increase in the size of the resulting Kripke structure.

7.1 Petri net model

The state machines in Figures 3, 5, 6 and 7 describing the design readily resemble Petri net notation. The states of the diagrams can be mapped directly to corresponding places of Petri nets. Only the mutex guarded resources of the design require special treatment.

Recalling the discussion in Section 6.1 describing how mutexes can be modeled with Petri nets, one has straightforward translation from the design into the model. Mapping the states directly into places, the transitions between the places are augmented with constraints modeling the mutex acquisitions and releases. The mutexes are then introduced as tokens that are moved to and from their initial places by the transitions. The absence of the token disables a transition requiring it and therefore models the blocking effect of a reserved mutex.

Taking advantage of the high-level nets and giving tokens identity, the initial places needed for the mutexes could be folded into one. However, for the sake of readability, they are folded into the seven places listed in Table 1.

Places	Tokens
Worker content storages	one token for each storage
Scheduler storages	token named "the_startups" representing the startups storage and token named "the_shutdowns" representing the shutdowns storage
Workers storage	token named "the_workers" representing access to all workers
Outputs storage	token named "the_outputs" representing access to all outputs
Outputs	token named "the_output" representing the only output of the model

Table 1: Places and tokens modeling the mutex guards of the shared resources of the design.

The resulting Petri net model is partitioned into three each part emphasizing synchronization between two or more processes of the design. The parts are listed in Table 2 and are discussed in turn.

Part	Processes	Synchronizer
First	Workers and Controller	Worker content storage tokens and "the_output"
Second	Schedulers and Controller	"the_shutdowns", "the_startups" and "the_workers"
Third	Shutdown scheduler and Controller	"the_shutdowns", "the_outputs" and "the_workers"

Table 2: Partitioning of the Petri net model.
Note the difference between "the_output" and "the_outputs" representing two separate resources.

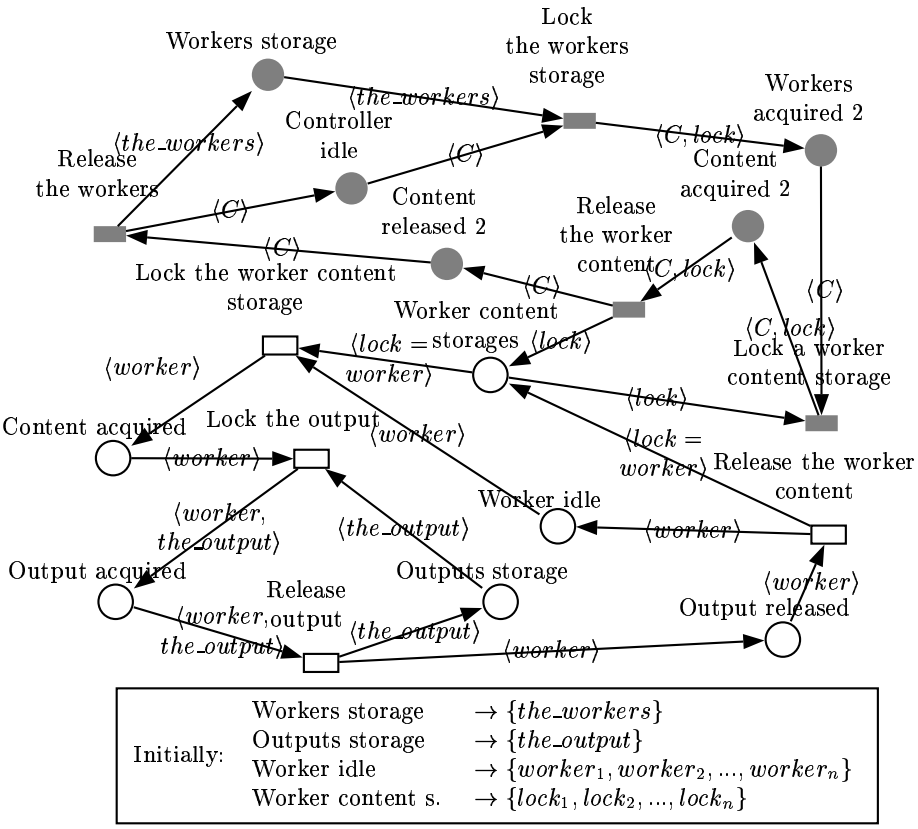


Figure 13: The first part of the model.
The shaded places and transitions depict the controller the rest representing the workers.

Workers and the controller

The first part of the model depicted in Fig. 13 represents worker processes and that part of the controller process that interacts with worker content storages. Worker processes are folded under high-level Petri net notation. That is, tokens representing worker processes use seemingly same places and transitions.

All worker tokens are initially situated in place *"Workers"* which represents the *"Worker idle"* state in Fig. 3. Places *"Content acquired"*, *"Output acquired"* and *"Output released"* represent the rest of the states of the workers correspondingly.

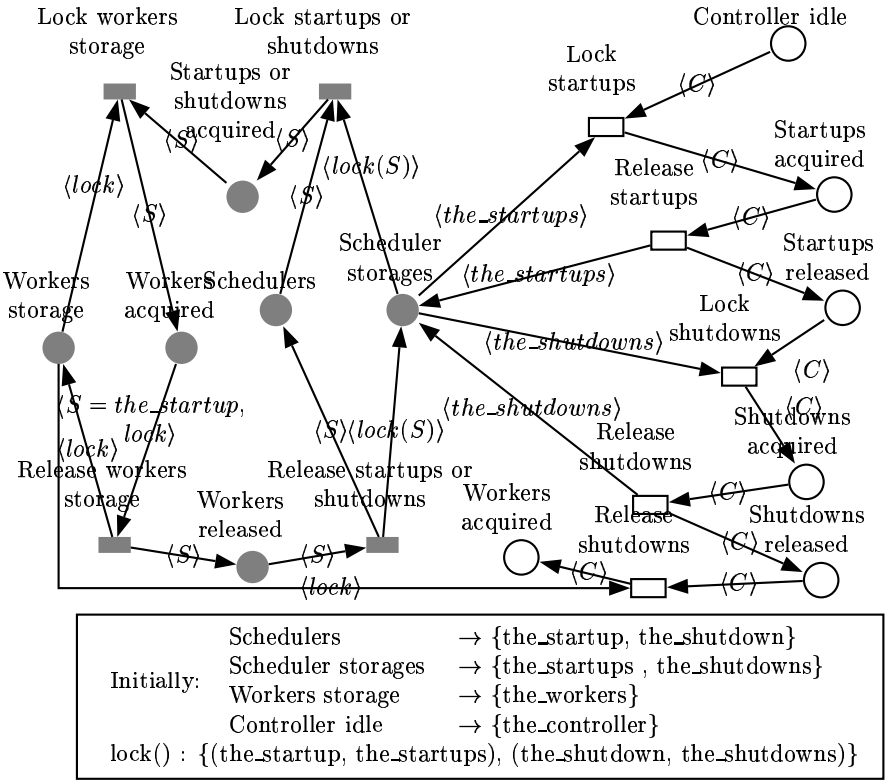
The only output token of the system is initially situated in place *"Outputs"* from which the token is moved into place *"Output acquired"* whenever transition *"Lock output"* is fired. Firing transition *"Release output"* in turn moves the output token back into place *"Outputs"*. The presence of the output token in *"Outputs"* control the transition *"Lock output"*. Effectively those worker processes that are in place *"Content acquired"* and desire to use this output, are either enabled to proceed or blocked respectively. Worker content storages are represented by one token each in similar fashion. They are moved to and from their initial place *"Worker content storages"* by transitions *"Release the worker content"* and *"Lock the worker content storage"* respectively. Like the "the_output" token controls the transition *"Lock the output"*, content storage tokens control transition *"Lock the worker content storage"* blocking executions of workers and the controller whenever

the token is absent in the place *"Worker content storages"*. Therefore the controller and the workers may block each others execution only through the worker content storages.

The shaded part of the net represents the controller and specifically the states to and from the state *"Controller idle"* in the upper loop of Fig. 5.

Schedulers and controller

The second part, depicted in Fig. 14, models worker life-cycle setting and removal performed by the controller. These actions are part of worker creation and deletion. In the state machine in Fig. 5 these are represented by the sequence of states starting from state *"Controller idle"* and ending in the state *"Shutdowns acquired"*.



Initially:	Schedulers	→ {the_startup, the_shutdown}
	Scheduler storages	→ {the_startups , the_shutdowns}
	Workers storage	→ {the_workers}
	Controller idle	→ {the_controller}
lock() : {(the_startup, the_startups), (the_shutdown, the_shutdowns)}		

Figure 14: The second part of the model.
The shaded places and transitions on the left represent the schedulers while the controller is depicted on the right hand side.

The shaded area in Fig. 14 models the common parts of both schedulers. It covers startup scheduler entirely, but leaves shutdown scheduler to be augmented in the third part of the model. The shaded are in Fig. 14 folds both schedulers in similar fashion as workers are folded in Fig. 13. Place *Schedulers* represents both states *"Startup scheduler idle"* and *"Shutdown scheduler idle"* found in state machines in Figures 6 and 7 respectively. Place *"Startups or shutdowns acquired"* similarly represents states *"Startups acquired"* and *"Shutdowns acquired"*. Unfolding this part of the net semantically means that transition *"Lock startups or shutdowns"* is enabled for startup scheduler when "the_startups" token is present in place *"Scheduler storages"* and for shutdown scheduler when "the_shutdowns" token is present. Shutdown scheduler deviates from the path of

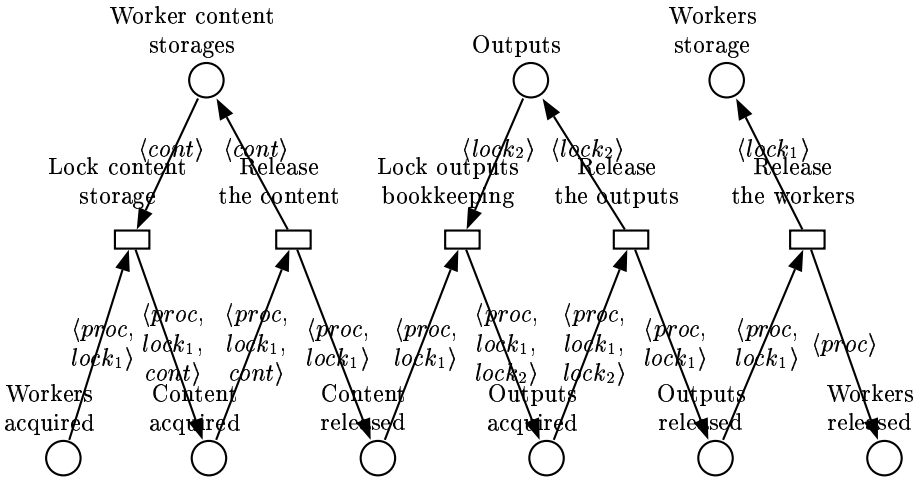


Figure 15: The third part of the model representing worker terminations.

startup scheduler after place *"Workers acquired"*. As the startup scheduler aims to proceed by releasing the storages, the shutdown scheduler still has the outputs storage to acquire. This is the behavior that is augmented in the third part of the model. The schedulers share the model again from the place *"Workers released"* on-wards. Shutdown scheduler is explicitly prevented from using transition *"Release workers storage"* by requiring place *"Workers acquired"* to provide the *"the_startup"* token as precondition.

When the schedulers cause the transition *"Lock startups and shutdowns"* to be fired, the corresponding storage tokens are removed from place *"Scheduler storages"*. By doing this the controller is prevented from proceeding either to *"Startups acquired"* or *"Shutdowns acquired"* depending on the storage being locked. Correspondingly, by firing transition *"Lock startups"*, for example, the controller prevents startup scheduler from proceeding from place *"Schedulers"* into place *"Startups or shutdowns acquired"*. The only way the two schedulers interact on each other, is by locking the workers storage and preventing the other from using it. This synchronization mechanism is represented by the transition *"Lock workers storage"*.

Shutdown scheduler and controller

The third and final part of the model in Fig. 15 augments the actions of the controller and the shutdown scheduler left open in the part two. The third part represents the net between the shutdown scheduler places *"Workers acquired"* and *"Workers released"*. By renaming the place *"Workers acquired"* into *"Workers acquired"* and the place *"Workers released"* into *"Controller idle"* in the third part, one constructs the missing section for the controller also.

7.2 Promela model

Modeling with Promela provides the designer variety of tools for representing both internal and external behavior of processes. The abundance of techniques naturally provides great flexibility, but also requires the designer to bear attention to their combined effects on the state-space generation. After all Spin has to be capable of analyzing the model efficiently. The design studied in this thesis does not require much from the modeling formalism on

its own, but the fairness properties required from the process scheduling are completely different matter altogether. The lack of built-in support for fairness constraints in Spin has fairly fundamental effect on the model. The support for the constraints needs to be constructed by the designer having roughly three possible approaches:

- to model the scheduler with strong/weak fairness properties,
- to formalize fairness constraints as conditions in liveness claims or
- to combine the former two.

Before going into details, there are a few apparent problems in each approach that should be considered first. If one recalls that fairness is a property defined meaningfully only for infinite computations, the first approach is far from being trivial to accomplish [11, pages 2-3]. Addition of the fairness constraints into the liveness claims may in turn impose a great burden on the LTL to automaton translation [16], not to mention the possible blow-up in the size of the state-space of the model. Claims in LTL often translate into non-deterministic automata which may have a great impact on the model checking. Badly modeled scheduler and LTL claims together may result model unfeasible to be analyzed in addition to the time wasted on construction.

Another fundamental modeling decision relates to the scheduling and native Promela constructs. Effectively it is a decision whether to model even the basis of the process scheduling or merely try to instruct Spin how the scheduling should be done. The corresponding alternatives are:

- to model the scheduler as native Promela process executing instruction(s) of one process of the design at a time or
- to model each process in the design as native Promela process using the *proctype* construct.

The straightforward choice would be the latter having one-to-one mapping between processes in the design and native processes in the model. Furthermore, section 4.2 readily hints a simple technique for modeling mutexes with native Promela processes. Only the fairness constraints are not addressed at all.

Monitoring processes

Constraining or forcing a process to be treated strongly or weakly fair requires one to somehow *follow up* when the process is *enabled* to execute and when the process is *selected* to execute. A scheduler forcing fair behavior would use the information to determine if a certain process needed to be selected. Similarly a model checker restricting its analysis to cover only fair executions would use the information for filtering out the unfair executions.

Consider the scheduler depicted in Fig. 16. The subroutines *state_machine()* represent selections transferring the corresponding processes from one configuration to another. The subroutine *pre_fairness()* in turn records if there are any transitions enabled for each of the processes in their current configurations. Note that *state_machine(Px)* is executed only if *pre_fairness()* finds at least one enabled transition for the Px. The subroutine *post_fairness()* simply records which of the *state_machine(Px)* got executed, i.e., which of the process was selected.

```
do /* proctype(scheduler) */
pre_fairness()
    if
    :: state_machine(P1)
    :: state_machine(P2)
    :: ...
    fi
post_fairness()
od
```

Figure 16: A scheduler monitoring fairness of its execution.

The scheduler in Fig. 16 represents a single native Promela process. The responsibility for scheduling the modeled processes $P1$, $P2$ and so on is therefore taken completely from Spin and as far as Spin is concerned, there is only one process to be examined. While giving the designer total control over the scheduling, this approach prevents Spin from using any optimizations requiring visibility to the concurrency of the model. And it could further be argued that one should rather be using a tool instead of working against it.

Letting Spin to handle the scheduling requires the subroutines *pre-* and *post_fairness()* to be embedded into the individual process bodies. This scheme is depicted in Fig. 17. Observing that *post_fairness()* is only needed when a process is selected and that even then it needs to update only information concerning the process itself, makes it straightforward to be embedded. The subroutine *pre_fairness()* in turn always needs to consider all the processes, i.e., which of the processes were enabled and which in turn were disabled by the last transition. Luckily enough, Spin already records this and provides a routine called *enabled(P)* for querying if a process P is enabled in the current configuration. One may therefore drop *pre_fairness()* in the model altogether.

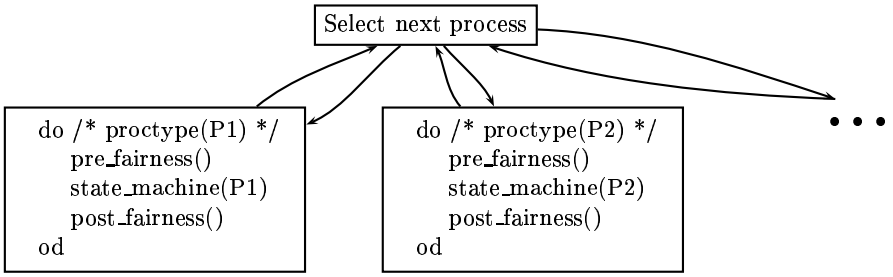


Figure 17: Scheduling processes with Spin.

Modeling *post_fairness()* can be done simply by introducing a boolean variable set initially to *false* for each the processes of the model and setting the value of the variable to *true* and back to *false* whenever the process is transitioned from a configuration to another. The fact that the variable just gets assigned the value *true* is enough to constrain weak or strong fairness. For instance, strong fairness constraint for process i could be expressed as $\Box\Diamond enabled(i) \rightarrow \Box\Diamond(selected_i = true)$ where $selected_i$ is the boolean variable associated with the process i.

```
IDLE:
do :: { /* worker(w) */
    atomic { (content_w = false) → selected_w := true;
        content_w := true; selected_w := false };

CONTENT_ACQUIRED:
    atomic { (the_output = false) → selected_w := true;
        the_output := true; selected_w := false };

OUTPUT_ACQUIRED:
    atomic { selected_w := true; the_output := false; selected_w := false };

OUTPUT_RELEASED:
    atomic { selected := true; content_w = false; selected_w := false }
} od
```

Figure 18: Worker process bodies expressed in Promela.

The process bodies

On the abstraction level of Section 3.2, the process bodies become simple repetitions of assignments of boolean variables, namely the ones representing mutexes and the ones representing process selections. For instance, worker process bodies become of the form listed in Fig. 18.

The labels highlighted by the capital letters are included for easing the claim formulation. One may refer to specific configurations of the processes using the labels. The use of the "atomic" construct is needed to combine a mutex operation and the corresponding update of a selected variable into atomically executable unit.

The branching of the controller process in Fig. 5 requires special attention. Unlike the rest of the processes which have only one outgoing transition from each state of the design, controller has to two going out from the state *"Controller idle"*. In the model this can be expressed with non-deterministic selection between the two branches shown in Fig. 19.

```
IDLE:
do {
:: atomic { (the_startups = false) → selected_c := true;
        the_startups := true; selected_c := false };
   STARTUPS_ACQUIRED:
   ...
   WORKERS_ACQUIRED:
   WorkerContentSelect();
   ...
   atomic { the_startups = true; selected_c := true; selected_c := false }

:: atomic { (the_workers = false) → selected_c := true;
        the_workers := true; selected_c := false };
   WORKERS_ACQUIRED_2:
   ...
  atomic { selected_c := true; the_workers = true; selected_c := false }
} od
```

Figure 19: The branching of the controller expressed in Promela.

```
WorkerContentSelect()
  if
      :: atomic { (contentLocks[0] = false) → selected_c := true;
           contentLocks[0] := true; selected_c := false };
CONTENT_ACQUIRED_0:
         atomic { selected_c := true; contentLocks[0] = true; selected_c := false };
CONTENT_RELEASED_0:
      :: atomic { (contentLocks[1] = false) → selected_c := true;
           contentLocks[1] := true; selected_c := false };
CONTENT_ACQUIRED_1:
         atomic { selected_c := true; contentLocks[1] = true; selected_c := false };
CONTENT_RELEASED_1:
         ...
      :: atomic { (contentLocks[n] = false) → selected_c := true;
           contentLocks[n] := true; selected_c := false };
CONTENT_ACQUIRED_N:
         atomic { selected_c := true; contentLocks[n] = true; selected_c := false };
CONTENT_RELEASED_N:
  fi
```

Figure 20: Non-deterministic selection between n workers of the model.

Recall that both the controller and the shutdown scheduler may consider any of the workers within the termination procedure. This behavior and the selection between the workers can in turn be modeled using non-determinism. The Promela fragment listed in Fig. 20 represents the precise mechanism used for both the controller and the shutdown scheduler. The fragment is referred to in Fig. 5 by "WorkerContentSelect()".

In practice, however, the Promela source code for the models to be used in the analysis is produced with heavy combination of GNU ANSI C precompiler [17] and GNU stream editor called *sed* [18].

The controller and neither of the schedulers (startup, shutdown) may hold the workers storage at the same time.
Schedulers (startup, shutdown) may not hold the workers storage at the same time either.
The controller and startup scheduler cannot hold the startup storage at the same time.
The controller and shutdown scheduler cannot hold the shutdown storage at the same time.
Two worker processes cannot hold the same output at the same time the controller and any of the workers cannot hold the corresponding worker content storage at the same time.

Table 3: Safety properties forcing data integrity.

8 The requirements

The models constructed in previous section make it possible for Maria and Spin to represent the computations of the design of interest. This section builds the corresponding LTL formulas for representing the requirements stated in Section 3.4.

A common practice is to check for pure safety properties first moving on to liveness only when the design passes the safety checks. This is because claims of safety are about conditions happening after *finite runs* and one can often express them simply as assertions. These kind of safety properties are the *invariants* of the system. Liveness in turn always considers *infinite runs* and therefore results more complex claims that cannot be expressed as assertions anymore.

The approach starting with safety properties is also supported by examining the requirements of interest. Checking for the integrity of a shared storage can be seen as equivalent to asserting that no two processes may enter their corresponding critical sections simultaneously. The second requirement about finiteness of operations and the third requirement about processes progressing infinitely can in turn be viewed as equivalent to checking for absence of both deadlocking and starvation. Where the first requirement is about validating that the mutexes to their job correctly, the latter two are about checking that the use of mutexes does not cause any ill side-effects. And observing that it would be rather foolish to check for the side-effects before checking that the purpose is fulfilled, it makes sense to start with the storage integrity.

The first requirement claims that a certain condition must never occur, making it a pure claim of safety while the latter two address both issues of safety and liveness. Deadlocking can be seen as belonging to safety category while starvation is always about a condition lasting infinitely placing it into the liveness category.

8.1 Safety properties as simple assertions

The safety properties to be checked are listed in Table 3. The translation of each entry into the context of the models can be started by partitioning the places of the processes involved into subsets, those holding the resource of interest and those that do not. For instance, the first entry regarding the worker content storages, the controller and the schedulers, results three sets of places holding the workers storage. The sets are for the controller, the shutdown scheduler and the startup scheduler in turn:

$$
\begin{aligned}
C_{workers} = \quad & \{Workers\ acquired, Content\ acquired, Content\ released, \\
& Outputs\ acquired, Outputs\ released, Workers\ acquired\ 2, \\
& Content\ acquired\ 2, Content\ released\ 2\}, \\
SH_{workers} = \quad & \{Workers\ acquired, Outputs\ acquired, Outputs\ released\} \text{ and} \\
ST_{workers} = \quad & \{Workers\ acquired\}.
\end{aligned}
$$

The first safety requirement can now be understood as stating that in case any place $p \in C_{workers}$ is occupied by the controller process token, then none of the places in $SH_{workers}$ and $ST_{workers}$ may hold the shutdown or the startup scheduler token respectively. Therefore one needs to consider every pair (a, b) such that $a \in C_{workers}$ and $b \in (SH_{workers} \bigcup ST_{workers})$ in turn. For instance, for any places $p_1 \in C_{workers}$ and $p_2 \in SH_{workers}$ one may write for Maria

$$reject\ !(place\ p_1\ equals\ empty)\ \&\&\ !(place\ p_2\ equals\ empty)\ \&\&\ fatal;$$

saying that p_1 and p_2 may not be occupied simultaneously. For Spin using *never claims*, $never(c) \equiv c \rightarrow assert(\neg\ c)$, the same could expressed as

$$never(controller@L_{p1}\ \&\&\ shutdownScheduler@L_{p2});$$

where L_{p1} and L_{p2} represent the labels in the Promela model corresponding to the places p_1 and p_2 respectively. The notation $P@L$ simply represents a boolean function telling if the process P is currently in a configuration corresponding to the label L in the model.

8.2 Fairness assumptions

The design does not introduce any mechanisms for preventing any of the processes to be ignored infinitely. Instead it relies on the process scheduling to force this behavior. The requirement of finiteness of operations and the one of processes executing infinitely are therefore relaxed to hold against the following *fairness assumptions*:

- all critical section exits are weakly fair and
- critical section entries are strongly fair except
- those of them which move a process out from its idle state.

All the transitions of the processes belong to one and only one of the three categories. Semantically the assumptions are equivalent to saying that in case a mutex is released infinitely often, every process that waits for the mutex infinitely often is finally given the mutex and that every process that is about to release a mutex does not delay the release infinitely either. Therefore in addition to requiring the process scheduler not to deny any process access to the processor(s) infinitely, the design requires the scheduler not to deny access to the storages infinitely either. However, nothing is said about processes leaving their idle states. Recall that the properties to be checked only require processes to return to their idle states once the idle state is left. This behavior does not require any assumptions about processes leaving their idle states.

It should be noted that these assumptions only require each process to get a slice of the resources of the system saying nothing more about how the slicing should be done.

In other words, as long as every process gets even the smallest possible slice, the slicing can be as uneven as it can get.

In case of Maria the fairness assumptions can be taken into account by declaring the mutex locking transitions strongly fair and the transitions releasing mutexes weakly fair [22].

In Spin, however, such built-in mechanisms do not exist, and one is forced to bring in fairness in terms of preconditions in the LTL claims.

Fairness and Spin

In Section 7, the Promela model was augmented with *selected* variables for each of the processes of the model. These can now be used in combination with the *enabled()* function of Spin to force the strong and weak fairness constraints on the mutex operations. For a process P all its transitions could be constrained to be strongly fair by the following LTL formula:

$$F = \Box\Diamond enabled(P) \rightarrow \Box\Diamond selected_P$$

stating that P must be selected infinitely often if its enabled infinitely often. Recalling that the transitions out from the idle states need not to be restricted, one augments F further into:

$$F_{aug} = \Box\Diamond(enabled(P) \wedge \neg idle(P)) \rightarrow \Box\Diamond selected_P$$

stating in addition to F that P does not need to be selected infinitely often in its idle state even if its enabled there infinitely often.

Still, F_{aug} does not make any distinction between mutex acquisitions and releases. However, a closer examination of the design of interest reveals that, each of the *mutex releases* forced to be strongly fair is in fact explicitly also weakly fair and vice versa. This is because a process stays enabled until it is blocked by a mutex, and that a process about to release a mutex cannot become blocked by another before it has released the first. In other words, a process trying to release a mutex stays enabled until the mutex is released when in turn the process becomes selected.

For all of the processes of the model the combined fairness constraint becomes:

$$\begin{array}{rlll}
 & (\Box\Diamond(enabled(C) \wedge \neg idle(C)) & \rightarrow \Box\Diamond selected_C) & \wedge \\
 & (\Box\Diamond(enabled(SH) \wedge \neg idle(SH)) & \rightarrow \Box\Diamond selected_{SH}) & \wedge \\
 & (\Box\Diamond(enabled(ST) \wedge \neg idle(ST)) & \rightarrow \Box\Diamond selected_{ST}) & \wedge \\
\mathcal{F}_{all} = & (\Box\Diamond(enabled(W0) \wedge \neg idle(W0)) & \rightarrow \Box\Diamond selected_{W0}) & \wedge \\
 & (\Box\Diamond(enabled(W1) \wedge \neg idle(W1)) & \rightarrow \Box\Diamond selected_{W1}) & \wedge \\
 & \ldots & & \wedge \\
 & (\Box\Diamond(enabled(Wn) \wedge \neg idle(Wn)) & \rightarrow \Box\Diamond selected_{Wn}). &
\end{array}$$

In practice, however, already three workers ($n = 2$) hit the limits of even the most efficient LTL translators. To the current knowledge a translator called *ltl2ba* from Denis Oddoux [16] should be the most scalable one, but an LTL claim with six or more strong fairness formulas is too much even for *ltl2ba*.

Another look at the model hints a way of simplifying $\mathcal{F}_{all}$ from the translator point of view while still keeping the overall verification task meaningful. The outputs of the

design are used solely by the worker processes, and the outputs in turn are modeled using the single mutex called *the_output*. Assuming *the_output* is released infinitely often, one could constraint the acquisition of *the_output* to be strongly fair by constraining every acquisition to succeed eventually. Formalizing this notion into $\mathcal{F}_{all}$ results the following:

$$\begin{aligned}
\mathcal{F}'_{all} = \; & (\Box\Diamond(enabled(C) \wedge \neg idle(C)) \rightarrow \Box\Diamond selected_C) \wedge \\
& (\Box\Diamond(enabled(SH) \wedge \neg idle(SH)) \rightarrow \Box\Diamond selected_{SH}) \wedge \\
& (\Box\Diamond(enabled(ST) \wedge \neg idle(ST)) \rightarrow \Box\Diamond selected_{ST}) \wedge \\
& (\Diamond\Box(enabled(W0) \vee worker[0]@CONTENT_ACQ.) \rightarrow \Box\Diamond selected_{W0}) \wedge \\
& (\Diamond\Box(enabled(W1) \vee worker[1]@CONTENT_ACQ.) \rightarrow \Box\Diamond selected_{W1}) \wedge \\
& \ldots \wedge \\
& (\Diamond\Box(enabled(Wn) \vee worker[n]@CONTENT_ACQ.) \rightarrow \Box\Diamond selected_{Wn})
\end{aligned}$$

requiring each of the worker processes to proceed eventually once the worker has acquired its content regardless of the rest of the system.

The additional assumption about the acquisitions of outputs succeeding eventually, of course weakens the results compared to those obtained from Maria. But verifying even this can be seen to have its merits. Even if nothing else, one can now compare Maria and Spin.

8.3 Liveness properties in LTL

Now that the fairness assumptions are in place, one may formulate claims for the requirements in Section 3.4 not covered by the safety assertions stated in Section 8.1. The second requirement can be seen to be equivalent to requiring each process not in its idle state to return to the idle state eventually. Only if an operation took infinitely long, would it be impossible for a process to return to its idle state.

The third requirement for the controller process can be seen to be covered implicitly by the very same claim. If all the processes can be shown to always return to their idle states, then all of the processes are always capable of starting over again.

For Maria the claim requiring the controller to always return to its idle state, could be written as:

$$\text{Controller controller \&\& } \Box\Diamond(controller\ subset\ place\ controller\_idle),$$

while for Spin, including $\mathcal{F}'_{all}$, the claim becomes:

$$\mathcal{F}'_{all} \rightarrow \Box\Diamond controller@CONTROLLER\_IDLE.$$

In case of Spin, the second requirement is meaningful only for the controller, the startup scheduler and the shutdown scheduler. This can be observed from the $\mathcal{F}'_{all}$ effectively forcing a worker to return to its idle state eventually. Therefore, not to burden the LTL to automata translator beyond its limits, Spin is used to verify the following claim instead:

$$\begin{aligned}
\mathcal{F}'_{all} \rightarrow \; & (\Box\Diamond controller@CONTROLLER_IDLE \; \wedge \\
& \Box\Diamond shutdown@SHUTDOWN_IDLE \; \wedge \\
& \Box\Diamond startup@STARTUP_IDLE).
\end{aligned}$$

Obviously one now relies mostly on Maria regarding the requirements two and three.

9 Verifying the claims

Given a model and a set of claims, safety or liveness, a model checker essentially verifies that the properties represented by the claims hold in every path of the Kripke structure represented by the model in turn. Specifically, a claim C and a Kripke structure K_M of a model M are thought to represent sets $S_C = \{p \mid p \models C\}$ and S_K of computation paths respectively and a model checker is thought to examine if $S_K \subseteq S_C$.

Using the classic language theory, every computation path can be thought as a word, the sets S_C and S_K then becoming languages each. The claim representing S_C is thought to be translated into automata A_C accepting the language $\mathcal{L}(S_C)$ while the Kripke structure K_M itself is thought as a device accepting $\mathcal{L}(S_K)$. In theory, one forms the product of K_M and the complement of A_C and checks if the language accepted by the product $K_M \times \overline{A_C}$ is empty. If not, then any of the words in $\mathcal{L}(K_M \times \overline{A_C})$ serve as an example of a computation that violates the claim C.

The computational bottleneck of the model checking usually resides in the generation of the Kripke structure. If the level of detail, i.e., the number of states, variables and concurrent processes in the model, is increased, the resulting Kripke structure tends to increase exponentially in size. This phenomenon is known as *the state-space explosion* [5]. Model checkers alleviate the problem by applying clever algorithms such as the *partial order reduction* [4] used in Spin. This work examines the phenomenon in practice by gradually increasing the number of workers in the models and by observing the corresponding increase in the computation times needed by Maria and Spin. The aim is to learn about the limits of Maria and Spin and to perform comparisons.

9.1 The results

The safety properties of the design can be verified using Maria up to eight workers. Despite of generating larger state-space than Maria, Spin accomplishes to verify the safety claims up to eight workers also. The details are plotted in Figures 21 and 22. It should be noted that for safety checks, the use of the selected variables in the Promela models is omitted. These are only needed for the liveness checks and the inclusion here would be simply increase the size of the Kripke structure unnecessarily.

The processing times needed for the liveness analysis by Maria are also listed in Fig. 21 while the corresponding data for Spin can be found in Fig. 23. However, the results should be compared with some care. First, the lack of built-in support for strong and weak fairness in Spin causes the Promela model to be extended which in turn results an increase in the Kripke structures. Second, the heavy LTL to automata translation required by the Promela models both add up to the total processing time and prevent Spin from being applied for more then five workers. Third, as the claims and the fairness assumptions differ, the computations done by Maria and Spin are not directly comparable either. And fourth, where Maria lists only the state space for the model, Spin includes the states corresponding to the LTL claims also.

In order to determine the total processing time needed for the Promela models, one needs to take into account the processing times needed by the ltl2ba for the translations. The processing times needed for the LTL translation with respect to the number of workers are depicted in Fig. 24. It is interesting to find out that Spin itself is not the bottleneck as Spin requires only a fraction of the time needed by ltl2ba.

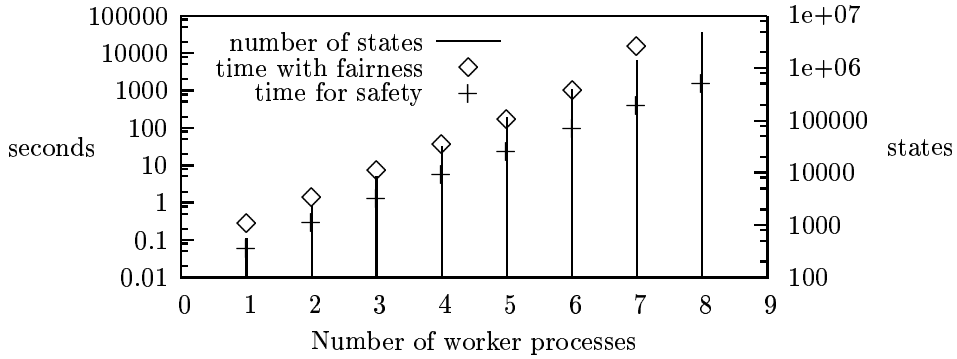


Figure 21: The computation times needed by Maria.
Liveness with eight workers and safety with nine workers cause Maria to run out of memory.

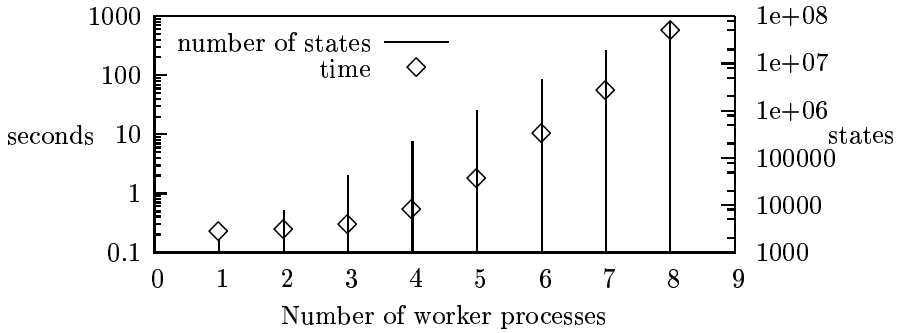


Figure 22: The computation times needed by Spin for safety checks.
Spin runs out of memory when the number of workers is increased to nine.

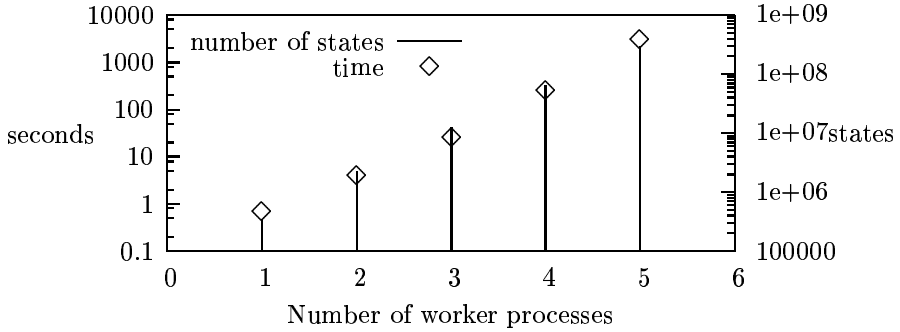


Figure 23: The computation times needed by Spin for liveness checks.
The LTL to automata translator, ltl2ba, fails to translate the claims when the number of workers is increased to six.

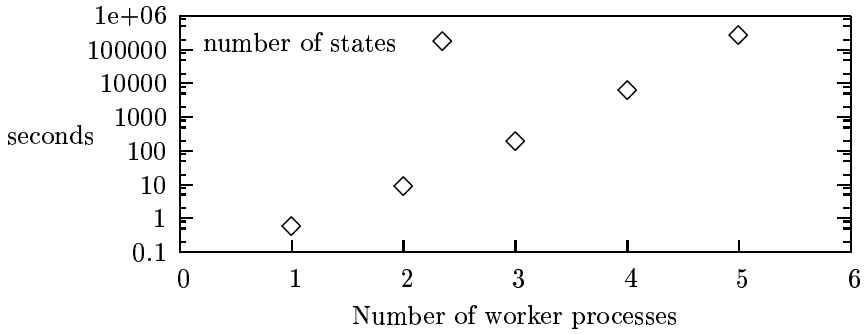


Figure 24: The computation times needed by ltl2ba for liveness claims.

Comparisons between the state spaces and the computation times

The relative computation times listed in Fig. 25 and the ratios of the sizes of state spaces between Maria and Spin listed in Fig. 26 shed some light into the overall performance of the two tools. The state space generated by Spin can be seen to increase faster than the one generated by Maria but Spin, however, seems to cope with the increase. From one to five workers, the ratio between the total number of states and total amount of time needed for processing, increases in exponential rate. The drop in the rate for eight workers is due to the search stack becoming too large to hold in RAM, and Spin is forced to use the disk for storing parts of the stack. In case of Maria the rate just decreases slowly, but stays well below that of Spin except for models with only one worker. The difference can be seen to be in the maximum roughly one hundred fold for models with five workers.

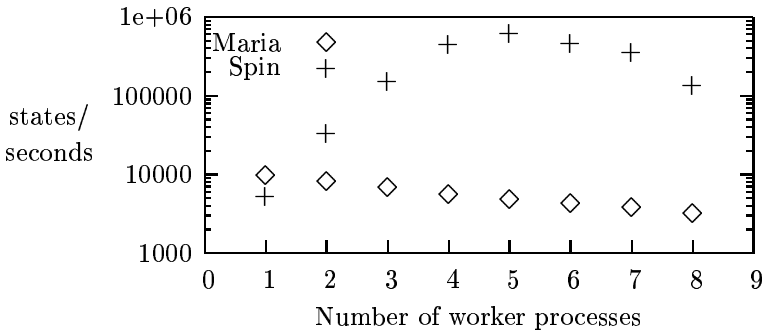


Figure 25: The number of states generated with respect to the time needed for the safety checks.

The steady growth in the ratio between the state spaces of Spin and Maria can be explained by observing how the workers are represented in Spin. Each worker corresponds to a native Spin process which in turn has specific start and end states. Despite the worker processes ever exiting their loops, the end states and their contribution to the number of configurations is taken into account by Spin. Hence the greater state space, i.e., the greater number of configurations in the reachability graph of Spin.

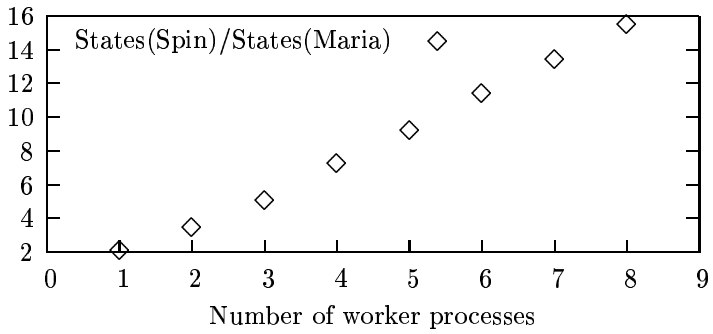


Figure 26: The ratio of the number of states generated by Spin and Maria in safety checks.

10 Conclusions

Regardless of the actual tool being used, model checking can be argued to have its place in the design process of software systems. First, the types of properties examined in this thesis are widely encountered and while their validation through conventional testing is often difficult and resulting in vague results, application of model checking can be seen to reduce the validation task into a rather straightforward thing to do. Second, the amount of work required for modeling a concurrent system need not be time consuming. The task can be partitioned into several steps by applying an appropriate level of abstraction in each step. The design in this work can be seen to lack quite a lot of detail while still enabling non-trivial properties to be verified. And third, there are efficient tools available for model checking providing powerful formalisms for modeling designs. The two tools, Maria and Spin examined here, are both good examples.

Concurrent software designs often impose requirements for the underlying operating system and one is therefore often forced to take these requirements into account when verifying the requirements of the design itself. In the problem studied here the most demanding task was the modeling of the fairness constraints on the process scheduling. While in Maria this was rather straightforward, in Spin it required a lot of trial and error. One needed to find a good balance between constraints in the Promela model and in the LTL claims. Although in the end the resulting (Spin) model did not accomplish to verify all the properties set in the beginning, such an exercise gave valuable insight to the subject of fairness itself both theoretically and in practice. Modeling fairness requires one to give a lot of thought how fairness is taken into account in real world process scheduling.

The chosen modeling formalism, Petri nets and Promela, proved to fit into task in hand well. The actual logic of the design was easy to model and thanks to the expression power of both languages the models could be kept small and compact. The big difference in the syntax and structure of the languages acted as a good teacher into deeper understanding of the actual processing done by the model checkers. After-all both the Petri net and the Promela model need to produce equivalent Kripke structures.

Between Maria and Spin, two rather obvious differences can be pointed out. While for the modeling task in hand Spin requires greater state space, Spin seems not to need significantly more time to process it. In fact, Spin accomplishes the task in less time than Maria as long as the properties can be expressed as simple assertions. But once the properties require one to include assumptions about fairness in concurrency, Maria begins

to gain rapidly on Spin. The built-in mechanism enabling one to constraint the model with strong and weak fairness gives Maria a clear edge compared to Spin [22]. In case of Spin such built-in mechanisms do not exist, and one is forced to augment the model and the LTL claims resulting in a significant decrease in the performance of Spin. The additions in the claims in turn causes one to hit the limits of the tool (ltl2ba) responsible for translating the claims for Spin, and therefore preventing one from applying Spin for more complex models.

This work readily suggests a few apparent subjects of further research. Augmenting Spin to provide mechanisms for constraining models with strong and weak fairness would increase the applicability of Spin in the field of concurrent software systems. Additionally, by increasing the scalability of LTL to automata translators, one would enable Spin to be applied to a wider range of models also.

References

[1] Marko Mäkela. Efficient Computer-aided Verification of Parallel and Distributed Software Systems. *Ph.D Thesis, Helsinki University of Technology, Laboratory of Theoretical Computer Science, Research Reports 81*, 2003.

[2] Jonathan Billington. High-level Petri Nets - Concepts, Definitions and Graphical Notation. *ISO/IEC 15909-1 Software and Systems Engineering*, 2003.

[3] B. Berard, M. Bidoit, A. Finkel, F. Larouissinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie. Systems and Software Verification. *Springer-Verlag, Berlin, Heidelberg, New York*, 1998.

[4] Edmund M. Clarke, Orna Grumberg and Doron Peled. Model Checking. *The MIT Press, Cambridge, Massachusetts*, 1999.

[5] Antti Valmari. The State Explosion Problem. *Lecture Notes in Computer Science, Vol. 1491: Lectures on Petri Nets I: Basic Models. Springer-Verlag*, 1998.

[6] Christos H. Papadimitriou. Computational Complexity. *Addison Wesley Longman*, 1994.

[7] William Stallings. Operating Systems. *Prentice Hall*, 1995.

[8] Gerald J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering, Vol. 23, No. 5*, 1997.

[9] Gerald J. Holzmann. The Spin Model Checker: Primer and Reference Manual. *Addison-Wesley, Boston*, 2003.

[10] Carl A. Petri. Kommunikation mit Automaten. *Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1*, New York, 1966.

[11] Nissim Francez. Fairness. Springer-Verlag, Berlin, Germany, 1986.

[12] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language, Second Edition. *AT&T Bell Laboratories, Murray Hill, New Jersey*, 1988.

[13] Programming Language C++. *ANSI International Standard 14882:1998*, 1988.

[14] Edsger W. Dijkstra. A Discipline of Programming. *Prentice Hall, Englewood Cliffs, NJ, USA*, 1976.

[15] Charles A. R. Hoare. Communicating Sequential Processes. *Series in Computer Science. Prentice-Hall International*, 1985.

[16] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. *LIAFA, Paris, France.*

[17] GNU Compiler Collection. *http://gcc.gnu.org.*

[18] Sed, a Stream Editor. *http://www.gnu.org.*

[19] M. Luby et. al. Asynchronous Layered Coding (ALC) Protocol Instantiation.*Request for Comments 3450*, 2002.

[20] J. Postel. DoD Standard Internet Protocol.*Request for Comments 760*, 1980.

[21] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification.*Request for Comments 2460*, 1998.

[22] T. Latvala and K. Heljanko. Coping with Strong Fairness. *Fundamenta Informaticae, Vol. 43, pages 175-193, IOS Press*, 2000.

[23] B. Joy, G. Steele, J. Gosling and G. Bracha. Java(TM) Language Specification. *Addison-Wesley Professional, 2 edition*, 2000.