Formal
methods
& Tools

University of Twente
department of
computer science

Design and Implementation of a Systematic State Explorer

A virtual machine based approach

Master's thesis: Michèl A.J. Rosien

FMT-03-2001
March 21, 2001

University of Twente
Department of Computer Science
Division: Formal methods & Tools

Supervisors:
prof.dr. H.Brinksma
dr.ir. J.P.Katoen
ir. T.C. Ruys

Abstract

Model checking is a model-based, automatic technique for the verification of finite state concurrent systems, which systematically checks all the reachable states of a model. An prominent example of model checking tool is SPIN.

This thesis introduces the systematic state exploration tool MAJoR, which explores all reachable states of a model and checks them for errors, similar to SPIN. Unlike SPIN, MAJoR uses a virtual machine, implemented by an interpreter. The code of the high-level language is first translated to code of the intermediate language. This intermediate code is then executed using the interpreter. This thesis discusses the design and implementation of the virtual machine, the interpreter, the high-level language and the intermediate language.

SPIN uses PROMELA as its high-level source language. The source language designed for MAJoR is very close to PROMELA. Two additional features, not present in PROMELA, are added though: handshake communication with more than two processes, and pre- and post-conditions. Both are described in detail in this thesis.

Samenvatting

Model checking is een model-gebaseerde, automatische techniek voor de verificatie van parallelle systemen met een eindig aantal toestanden, die systematisch alle bereikbare toestanden van een model controlleerd op fouten. Een voorbeeld van een vooraanstaand model checking tool is SPIN.

Deze thesis introduceert de systematic state exploration tool MAJoR, dat all bereikbare toestanden van een model bekijkt en controleert op fouten, zoals SPIN. Anders dan bij SPIN, gebruikt MAJoR een virtuele machine, die geimplementeerd is met behulp van een interpreter. De code van de high-level taal wordt eerst vertaald naar code van een tussentaal. Deze tussentaal code wordt dan uitgevoerd door de interpreter. Deze thesis beschouwt het ontwerp en de implementatie van de virtuele machine, de interpreter, de high-level taal en de tussentaal.

SPIN gebruikt PROMELA als high-level taal voor de code van een model. The high-level taal ontworpen voor MAJoR ligt qua syntax erg dicht bij de syntax van PROMELA. Twee nieuwe mogelijkheden, die niet mogelijk zijn in PROMELA, zijn echter toegevoegd: handshake communicatie met meer dan twee processen, en pre- en post-condities. Beide worden uitvoerig behandeld in deze thesis.

Contents

List of Figures

Chapter 1

Introduction

We are getting more and more dependant on embedded systems. Hand computers, mobile phones and high-end television sets are appearing in our everyday life in large numbers. Services such as telebanking and teleshopping have become reality. The development of modern means of transportation uses up 20 percent of the budget, caused by the introduction of ICT-systems. ICT-systems have become a very important aspect of our everyday life. Because we are so dependant on these systems, it is of great importance to the society that they are reliable and function correctly. Apart from good performance in terms of response time and throughput, the most important measure of quality is the absence of troublesome errors. Errors cost money, and may even cost lives. Programming errors in the software of a television are not life threatening but they do have considerable financial consequences for the producer. The error in the pentium chip, in the early 90's, caused the chip manufacturer, Intel, 475 million dollar. Not to mention a lot of embarrassment. More troublesome is when bugs in software cause disasters where human life may be at stake. The fatal blunders in the control software of the Ariane V, aircraft, chemical processes or nuclear power plants are notorious. It was a software error in the computer-controlled cancer therapy machine, Therac-25, that caused patient injury and death, due to severe radiation overdoses.

It is of great importance to find errors in embedded software and hardware. The earlier, the better. The cost to repair a software error in the operational phase is approximately 500 times the cost when the error is found in an earlier development phase. This has lead to the development of improved specification methods for structured design, for example UML and the increased use of version- and configuration-management tools. With complex systems or critical applications, software engineers use so-called formal methods more often. Formal methods can be thought of as the "applied mathematics used to model and analyze ICT systems". International research in this area has lead to the development of promising validation techniques and accessory software tools to detect design- and programming errors in an early stage of development. The approaches can be split up into two categories: the deductive and model-based methods. The first category reduces the correctness of systems to properties of a formal mathematical theory. Tools such as a "theorem prover" or "proof assistants" are then used to attempt to prove these properties. Example of leading tools in this area are PVS, Coq, HOL en Isabelle. Although deductive methods have their successes, for example with smart-card software, the model-based methods have generally more success. The last category will be described in more detail.

As the name suggests, model-based methods use models which describe the possible system behavior with mathematical precision and unambiguity. The models are accompanied by algorithms. These algorithms systematically explore all possible states of the system behavior, using the model. A multitude of validation techniques can be based on this model-based approach. These include a complete exploration of the state space (model checking), experimenting with limited scenarios of the model (simulation) or in reality (testing). Because of continuing improvements of underlying algorithms and data structures, and the increasingly faster and cheaper computers and memory, validation techniques that, 10 years ago, worked on simple examples, can now be run on models within reach.

1.1 Simulation

One of the most well known and most used validation technique is simulation. The tool, the simulator, allows the user to study the system behavior. This is done by calculating, using the model, how the system will respond to specific scenarios (stimuli). Such scenarios, which are used as input to the simulator, can be generated by the user or by tools (such as a random scenario generator). Simulations are often appropriate to get a first impression of the quality of the design (prototype), but are less suited to find subtle errors, because it is impossible to simulate, or even determine, all representative scenarios.

The number of scenarios increases very rapidly. With a mobile phone or remote-control with only a limited number of choices for each step, say five, the number of scenarios with 20 steps is already 5 raised to the power of 20 (approximately 100.000.000.000.000). A lot of time and money is needed to generate and simulate all possible scenarios. That's why, in practice, only a subset of all possible scenarios are tested. However, this introduces the significant risk of leaving very subtle bugs in the system. When only a limited number of scenarios are tested, the reliability of the system is hard to quantify.

1.2 Model checking

Model checking ([17, 8, 9, 15]) is a validation method which tries to systematically find all relevant system states, and with that all possible scenarios, with brute force. This can be used to prove that a given system model satisfies a specific property. The challenge is, by using all means possible, to explore an as large as possible state space. State-of-the-art model checkers, such as SPIN [13, 14], manage state spaces of as much as a billion states. Using smart algorithms, an even larger state space can be reached for specific problems. A problem with 10^{476} states has been reported in literature. Typical properties which are proved using model checking are of a functional, qualitative nature. Is the computed result correct? Can the system end up in a deadlock (for example when two programs wait for each other and freeze the system because of that)? Furthermore, a lot of work has been done to allow the verification of realtime properties. Is the result result available in 20 seconds? Can a deadlock occur within one hour after a reset? Model checking requires that the desired properties are describe precisely and unambiguous. Similarly to the process of creating an accurate system model, this often leads to the detection of all kinds of inconsistencies and errors in the informal documentation. It has been determined that for a part of the ISDN user part protocol, a staggering amount of 55 percent of the original system requirements were inconsistent. With model checking, a model specification is used, for example written in an appropriate dialect of the programming language C or VHDL, from which all possible system states can automatically be derived. The model checker (the tool) systematically checks every states for the desired property. If a state has been found that which does not satisfies the desired property, an counterexample, which contains this state can be generated which shows how an undesired state can be reached. A simulator can then be used to simulate that scenario. The model can then be adjusted accordingly. Model checkers have been successfully applied to a multitude of ICT systems and applications. Deadlocks have been detected in flight-ticket-reservation systems using the internet, problems have been found in e-commerce protocols and a number of studies of international IEEE-standards have lead to significant modifications to the specifications.

1.3 Testing

A model specification is the first thing validation techniques such as simulation and model checking need. This model specification is used to generate all possible system states. However, a well known technique such as testing is also usable in situations where it is hard or even impossible to create a system model. Testing is the process of using a specific behavioral scenario as input for the simulator and to check whether the system behaves accurately. An important distinction can be made about how accessible the internal information of the system that is being tested is. Full access (white box testing), partial access (grey box testing) or no access at all (black box testing). The advantage of testing is that it can almost always be used. Especially on end products rather than models. The disadvantage of testing is almost the same as with simulation. Exhaustive tests are not possible. Similar to simulation, testing can detect the presence of errors, not the absence of errors. Another disadvantage is that it can only be used ”late” in the development trajectory.

A disadvantage of many of the current test methods is that they are often ad-hoc and not systematic. This make the testing process an time consuming, uncontrollable task. Especially the manual generation and maintenance of test cases is a bottleneck. A much more systematic approach is made possible by model based testing. Generation and execution of tests can be done automatically. Test cases are generated from an unambiguous specification similar to the input used for model checking. This form of systematical testing has proved itself in practical situations. Errors have been found in software for information transfer between a TV and an VCR which could not be found using conventional testing methods.

1.4 Goal of the project

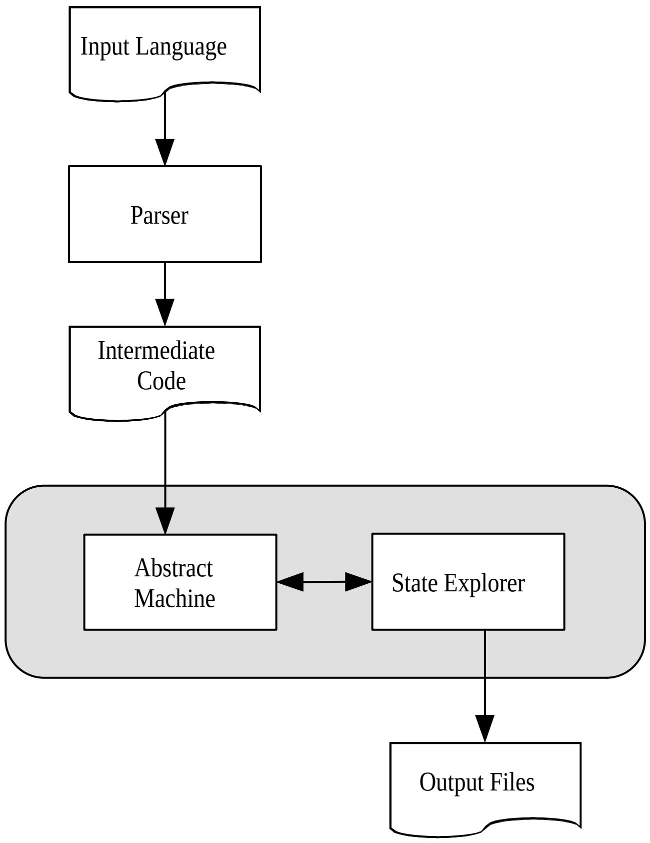


Figure 1.1: *MAJoR*, The systematic state explorer, Components

SPIN, [13, 14, 2] is a state of the art model checking tool. The PROMELA [3] language is used to write the models which are checked by SPIN.

The goal of the project is to design and implement a working prototype, called *MAJoR*, of a systematic state explorer which can explore all states of a program and check those states for errors, similar to SPIN. In addition, an input language, which had to be as close to PROMELA as possible, in which the models have to be written had to be designed, and a parser for that language had to be implemented. The input language provides some additional features which are not provided by PROMELA: Handshake communication using multiple (more than 2) processes and pre- and post-conditions. Both features are described in chapter 2.

Unlike SPIN, MAJoR uses an intermediate language, which had to be designed in the project, which is interpreted and executed by an abstract machine. The input language is first translated to this intermediate language. This approach is based on earlier work on a model checker [12], which also used an intermediate language and an abstract machine.

A visual explanation about how MAJoR works is depicted in figure 1.1. The input file, which contains a model written in the input language, is parsed by the parser module. The output of the parser module, the intermediate code is then used as input for the actual tool. The abstract machine interprets the code and generates states. The state explorer uses the states generated by the abstract machine to explore the complete state space, run a random simulation or run an interactive simulation. The state explorer then returns the result in some output files. The output files are a `.cod` file, which contains the code of the intermediate language and a `.trc` file which contains possible errors and traces.

1.5 Layout of the report

- **Chapter 2** describes the design of the source language and its additional features.

- **Chapter 3** describes the abstract machine and the instructions it supports.

- **Chapter 4** describes the translation of the source language to the intermediate language.

- **Chapter 5** describes how the abstract machine is implemented.

- **Chapter 6** describes the design and implementation of the systematic state explorer.

- **Chapter 7** describes the tool MAJoR. In addition a number of detailed examples of models are presented which are then checked by MAJoR.

- **Chapter 8** presents the conclusions and recommendations, the errors which are still in MAJoR and possible improvement and additions.

Chapter 2

The source language

The source language is the high-level modelling language, in which the models which are to be checked have to be written. It was decided that the language should be close to the PROMELA (see [3]) language. The following PROMELA features are included in the language:

- The basic language structure.
  - expressions & operators.
  - local & global variables & assignment.
  - the `skip` & `goto` commands.
  - an `assert` statement.
- Non-determinism.
  - **if** statements (including **else**).
  - **do** statements (including **else** and **break**).
- Asynchronous communication using channels.
- Multiple processes running in parallel.
  - **proctypes**
  - Multiple instances of the same process definition
  - Interleaving of processes, according to the semantics of SPIN. (proctype)
- An "init" section where all the processes are "started".
- The possibility of using expressions as statements.

The following additional, non PROMELA, features are added to the source language.

- Handshakes (synchronous communication) with more than two participants
- Pre & Post conditions (Extending statements with "before" and "after" conditions)
- "Real" Atomicity (The final implementation just implements PROMELA's `d_step` construct).

2.1 Basic language structure

The source language supports the use of variables. Global variables, which are accessible by all processes, and local variables, which are only accessible by the process in which they were declared. Variables can be assigned to, using the assignment operator ("=").

Like PROMELA, the source language includes expressions and operators. Most C operators are included. Expressions can also be used as statements, in which case the value of the evaluated expression indicates whether that statement is executable(see section 2.8).

The operators that are included are:.

- arithmetic operators: addition(+), subtraction(-), multiplication(*), division(/), remainder(%)
- shift operators: left shift(<<), right shift(>>)
- relational operators: equal to(==), not equal to(!=), less than(<), greater than(>), greater than or equal to(>=), less than or equal to(<=)
- logical operators: logical and(&&), logical or(||)
- bitwise operators: bitwise and(&), bitwise or(|)

To support communication, two additional operators, namely the send(!) and receive(?) operators are included.

The source language also includes 3 simple statements, these are:

- The `skip` statement, which does simply nothing.
- The `goto` statement, which jumps to a specific location in the code.
- The `assert` statement, which is used to test specific conditions at runtime.

2.2 Non-determinism

Like PROMELA, the source language supports non-determinism. This is done exactly the same as in PROMELA, using the so called `if` statements and `do` statements. See "Dijkstra's guarded command language", a language invented by Edsger Dijkstra ca. 1974. It introduced the concept of guards and committed choice nondeterminism (don't care nondeterminism). Described and used in [11]

At any time, a process can have the option of "choosing" which statement to execute next. A choice between three statements is written as seen in figure 2.1. The `IF` and `FI` keywords can be

```
if
:: statement 1
:: statement 2
:: statement 3
fi
```

Figure 2.1: Non-determinism

replaced by `DO` and `OD`. When this is done, the statement is not only a non-deterministic choice, but also a loop. After a choice is executed, an `IF` statement will resume execution at the point *after* the `FI`, while a `DO` statement will resume execution just *before* the `IF`.

Non-determinism introduces two new keywords, namely `else` and `break`. The `break` command can only be used inside a `DO` statement, and will transfer execution to the point just after the `OD`[1].

[1]Together with a 'goto', this is the only way to exit a DO statement

The `else` statement is a convenient shortcut for a statement that is executable(see section 2.8) if and only if all other choices are *not* executable. The `else` may only be used inside an `IF` or `DO`. Note that unlike PROMELA, there are no restrictions when using the `else` statement. In PROMELA, `else` may not be used when a communication statement is used as a possible choice.

2.3 Interleaving

It is possible to run more than one process in parallel, even to run more than one instance of the same process in parallel. To simulate parallelism, processes will interleave. The smallest step a process can take is to execute a single statement[2]. Interleaving means that when a process has executed a statement, *any* available process may then execute the next statement[3].

For example when process *1* can execute the series of instructions: 1, 2, 3, 4 and process *2* can execute the series: A, B, C, D, the following statement sequence could be the result: 1, A, B, 2, C, D, 3. In general, the resulting sequence is *any* combination of statements as long as the statement order *per process* does not change.

Definition 2.1 (Interleaving) *The sequence of statements that are executed is a* random *combination off all the statements of all processes, as long as the ordering of statements from a single process does not change.*

2.4 The Init section

Just like PROMELA, the source language has an `init` section where processes are instantiated and started. But the possibilities of this `init` section are limited compared to PROMELA. While PROMELA allows other statements and variable declarations inside the `init` section, the source language allows only *one*, the `run` statement, which instantiates and starts a process.

Also, unlike PROMELA, the statements in the `init` section are treated as one big atomic section. This implies that *before* other processes can start executing, the `init` section has finished completely.

The reason the `init` section has this constraints is that it will guarantee a fixed size state vector because the size of the state vector will be known at compile time and cannot change during execution of the program. Would the `init` section not be atomic, it could be possible that processes are started when other processes are already running. Since new processes take up space in the state vector, the state vector would dynamically grow. This was found to be too hard to implement, and not really necessary. Dynamic process creation is thus not allowed.

2.5 Communication

Processes need to be able to communicate with each other. Typically this is achieved by using messages which can be send and/or received by processes. Another form of communication is process synchronization, for example one process can not continue until another process has reached a certain point in its code. When processes are synchronizing, they in fact execute the synchronizing statements ate the same time. These two types of communication, message passing and synchronization, can be combined so that processes can synchronize while passing messages to each other.

This section describes the various techniques that can be used to model this kind of communication between processes.

[2]What this means is that a statement will not be divided into its smaller instructions of the abstract machine. Interleaving takes place on statement level, not on instruction level.

[3]Note that this also means that the last process can execute a statement again.

2.5.1 Description

What follows is an informal description of the various communication methods. A more precise and formal description is presented later in this chapter.

Communication using global variables

The easiest form of communication is the use of global variables. A process that wishes to send a certain message to another process will store this message in a global variable (global variables are accessible by all processes) and the receiving process then reads this variable.

One might argue that this is not communication between processes. This is true, but the effect is, usually, the same and, for completeness sake, it is included here. However, care has to be taken when using global variables especially when more than one process can write to this variable, since another process might modify the contents of the global variable before the receiving process has had a chance to read the global variable.

Asynchronous Communication

The second form of communication is asynchronous communication. When processes use this communication scheme, there is always a sender process and a receiver process. The sender process will send a message to the receiver process. What makes this communication asynchronous is the fact that the sender will not wait until the receiver gets the message. In other words, processes will not synchronize on this communication. It is important to note that the sending process does not know which process will receive the message. The first process that is willing and able to receive the message will do so. It might be the case that the message is never received.

Synchronous Communication

Synchronous communication, also called *a handshake* or *rendez-vous*, can only take place when all participants of the communication are ready and willing to communicate. In other words, all participants synchronize on this communication. Thus, unlike asynchronous communication, the sending process will have to wait for the receiver. This communication scheme ensures that the message, eventually, arrives at a receiver. If there is never a process willing to receive the message, the sending process will wait forever (the sending process is in a deadlock[4]). Note that, also in this type of communication, the sending process does not know beforehand which process will be the receiver of the message if more than one process is able to communicate with the sender.

Multiway Synchronous Communication

Until now it was assumed that exactly two processes would participate in a communication, a sending process and a receiving process. But what if multiple, i.e. more than two, processes have to synchronize with each other?. Or a process wishes to broadcast, i.e. send a message to all other processes?

To facilitate more than two participants in a communication a technique called *multiway communication* is introduced, see also: [6]. Multiway communication allows an arbitrary, but fixed, number of processes to participate in a communication.

Multiway communication can be applied to synchronous communication. However multiway communication can get very complicated, especially multiway synchronous communication. So very precise semantics are needed to describe multiway communication. These semantics will be presented later in this chapter.

[4]Unless the process has another choice.

2.5.2 Communication features

The following is a list of all the communication features the language supports.

1. Multiway synchronous communication (handshake) has to be supported.
2. Conditional communication has to be supported. (A communication statement can be extended with an additional boolean expression indicating whether a communication can succeed or not.) In the final implementation this is called the pre and post conditions, as described in section 2.7.
3. The semantics of communication should be as close as possible to the semantics of communication in PROMELA.

All these features will be described in the following sections.

2.5.3 Handshake support in the source language (Design issues)

The high level language provides for specific language constructs that will allow the programmer to use handshakes. There are two parameters that the user needs to supply. The type of the message which will be sent/received and the number of processes participating in the handshake.

It was decided to introduce specific handshake channels to the high level language, much like the channels used in the PROMELA language. In fact, handshake channels are almost the same as PROMELA's channels of length zero (which is the PROMELA way of implementing synchronous communication between two processes). However there is one crucial difference, handshake channels allow more than 2 processes to participate in the communication. The syntax of a handshake declaration is given below.

HS *identifier* = [*expression*] OF {*type*, *type*, ...}

Note that this definition is identical to the normal channel definition of PROMELA (apart from the keyword HS of course). The difference is that the value of the evaluated expression does not denote the length of the channel but the number of processes participating in the handshake. [1]

Processes that wish to communicate using the handshake method must do so using operations on a handshake channel. There are two operators which operate on a handshake channel, a *send* and a *receive* operator. Just as in PROMELA the notation for these operators is '!' and '?'. Hence the syntax of a handshake communication is of the form:

identifier!*expr*, *expr*, ...

or

identifier?*expr*, *expr*, ...

where *identifier* is the name of the corresponding handshake channel and the various expressions denote the message data (expressions that evaluate to a constant, or a variable reference). For now let's assume that the semantics of these operators are the same as the semantics of these operators in PROMELA. Later in this chapter the exact semantics are given.

To allow conditional communication, this notation has to be extended. One could for example use the notation:

identifier?[1] *expr*, *expr*, ... & *bexpr*

where *bexpr* evaluates to false(zero) or true(non-zero). This boolean then indicates whether or not the communication can succeed.

However, a couple of problems arise when using this approach. For example consider the following communication statement:

[1]An array of handshake channels can be created as well, using the same syntax as PROMELA.

[1]The '?' here is just an example, the same notation is valid for the '!' operator.

$aChannel?X \ \& \ X \geq 3$

The process executing this statement wants to receive a value[2] and store that value in variable X. But this communication can only succeed if $X \geq 3$. Which value does the second X refer to? There are two possibilities: it refers to the value of X *before* or *after* the communication has taken place. The latter choice seems the most useful because one can then put additional restrictions on the value that can be received in the communication. In this case this would mean: Only communicate when the value that can be received is greater or equal than three.

So this problem can be solved by letting all variables in the conditional expression refer to values of the variables *after* the communication has taken place. But this still doesn't solve all the problems, consider another, more complicated, example.

Suppose we have a *global* variable X with the current value 2. And, in addition, we have three processes that wish to perform a handshake, and that those processes execute the following communication statements.

1. *aChannel* ! *5*
2. *aChannel* ? $Y \ \& \ X \leq 3$
3. *aChannel* ? $X \ \& \ X \geq 4$

The question is: can these processes communicate with each other using these communication statements?. At first glance one might say yes, since statement 2 can execute because the X variable in the conditional part evaluates to 2, which is less than or equal to 3. And statement 3 can execute because X evaluates to the value received, in this case 5 which is greater or equal to 4. The result of this communication would be that the variables X and Y both contain the value 5.

But, this is not how it works, since it was decided that variables in the conditional part of the communication refer to the values *after* the communication. Thus, statement 2 will not be able to communicate since the value of X refers to the value *after* the communication, which is in this case 5. And $5 \leq 3$ is false.

So, the problem in this case is that the second statement refers to the value of X *before* the communication, and the third statement refers to the value of X *after* the communication.

One could argue that this problem can simply be solved by redefining the semantics. For example: a variable in the conditional part of the communication refers to the value before the communication *unless* the same variable occurs as parameter in the communication. Note that this problem occurs only in *receive* statements, since *send* statements do not modify the contents of a variable.

Implementing these semantics introduces some runtime overhead. Because before a conditional expression can be evaluated, it has to be checked whether a variable in the conditional expression also occurs in the parameter list of the communication. Since an expression can contain more than one variable, this has to be done for each variable in the expression. For every communication we obtain the following, pseudo code.

```
for every communication statement $c$ do
    for every variable $i$ in the conditional expression of statement $c$ do
        for every variable $j$ in the parameter list of statement $c$ do
            if $i = j$ then
                variable $i$ refers to the value after the communication.
            else
                variable $i$ refers to the value before the communication.
```

Luckily, the number of variables in a communication statement is usually not very large. So the runtime overhead of this solution shouldn't pose to great a problem.

[2]The type of the variable and the type of data received have to match of course.

But, there is another, more flexible, solution to this problem. A solution which does not introduce additional runtime overhead but does provide the language with more expressiveness.

The idea is to strictly separate both types of variables, i.e. variables evaluated *before* or *after* the communication. Consider the following modification to the communication syntax:
Instead of

identifier ? *expr*, *expr*, ... & *bexpr*

the following syntax is used[1]:

$\{bexpr_1\}$ *identifier* ? *expr*, *expr*, ... $\{bexpr_2\}$

In the above syntax $bexpr_1$ is evaluated *before* the communication takes place and $bexpr_2$ is evaluated *after* the communication takes place. Thus variables occurring in $bexpr_1$ refer to the before values, and variables occurring in $bexpr_2$ refer to the after values. Note that both conditional expression can just be *true*. When a conditional expression is always true, the appropriate $\{bexpr\}$ can just be omitted.

This new communication syntax solves the problem as described above, since the programmer can exactly specify what he means, or wishes to accomplish. For example the intended effect of the following statements:

1. *aChannel* ! *5*
2. *aChannel* ? Y & $X \leq 3$
3. *aChannel* ? X & $X \geq 4$

is instead typed as follows:

1. *aChannel* ! *5*
2. $[X \leq 3]$ *aChannel* ? Y
3. *aChannel* ? X $[X \geq 4]$

A disadvantage is that the programmer is given additional responsibility in the sense that he has to recognize the fact that the first three statements do not produce the intended effect. The first three statements will not be able to communicate since $X \leq 3$ and $X \geq 4$ can not be true at the same time.

As said before, this solution gives additional expressiveness to the language. Consider the following example:

$[X \leq 3]$ aChannel ? X $[X \geq 5]$

Informaly this means: "Only communicate when the current value of X is less than or equal to 3 *and* only if the value of X after the communication is greater or equal to 5."

This kind of expressiveness was not possible in the other solutions. This new syntax is chosen to be the syntax of communication in the high level language.

However, there are still some cases, with global variables, which might cause problems. Consider the following two communication statements:

1. *aChannel* ! *5, 6*
2. *aChannel* ? *X, X*

[1]The brackets, '{' and '}' are part of the syntax, they do not mean the code in between is optional.

What is this suppose to mean? After the communication variable X is assigned the values 5 and 6? Obviously, this is not possible. One could say the value of X after communication is the last value received. In this case 6. This could easily be implemented (And this is how it is currently implemented). However, it is probably better to produce a runtime error because this kind of communication makes no sense at all. Note that it is not possible to check for these conditions at compile time because at compile time it is not yet known which processes communicate with eachother. The same applies for communications of the form, when X is a global variable:

1. *aChannel* ! *5, 6*
2. *aChannel* ? *X, 6*
3. *aChannel* ? *5, X*

So, whenever a variable can be assigned two (or more) different values after the communication, a runtime error should be produced. This also applies when a variable is assigned the same value more than once. If, for example the 6 in the above example is replaced by a 5 then the variable X will be assigned the value 5 two times. This is not allowed either. *Except* when the statements have a form similar to this:

1. *aChannel* ! *5, 6*
2. *aChannel* ? *X, 6*
3. *aChannel* ? *X, 6*

Here the variable x is assigned to twice. But it is the same value *and* the positions of the variable X in the parameter list is the same. In this case it is allowed and the variable X is assigned the value 5.

1. *aChannel* ! *5, 6*
2. *aChannel* ? *X, 6*
3. *aChannel* ? *5, X*

The reason for this is that is is often the case than one instantiates multiple instances of the same process to be run in parallel. In this case it is very difficult to avoid this problem. So it was decided to allow it only in this case.

There is one thing that has not been considered yet. What happens if there are processes willing to communicate, but not a single process does so using the '!' operator. For example:

1. *aChannel* ? *X*
2. *aChannel* ? *Y*
3. *aChannel* ? *Z*

In this case all processes want to communicate, they all want to receive a value and put that value in a variable. Even if all variables are of the same type, it is not defined what the value of the variables after the communication is. One could say that, in this case, communication can continue (if all variables are of the same type) but no data transfer takes place. In other words: they only synchronize and do not communicate any values. But this makes no sense because this effect can easily be obtained by only using '!' operators.

It is clear that when some process wants to receive a value, there should be at least *one* process that sends that value. This implies the following rule:

In any handshake, there should be at least one *process executing a send ('!') statement.*

If this condition is not met, it is not a valid handshake, and will not be executable.

2.5.4 Formal definition of handshake communication

This section will formally describe how a handshake is performed. First a number of definitions will be presented.

Definition 2.2 *A constant value or an expression that evaluates to a constant value will be referred to as: cexpr. Note that an expression may also contain variables.*

Definition 2.3 *A variable reference, i.e. an identifier which denotes a variable, will be referred to as: var.*

Definition 2.4 *A message component is either a cexpr or a var and is referred to as: mcomp.*

Definition 2.5 *A message consists of an arbitrary but fixed number of message components. A message consisting of N message components is, thus, of the form:*

$$\{mcomp_1, mcomp_2, ..., mcomp_n\}$$

A message will be referred to as M.

Definition 2.6 *The function TYPEOF identifier returns the type of the identifier. Where identifier can be one out of the following list.*

- *TYPEOF cexpr returns the type of the value of the cexpr.*
- *TYPEOF var returns the type of the variable var.*
- *TYPEOF mcomp returns TYPEOF cexpr if mcomp is a cexpr or returns TYPEOF var if mcomp is a var.*
- *TYPEOF M returns the type of the message M. This type is of the form:*

  $\{TYPEOF\ mcomp_1,\ TYPEOF\ mcomp_2,\ ...\ ,\ TYPEOF\ mcomp_n\}$

  This type will be referred to as mtype.

Handshake channel declaration

A handshake channel is declared as follows:

HS *identifier* = [*expr*] OF *mtype*

Where *identifier* is the name of the channel, *expr* evaluates to a constant value and denotes the number of participants. And *mtype* is the type of the messages to be transferred over this channel.

The number of participants on this channel is referred to as: *identifier.count*. The type of the messages that can be transferred using this channel is referred to as: *identifier.mtype*.

Handshake channel communication

Communication over a handshake channel *HS* is of the form:

$\{bexpr_{pre}\}$ HS ! M $\{bexpr_{post}\}$

or

$\{bexpr_{pre}\}$ HS ? M $\{bexpr_{post}\}$

Executability of handshake communication

A communication statement will either succeed or fail. When a communication statement fails, the system can not execute this statement. When the system can not find another statement that it can execute instead, the process blocks on the communication i.e. it will wait until the communication can succeed. A communication statement that can succeed is also called: *enabled.*

A communication statement is enabled if and only if the following conditions are true.

- The $bexpr_{pre}$ condition must hold, before anything can be done.
- TYPEOF *M* must be equal to *HS.mtype.* Note that this can be checked at compile time.
- The process must find exactly $HS.count - 1$ processes that have the possibility to execute a communication statement as well. The communication statements of the other $HS.count - 1$ processes have to be enabled as well. Enabledness of these other statements depends on the enabledness of this statement.
- All $HS.count$ communication statements have to communicate over the same channel *HS.*
- All $HS.count$ communication statements have to be compatible. (a precise description of 'compatible' is given below)
- After the communication the $bexpr_{post}$ condition must hold. Note that the 'effects' of the communication i.e. the transfer of values has already taken place at this time. If after the communication $bexpr_{post}$ does not hold, the system should return to the state it was in before the communication.

Compatibility of messages

Recall that a message *M* was of the form:

$$\{mcomp_1, mcomp_2, ..., mcomp_n\}$$

The following communication statement:

$\{bexpr_{pre}\}$ HS ! *M* $\{bexpr_{post}\}$

can also be thought of as:

$\{bexpr_{pre}\}$ HS *M'* $\{bexpr_{post}\}$

Where *M'* is of the form:

$$\{!mcomp_1, !mcomp_2, ..., !mcomp_n\}$$

Now, let $!mcomp$ be replaced with $mcomp'$ then a message *M'* can be written in the form:

$$\{mcomp'_1, mcomp'_2, ..., mcomp'_n\}$$

Of course the same holds for the '?' operator.

Definition 2.7 *The ith mcomp' of message M' is referred to as M'[i].*

Definition 2.8 *Two messages M'_1 and M'_2 are compatible if and only if:*

$$\forall i, j \in 1..N \bullet (i = j) \Rightarrow M'[i] \textit{ is compatible with } M'[j]$$

Where N is the number of message components in a message.

Definition 2.9 *mcomp' compatibility.*

- *! cexpr is compatible with:*

  - *! cexpr, if and only if both cexpr evaluate to the same value.*
  - *! var, if and only if cexpr and var evaluate to the same value.*
  - *? cexpr, if and only if both cexpr evaluate to the same value.*
  - *? var, if and only if TYPEOF var equals TYPEOF cexpr.*

- *! var is compatible with:*
  - *! var, if and only if both var evaluate to the same value.*
  - *? cexpr, if and only if var and cexpr evaluate to the same value.*
  - *? var, if and only if both TYPEOF var are the same.*

- *? cexpr is compatible with:*
  - *? cexpr, if and only if both cexpr evaluate to the same value.*
  - *? var, if and only if TYPEOF cexpr equals TYPEOF var.*

- *? var is compatible with:*
  - *? var, if and only if both TYPEOF var are the same.*

Definition 2.10

$$mcomp'_1 \textit{ is compatible with } mcomp'_2 \Longleftrightarrow mcomp'_2 \textit{ is compatible with } mcomp'_1.$$

Definition 2.11 *The messages: $M'_1, M'_2, ..., M'_N$ are compatible if and only if:*

$$\forall i, j \in 1..N \bullet (i \neq j) \Rightarrow M'_i \textit{ is compatible with } M'_j$$

$$\wedge$$

$$\forall j \in 1..P \bullet (\exists i \in 1..N \bullet M'_i[j] \textit{ is of the form: !mcomp})$$

Where N is the number of processes participating in the communication and P the number of mcomp in a message.

Informally this means: All communication statements should be compatible with each other *and* there should at least be one message containing only *send* components. In other words: There should always be one process executing a handshake *send* statement.

Data transfer in communication

Definition 2.12 *The boolean expression: $mcomp' = !$ is true if and only if $mcomp'$ is of the form: !mcomp. The same definition holds were both '!' are replaced by '?'.*

Definition 2.13 *The boolean expression: $mcomp' = var$ is true if the $mcomp'$ is a var.*

Definition 2.14 *$mcomp'_1 \gg mcomp'_2$ means: The evaluated value of $mcomp'_1$ is copied to the variable $mcomp'_2$.*

Definition 2.15 *Communication is performed as follows:*

$$\forall k \in 1..P \bullet (\forall i, j \in 1..N \bullet$$

$$(M'_i[k] = \ ! \ \wedge M'_j[k] = \ ? \ \wedge M'_j[k] = \ var\ ) \Rightarrow M'_i[k] \gg M'_j[k])$$

Where N is the number of processes participating in the communication and P the number of mcomp in a message.

2.5.5 Asynchronous communication support in the source language

Asynchronous communication is included in the source language. The language provides mechanisms to send and receive from channels. The semantics are kept as close as possible to the semantics of PROMELA's asynchronous communication. A channel for this kind of communication is declared as follows:

CHAN *identifier* = [*expression*] OF {*type*, *type*, ...}

The *identifier* is the name of the channel, the *expression* is the *capacity* of the channel, i.e how many elements can be stored in the channel. Finally the types between the braces denote the type of the elements in the channel. See also definition 2.5.

Definition 2.16 *The capacity of a channel is the amount of elements that can be stored in the channel.*

The communication syntax for sending to, and receiving from such a channel is exactly the same as for handshake channels. See section: 2.5.3.

The *!* operator is used to put a value into the channel, while the *?* operator is used to remove a value from the channel. A value that is removed from a channel may then be stored in a variable. See the following examples.

1. $chan\_id!2, x$
2. $chan\_id?3, y$

The first example *sends* the value 2 and the value of x to the channel *chan_id*. The second example removes the value 3 from that channel and another value which is then stored in the variable y.

Sending to a channel is only allowed when that channel is not full. Receiving from a channel is only allowed when that channel is not empty.

Definition 2.17 *An asynchronous channel is full when the number of elements in the channel equals the channel's capacity.*

Definition 2.18 *An asynchronous channel is empty when the number of elements in the channel equals 0.*

The channel behaves like a FIFO queue. The first value inserted, is also the first value to be removed. If a process sends to a non-empty channel, the newly inserted element is queued. When a process receives from a non-empty channel, it receives the *oldest* element from the channel.

When an asynchronous communication attempt fails, i.e. sending to a full channel or receiving from an empty channel, the communication is blocked, or, not enabled. The communication is also blocked when the parameters are incompatible.

2.6 Atomicity

Sometimes it is desirable that certain groups of statements will be executed atomic. This means that it is guaranteed that these statements will be executed *before* an other process can execute a statement.

Definition 2.19 *Groups of statements that are executed atomic, will be executed before other processes can execute a statement. If an atomic statement has started, it has to finish before any other process can continue.*

atomic {
  statement$_1$
  statement$_2$
  statement$_n$
}

Figure 2.2: Example of an atomic statement.

The source language supports this. Groups of statements can be made *atomic* by enclosing them in an atomic statement, as show in figure 2.2. Because all statements in an atomic statement are treated as *one* statement, it has to be defined when an atomic statement is executable. One could argue that an atomic statement is always executable, and it simply always starts executing the first statement. But what if there is a statement inside the atomic that is not executable? Since no other process can execute a statement before the atomic statement has finished, and the atomic statement can never finish, a deadlock has occurred.

The system cannot detect whether a statement in the middle of an atomic is executable or not, since that statement hasn't been tried yet. But it *can* try the first statement of the atomic statement. This yields the following definition:

Definition 2.20 *An atomic statement is executable if and only if the first statement of that atomic statement is executable. If any other statement than the first is not executable, an atomic deadlock occurs.*

2.7 Pre and Post conditions

In section 2.5.3, it was discussed how a handshake statement could be surrounded by two conditional statements. One statement that has to be true *before* the handshake executes, and one that has to be true *after* the handshake has executed. If one of these conditional statements is false, the whole handshake is enabled. These two conditional statements are called the *pre* and *post* conditions.

This idea of pre and post conditions can be extended to not only allow them with handshakes, but with an arbitrary statement. This leads to the following definition.

Definition 2.21 *Any statement in the source language can have a pre- and/or post-condition. When a statement has a pre- and/or post-condition,* both *conditions have to be true before that statement is executable.*

The syntax for a statement becomes:

[**{**pre_condition**}**] statement [**{**post_condition**}**]

Where the brackets mean that the conditions are optional.

The most obvious use for the pre and post conditions is with communication (Handshake and Asynchronous). One could program statements like: "Only receive this value if that value is greater than 3". This cannot, or not as easily, be programmed without the use of a post-condition.

Another, less obvious, use, is to make it easy to program the *if* statement as known in other program languages. The statement:

if expression **then** statement **else** block

Can be programmed as:

{pre_condition} statement

Programming an *if*-like statement like this will not increase the amount of statements in the code and thus the number of states will also not increase, since a pre-condition is part of the statement. The standard PROMELA solution, as shown below, however will always introduce new states and thus a bigger state space.

pre_condition ; statement

or

pre_condition -> statement

The two solutions both need two states, one after the condition and one after the statement. The pre-condition solution only needs 1 state.

2.8 Executability

Sometimes, a certain statement can not be executed. Such a statement is called: *blocked* or *not enabled* or *not executable*. The opposite is called: *enabled* or *executable*[5]. When a statement is blocked, the process trying to execute the statement is blocked too, *unless* the process has other *enabled* choices available (see section 2.2). When there *are* other choices, but those are all blocked, the process will block also.

When a process is blocked, it is blocked until a statement becomes executable again. Note that this is only possible when *another* process can make the blocking statement executable again. When another process *never* unblocks a blocking statement, that statement will block forever. Thus making it impossible for that process to finish it's code. This is an error, an undesirable situation.

When not a single process can execute a statement, in other words *all* statements in *all* processes are blocked, a `deadlock` has occurred. This is an undesirable situation also.

What determines whether a statement is executable? This is explained below.

Skip A `skip` statement *always* executable.

Goto A `goto` statement is *always* executable.

Assert An `assert` statement is *always* executable.

If An `if` statement is executable, if and only if, there is at least on choice in the if statement that is executable.

Do A `do` statement is executable, if and only if. there is at least on choice in the do statement that is executable.

Atomic See definition 2.20.

Handshake communication See section 2.5.4.

Asynchronous communication An asynchronous communication statement is blocked when trying to send to a full channel, or receive from an empty channel, or when the communication parameters are incompatible. See also section2.5.5.

An expression used as statement Such an expression is enabled when the expression evaluates to a non-zero value i.e. true.

Run A `run` statement is always executable.

[5]All of these terms will be used throughout the report.

Chapter 3

The abstract machine

The abstract machine is the heart of the system. Its purpose is to interpret the instructions of the intermediate language and to generate (new) states. In fact, the abstract machine does only *one* thing, it generates the *first possible* state (see section 3.1 for a definition of `state`) to which the system can go, starting from the current state. This state is also called the `next` state. Later on, the next state will be used by the model-checker.

A change from one state to another state is called a *transition* (see section 3.2). There is often more than one possible transition and not all of these transitions are always enabled. That is why the abstract machine will take the *first possible* transition.

Definition 3.1 (Purpose of the abstract machine) *The abstract machine generates the first possible next state* and *the transition it took to get there.*

3.1 State representation

The `state` of the system is the information needed to uniquely determine in what situation the system currently is. This information, when stored in memory, is also called the *state vector*.

What information should be in the state vector? It's obvious, that the contents of the memory should be in it. In other words the values of all global variables and the values of all local variables of all processes. In addition, the state vector should contain the instruction pointers of each process.

Definition 3.2 (State vector) *The state vector contains the following elements:*

- *The values of all global variables (including channels).*
- *For each process, the values of all local variables.*
- *The instruction pointers of all processes.*

The memory representation of the state vector is shown in figure 3.1. This figure shows the state vector representation for n processes. The total size of the state vector is m bytes.

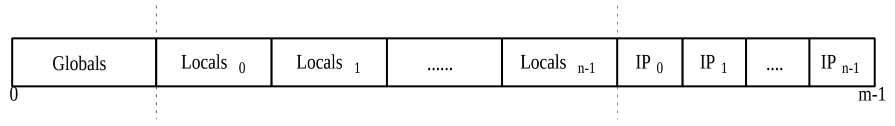


Figure 3.1: State vector representation.

```
proctype p1() {              proctype p2() {
    if                           a = 1;
    :: skip;                 };
    :: a = 2;
    fi;
};
```

Figure 3.2: Example of a non-deterministic transition

```
proctype p1() {
    if
    :: skip;
    :: if
       :: a = 1;
       :: a = 2;
       fi;
    fi;
};
```

Figure 3.3: Example of a `nested if` transition

3.2 Transition representation

A transition is a change from one state to another state, which occurs when a statement in the source language is executed. When a transition occurs, the information that defines that transition has to be stored so that the system knows exactly which transition has taken place. This information is later used, amongst other things, to determine which transition should be tried next.

Definition 3.3 (Transition) *From the current state, the next state can be calculated using only the information stored in the transition.*

What information is needed to be able to calculate the next state? Since the source language allows multiple processes to execute in parallel, the process id should be included to know which process was executing a statement. The process id alone is not enough because it is possible that one process can randomly choose which statement it will execute, i.e. *non-determinism*. Thus, the choice the process took has to be included as well. Consider figure 3.2; In this situation, process *p1* could execute the statement *a = 2*. This means that from the current state, process p1 executes choice 2. The transition is the tuple: (1, 2). When process *p2* executes the statement *a = 1* however, the transition is: (2), since process *p2* did not have a choice.

However, this transition storage scheme is not sufficient to cover all possibilities. There are two features of the source language that require additional attention, namely: *nested - if clauses* and *handshake communication*. Figure 3.3 is an example of a nested if clause. When this process executes the statement *a = 1*, what will the transition information be? It cannot be (1, 2), since the second choice is the *if* statement, *not* the statement *a = 1*. To cover this, the transition information will be: (1, [2, 1]). This indicates that process *p1* chooses the second choice and then the first choice.

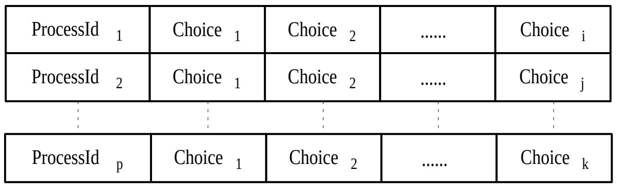
To facilitate handshake communication, one has to note that when a handshake takes place, *more than one* process will execute a statement. Consider figure 3.4; In this example, *a* is a handshake channel. At this moment process *p1* and *p2* can execute a handshake, namely: *a!2* and *a?2*. Since more than one process takes a transition here, and the syntax of *one* transition is already defined, it makes sense to store this handshake transition as follows: [(1, [2]), (2, [2, 2])].

Definition 3.4 (Transition format) *A transition is defined by [(process_id, [choice_id])]*

```
proctype p1() {          proctype p2() {
    if                       if
    :: skip;                 :: skip;
    :: a!2;                  :: if
    fi;                         :: skip;
};                              :: a?2;
                                fi;
                             fi;
                         };
```

Figure 3.4: Example of a handshake transition

Note that when a process can not make a non-deterministic choice, the choice list is simply an empty list. Also, when no handshake communication takes place, the transition list has only *one* element. Figure 3.5 shows a representation of how a transition can be stored in memory.



ProcessId $_1$	Choice $_1$	Choice $_2$	......	Choice $_i$
ProcessId $_2$	Choice $_1$	Choice $_2$	......	Choice $_j$
⋮	⋮	⋮	⋮	⋮
ProcessId $_p$	Choice $_1$	Choice $_2$	......	Choice $_k$

Figure 3.5: Transition representation.

3.3 Components of the abstract machine

This section will describe the components of the abstract machine in detail. Figure 3.6 shows a simplified representation of the abstract machine. One can identify two main modules, the *interpreter* (described in subsection 3.3.3) and the *next state generator*(described in subsection 3.3.4). These two modules interact with each other and the *system data structures* (described in section 3.3.1).

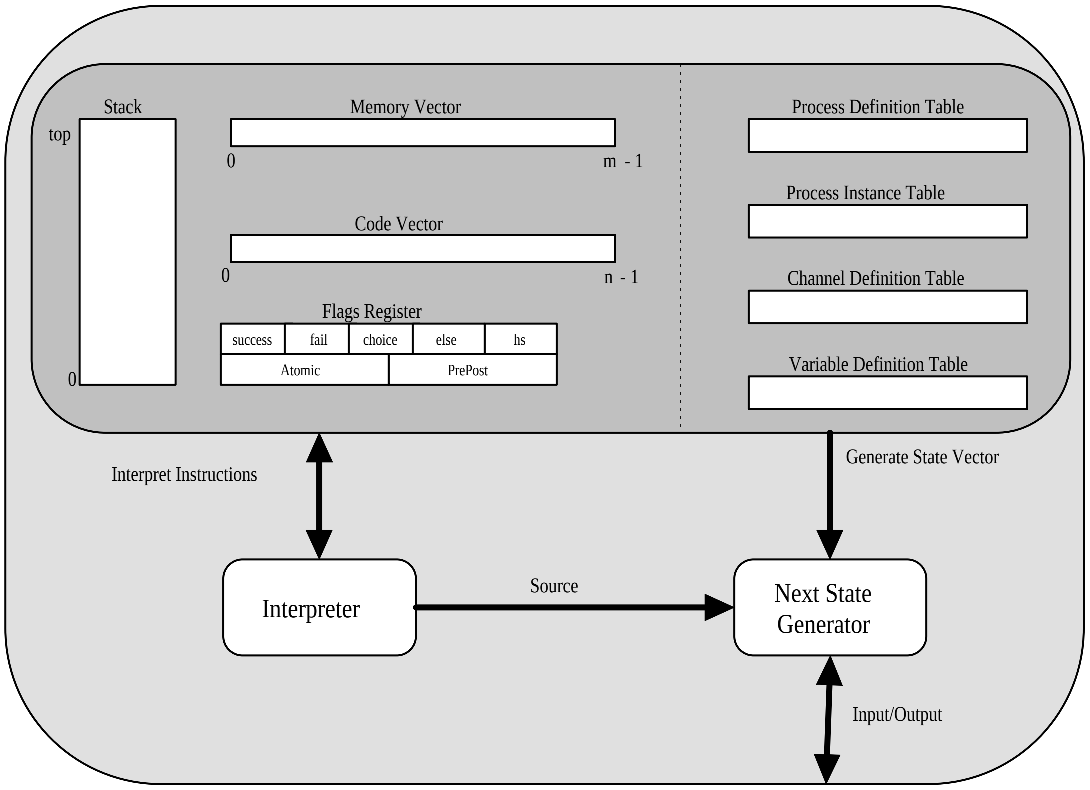


Figure 3.6: Overview of the abstract machine.

3.3.1 System data structures

The abstract machine needs the following data structures to be able to interpret instructions:

- The main memory, or the `memory vector`
- A place to store the instructions, or the `code vector`
- A stack to evaluate expressions and to store intermediate data
- A flags register

This section will explain these data structures in more detail.

The code vector

The `code vector` is the place where all the instructions are stored[1]. There are two possible implementations of the code vector:

[1]Also, some instructions have additional source code information as explained in section 3.3.3.

1. One code vector for *each* process instantiation.

2. One code vector for *all* processes.

Both implementations work, but the first implementation has a drawback. Since it is possible that a process is instantiated more than once, in other words, two or more copies of a process are running, the first option will use more memory. The reason for this is that, in case of the first option, two *identical* pieces of code have to be kept in memory. The second option does not have this problem because process instantiations can *share* each others code. Thus the `code vector` of the second option will require less memory. This is the reason why the second option is chosen.

The memory vector

The memory vector holds the main system memory. The only information stored in it, is the contents of variables[2]. One can identify two kinds of variables, *global* variables and *local* variables. Currently, only integer variables (int) are implemented. Obviously, separate instantiations of processes can *not* share their local variables as they could with their code. So the local variables of *each instantiation* have to be stored seperately.

Definition 3.5 (Contents of the memory vector) *The memory vector stores all global variables and all local variables of each process instantiation.*

The flags register

The flags register stores all the system flags. One can identify two different kinds of flags, flags that can only be *on* or *off* and flags that have a *count* associated with them.

On/Off flags

Success flag The interpreter sets this flag when an instruction is encountered which signals the end of a statement. In other words, the success flag is set when a complete statement from the input language is successfully interpreted.

Fail flag The interpreter sets this flag when it has decided that the current statement is not executable. Certain instructions always block while others block on a certain condition.

Choice flag The interpreter sets this flag when it detects that a choice between different statements can be made [3]. This flag is of importance for the `executable` function. See section 5.1.2.

Else flag When the interpreter detects that the choice it is about to make is an else choice, it sets this flag[4]. The `executable` function can then take special action to deal with the else. Special action is needed because an else choice is only executable when all the other choices are not.

Hs flag This flag is set by the interpreter when the system is ready to handshake. This means that a possible handshake has been found i.e. compatible channels and enough partners. The `executable` function can then try to execute the handshake to check whether it is executable or not.

[2]Channels are variables too. Handshake channels are variables too, but they are 0 bytes in size so they are not stored.

[3]There is only one instruction which sets this flag, the CHOICES instruction.

[4]Only the ELSE instruction sets this flag.

Count flags

Atomic flag The interpreter increases the value of the atomic flag when the interpreter goes in *atomic* mode. When it leaves *atomic* mode, the value of the atomic flag is decreased.

PrePost flag This flag functions the same as the `atomic` flag. But this flag is modified when the system enters or leaves *prepost mode*.

The stack

The stack is used to store intermediate results from expressions, and to store other intermediate data. Most instructions that transfer data, do this via the stack. As with the code vector, two approaches concerning the stack can be identified.

1. One stack for *each* process instantiation.

2. One stack for *all* processes.

Again, both solutions probably work but in this implementation the second alternative is chosen. The reasoning behind this is, that instructions are grouped together to form a statement from the input language. Since these statements are atomic i.e. the instructions of different statements can not interleave, it follows that the stack before and after the execution of a statement is the same. Thus there is no need for a separate stack for each process.

.

3.3.2 System tables

The abstract machine uses a number of tables to keep important information in memory (see figure 3.6). This section will describe each table in more detail.

Process definition table

A process definition contains the following information (see figure 3.7):

- A unique identifier of the process definition, `id`.

- The size of the memory for the local variables of this process (MemSize).

- The address in the code vector that indicates the first instruction to be executed by the process (Start IP)

- The name of the process, as used in the input language.

- A variable definition for all variables of the process(see section 3.3.2).

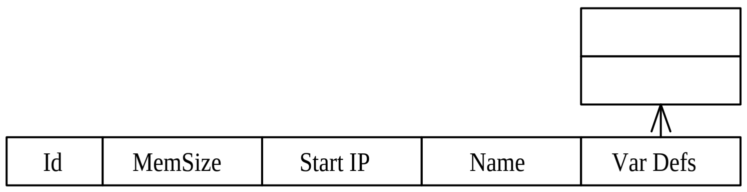


Figure 3.7: Information contained in a process definition

Process instance table

As more than one instance can be created from a specific process definition, additional information is needed to distinguish the process instances. Also, the process instance stores the information that belongs to a specific instance. The following information is included in a `process instance` record (see figure 3.8):

- a unique process instance identifier, an `id`.
- The current position in the code vector, the `instruction pointer` or `IP`
- The address of the *first* local variable, to access its local variables in the memory vector. (`memory start`).
- A field containing status information, for example whether the process instance is halted or running.
- A pointer to the process definition, for example, to determine the name or required memory size.

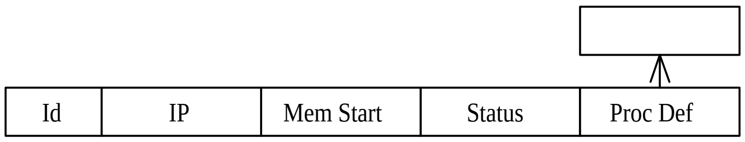


Figure 3.8: Information kept for a process instance

Channel definition table

Channels are an extended kind of variables in the sense that they need additional information to describe them. This additional information is stored in a channel definition:

- The address in the memory vector the channel information starts and ends, the `start addr` and `stop addr` (points just *after* the last memory address belonging to this channel).
- The size of one element in the channel, the `element size`.
- The maximum number of elements that the channel can handle. The `element count`.
- The channel type, indicating whether the channel is asynchronous or a handshake.
- The name of the channel.

Start Addr	Element Size	Element Count	Type	Name	Stop Addr

Figure 3.9: Channel definition

Variable definition table

Variable definitions have been included to allow the system to make a mapping from memory addresses to variables used in the input language. A variable is a simple record containing the following elements:

- The memory address of the start of the variable

- The memory address of the end of the variable[5]
- The name of the variable.

Start Addr	Stop Addr	Name

Figure 3.10: Variable definition

3.3.3 The interpreter

The interpreter module interprets an instruction stored in the code vector and update the memory vector and stack accordingly[6]. See section 3.6 for a definition of the instructions. In addition, certain flags in the flags register can be set or cleared[7].

The interpreter module is, very often, invoked by the next state generator module in order to generate the next state. Since it is desirable that it is also known which statement in the input language caused the transition to the next state, the interpreter will also send source code information to the next state generator. That information can then be returned to the model checker for further use.

Source code information

To be able to send the source code information to the next state generator, instructions in the code vector can have additional data. This data stores the source code information. Because the next state generator is only interested in *complete* statements from the input language, and such a statement typically consists of more than one instruction in the intermediate language, it has to be decided what source code information to be stored in which instruction. The easiest way to do this is to store the complete statement in the *last* instruction from that statement. So, for example, the instruction `a = 2 + 4;` in the source language is translated to the code shown in figure 3.11. In this example only the `STORE` instruction, which is the last instruction of the

```
LDGADDR 0       ; Push the address of variable 'a' on the stack
PUSH 2
PUSH 4
ADD
STORE           ; Write to variable 'a'
```

Figure 3.11: Possible translation of `a = 2 + 4`

statement, will have source code information.

What information should be in the source code information? Of course the exact statement should be included. Since there is more than one process, it is desirable to include the exact name of the process. Finally, to make it easier to find the statement in the original source code the line number and column number are included.

Definition 3.6 (Source code information) *Source code information consists of the following elements:*

- *The statement from the source code.*
- *The name of the process executing a statement.*
- *The line number and column number of the statement.*

[5]Only differs from the start address in case of a channel.
[6]Only if the instruction modifies these ofcourse.
[7]Incremented or decremented in case of the atomic/prepost flags.

3.3.4 The next state generator

Generating the first possible next state from the current state is the *only* function of the abstract machine. This function is realized by the *next state generator* (In the future, the *next state generator* will be referred to as the *next function*). In order to generate the next state, the next function needs two input values, which it normally receives from the model checker. These two values are the *current state*[8], for which the next function needs to generate the next state, and the *last transition*.

Definition 3.7 (Last transition) *The last transition is the previous successful transition from* the current *state.* [9]

The *last transition* is used to determine which transitions the next function should skip i.e. not generate.

Definition 3.8 (Next state) *The next function will generate the first possible next transition* after *the* last transition.

When the next state is successfully generated, the next function will return three output values to the 'caller', the model checker. These output values are the following:

- The next state.
- The transition to get to this state, the *next transition*
- The source code of the high-level language statement that caused this transition.

The functioning of the next function is described in detail in chapter 5

3.4 Asynchronous communication channels

Asynchronous communication channels are implemented using a queue, which is mapped onto the main memory vector. The information stored in this queue consists of the values of all elements in the channel plus a count which denotes the number of elements in the queue. An element consists of the number of parameters used in the communication. See figure 3.12. Empty places

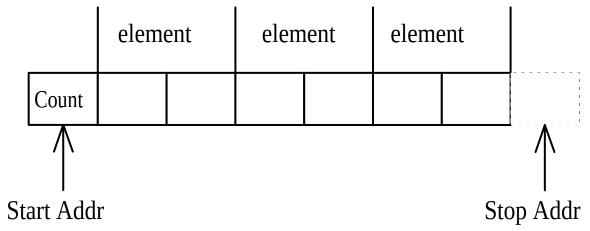


Figure 3.12: Channel queue for elements with two components with capacity three.

in the queue are filled with zeros. When an element is added to the queue it is added to the right of existing elements. When an element is removed, the left most element is removed and the remaining elements are shifted to the left. Note that it is not possible to use a cyclic queue to optimize speed. When using a cyclic queue, it is possible that two queues with the same contents will generate a different state vector. This is not allowed.

[8]When the current state is not supplied, the next function uses the state it is in now.
[9]Note that the last transition is not defined in the initial state of the system.

3.5 Global variables of the abstract machine

The abstract machine uses the following *global* variables:

- `current_channel`, the id of the communication channel that is selected for communication.
- `current_process`, the id of the process that is currently executing instructions.
- `handshake_initiator`, the id of the processes that initiates a handshake.
- `handshake_buffer`, a buffer used to temporarily store data, transferred during a handshake.
- `current_selected_choice`, when non-determinism is possible, the variable denotes which choice the interpreter is currently trying. This variable equals 1 when no non-determinism is possible.
- `choice_count`, When non-determinism is possible, the variable denotes the maximum number of choices are available. This variable equals 1 when no non-determinism is possible.

3.6 Instruction set

Before the instructions, supported by the interpreter, can be explained, a couple of definitions will be introduced.

Definition 3.9 (push *x*) *The term "push* x*" will be used to describe the effect of putting the value* x *on top of the stack.*

Definition 3.10 (pop *x*) *The term "pop* x*" will be used to describe the effect of removing the top value* x *from the stack. That value will be stored in variable* x.

Definition 3.11 *The term "mem[*n*]" will be used to denote the value stored at memory address* n.

3.6.1 Jump instructions

JMP

Syntax:	**JMP** *address*
Description:	Jumps current process to address *address*. Internal jump.
Effect:	$current\_process.IP = address$
Flags affected:	None

GOTO

Syntax:	**GOTO** *address*
Description:	The current process jumps to address *address* and sets the `SUCCESS` flag.
Effect:	$current\_process.IP = address$ *set* `SUCCESS` flag
Flags affected:	`SUCCESS` flag is *set*

3.6.2 Stack instructions

PUSH

Syntax:	**PUSH** x
Description:	Push the value x on top of the stack.
Effect:	`push` x
Flags affected:	none.

LDGADDR

Syntax:	**LDGADDR** x
Description:	Push the value x on top of the stack. x is an address of a global variable.
Effect:	`push` x
Flags affected:	none.

LDGVAR

Syntax:	**LDGVAR** x
Description:	Push the value at memory address x, on top of the stack. x is an address of a global variable.
Effect:	`push` `mem`$[x]$
Flags affected:	none.

LDLADDR

Syntax:	**LDLADDR** x
Description:	Converts the local address x to a global address in memory and puts the converted value on top of the stack. x is an address of a local variable.
Effect:	`push` $(x + current\_process.MemStart)$
Flags affected:	none.

LDLVAR

Syntax:	**LDLVAR** x
Description:	Converts the local address x to a global address in memory and puts the value at that memory address on top of the stack. x is an address of a local variable.
Effect:	`push` `mem`$[(x + current\_process.MemStart)]$
Flags affected:	none.

XCHG

Syntax:	**XCHG**
Description:	Exchanges the top two values on the stack
Effect:	`pop` b; `pop` a; `push` b; `push` a
Flags affected:	none.

STORE

Syntax:	**STORE**
Description:	Move data from stack in memory
Effect:	`pop` b; `pop` a; `mem`$[a] = b$
Flags affected:	`SUCCESS` flag is *set*.

EVAL

Syntax:	**EVAL**
Description:	Removes the top value from the stack. Sets the `SUCCESS` flag if this value is non-zero. Sets the `FAIL` flag otherwise.
Effect:	`pop` a;
Flags affected:	`SUCCESS` flag is *set* if $a! = 0$. `FAIL` flag is *set* if $a == 0$.

3.6.3 Flag instructions

CLRA

Syntax:	**CLRA**
Description:	Decreases atomic flag.
Effect:	`ATOMIC` flag is decreased
Flags affected:	`ATOMIC` flag is decreased

SETA

Syntax:	**SETA**
Description:	Increases atomic flag.
Effect:	`ATOMIC` flag is increased
Flags affected:	`ATOMIC` flag is increased

CLRPP

Syntax:	**CLRPP**
Description:	Decreases pre-post flag.
Effect:	`PREPOST` flag is decreased
Flags affected:	`PREPOST` flag is decreased

SETPP

Syntax:	**SETPP**
Description:	Increases pre-post flag.
Effect:	`PREPOST` flag is increased
Flags affected:	`PREPOST` flag is increased

3.6.4 Non-determinism instructions

CHOICES

Syntax:	**CHOICES** *count*
Description:	Initiate non-determinism with *count* choices. Jumps to the address of the currently selected choice.
Effect:	**let** $choice\_count = count$ $current\_process.IP = current\_process.IP + current\_selected\_choice$
Flags affected:	None

CHOICE

Syntax:	**CHOICE** *address*
Description:	Signals the interpreter that the address of this choice is at address *address* and transfers control to that address.
Effect:	$current\_process.IP = address$ *set* `CHOICE` flag
Flags affected:	`CHOICE` is *set*

ELSE

Syntax:	**ELSE** *address*
Description:	Signals the interpreter that the address of this choice is at address *address* and transfers control to that address. Also indicates that this is an *else* choice.

Effect:
```
$current\_process.IP = address$
set CHOICE flag
set ELSE flag
```

Flag affected: `CHOICE` and `ELSE` flags are *set*

3.6.5 Communication instructions

HS

Syntax:	**HS** *action*, *memory_mode*, *channel*
Parameters:	*action*: `RECV` or `SEND` *memory_mode*: `LOC` or `GLOB` *channel*: channel Id
Description:	Try to initiate handshake communication. Fails if handshake communication is not possible.

Effect:
```
if $action == RECV$ then
    set FAIL flag
else begin
    if $memory\_mode == LOC$ then
        let $current\_channel = mem[(channel + current\_process.MemStart)]$
    else
        let $current\_channel = channel$
    let partners = number of processes that are able to execute a HS.
    if not $partners = channel.ElementCount$ then
        set FAIL flag
    else
        let $handshake\_initiator = current\_process$
end
```

Flags affected: See: Effect

CHAN

Syntax:	**CHAN** *action*, *memory_mode*, *channel*
Parameters:	See **HS**
Description:	Try to initiate asynchronous communication. Fails if that is not possible.

Effect:
```
if $memory\_mode == LOC$ then
    let $current\_channel = mem[(channel + current\_process.MemStart)]$
else
    let $current\_channel = channel$
if $action == SEND$ and $channel.full()$ then
    set FAIL flag
if $action == RECV$ and $channel.empty()$ then
    set FAIL flag
```

Flags affected: See: Effect

RECEIVE

Syntax:	**RECEIVE**
Description:	Pushes the values which are going to be received on the stack.

Effect:

```
if $current\_channel.type == CHAN$ then
begin
    remove element $e$ from $current\_channel$.
    for all values v in element e do push $v$
end
if $current\_channel.type == HS$ then
    for all values v in handshake_buffer do push $v$
```

Flags affected: None.

SEND

Syntax:	**SEND**
Description:	Removes the values which are about to be send from the stack. Fails if incompatible communication parameters are detected.

Effect:

```
if $current\_channel.Type == CHAN$ then
begin
    for $i = 1$ to $current\_channel.ElementSize$ do pop $v_i$
    Add element consisting of values $v_1$ through $v_{current\_channel.ElementSize}$ to
        $current\_channel$
    set SUCCESS flag
end
if $current\_channel.Type == HS$ then
    if $current\_process == handshake\_initiator$ then begin
        for $i = 1$ to $current\_channel.ElementSize$ do begin
            pop $v_i$
            store $v_i$ in $handshake\_buffer$
        end
        set HS flag
    end
    else
        for $i = 1$ to $current\_channel.ElementSize$ do begin
            pop $v_i$
            compare popped $v_i$ with $v_i$ in $handshake\_buffer$
        end
        if $\forall i \in \{1 \ldots current\_channel.ElementSize\} \bullet$
            popped $v_i == v_i$ in $handshake\_buffer$ then
                set SUCCESS flag
            else
                set FAIL flag
```

Flags affected: See: Effect

3.6.6 Mathematical instructions

Name	Description	Effect
ADD	Addition	`POP` B; `POP` A; `PUSH` A + B
SUB	Subtraction	`POP` B; `POP` A; `PUSH` A - B
MUL	Multiplication	`POP` B; `POP` A; `PUSH` A * B
DIV	Division	`POP` B; `POP` A; `PUSH` A / B
MOD	Remainder	`POP` B; `POP` A; `PUSH` A % B
INC	Increment	`POP` A; `PUSH` A + 1
DEC	Decrement	`POP` A; `PUSH` A - 1
NEG	Negate	`POP` A; `PUSH` -A
BAND	Bitwise AND	`POP` B; `POP` A; `PUSH` A & B
BNOT	Bitwise NOT	`POP` A; `PUSH` ~A
BOR	Bitwise OR	`POP` B; `POP` A; `PUSH` A | B
BXOR	Bitwise XOR	`POP` B; `POP` A; `PUSH` A ^ B
LAND	Logical AND	`POP` B; `POP` A; `PUSH` A && B
LOR	Logical OR	`POP` B; `POP` A; `PUSH` A || B
LNOT	Logical NOT	`POP` A; `PUSH` !A
SHR	Shift right	`POP` B; `POP` A; `PUSH` A >> B
SHL	Shift left	`POP` B; `POP` A; `PUSH` A << B

3.6.7 Relational instructions

Name	Description	Effect
EQ	Equal	`POP` B; `POP` A; `PUSH` A == B
NE	Not equal	`POP` B; `POP` A; `PUSH` A != B
GT	Greater than	`POP` B; `POP` A; `PUSH` A > B
GE	Greater than or equal	`POP` B; `POP` A; `PUSH` A >= B
LT	Less than	`POP` B; `POP` A; `PUSH` A < B
LE	Less than or equal	`POP` B; `POP` A; `PUSH` A <= B

3.6.8 Miscellaneous instructions

ASRT

Syntax:	**ASRT**
Description:	Remove value from the stack and throws an assertion failed exception if that value is zero
Effect:	`pop` x **if** $x == 0$ **then** assertion failed
Flags affected:	`SUCCESS` is *set*

NOOP

Syntax:	**NOOP**
Description:	Does nothing
Effect:	None
Flags affected:	`SUCCESS` is *set*

HALT

Syntax:	**HALT**
Description:	Stops the current process
Effect:	$current\_process.Status = HALTED$ $current\_process.IP = -1$
Flags affected:	`SUCCESS` is *set*

RUN

Syntax:	**RUN** *process_id*
Description:	Creates and starts a new process instance from process definition *process_id*. The parameters for this process are on the stack. Control is transferred to the beginning of the new process.
Effect:	current_process = new process instance created from process definition *process_id* current_process.IP = start of new process.
Flags affected:	None

IRET

Syntax:	**IRET**
Description:	Transfers control back to the init section.
Effect:	current_process = 0 (the init section)
Flags affected:	`SUCCESS` flag is *set*

Chapter 4

Translation

The source language is translated to intermediate code before it is interpreted by the interpreter of the abstract machine. This chapter describes how the source language is translated into the intermediate language. (For more details, see also: Appendix B and C)

4.1 Expressions

Expressions consist of data (constant numbers or variables), and operators that perform certain computations on that data and return some result.

Operators

All operators in the intermediate language take zero parameters. Instead, they get(pop) their parameters from the stack. The result of the computation of that specific operator is put(pushed) on the stack, where it can processed further.

For example, the **ADD** instruction removes 2 values from the stack, adds them, and puts the result of the addition back on the stack. Figure 4.1 shows how this is done for the expression 3 + 5. First 3 and 5 are pushed on the stack, then the **ADD** instruction is executed and the result,

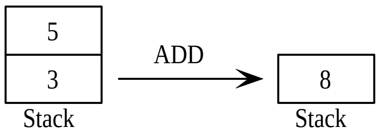


Figure 4.1: **ADD** instruction example, 3 + 5

8, is pushed on the stack.

Because the result of this expression is put on the stack, it can easily be reused as subexpression for another expression. For example 2 * (3 + 5). (See figure 4.2)

Note that 3 + 5 is enclosed in parentheses to indicate that this subexpression should be evaluated *before* it is multiplied with 2. Would the parentheses be omitted, the result would be 2 * 3 + 5 = 6 + 5 = 11, since multiplication has a higher precedence than addition. This example requires that an intermediate result (2 * 3) is evaluated before all the data (2,3,5) is on the stack. This is shown in figure 4.3. For a description of all operators, refer to appendix B.1 for the mathematical operators and appendix B.2 for relational operators.

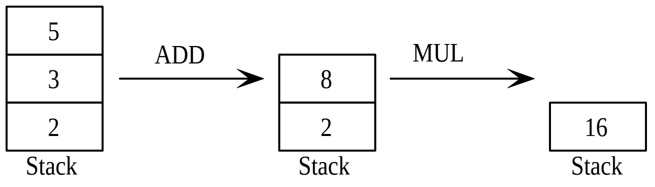


Figure 4.2: **ADD** instruction example, 2 * (3 + 5)

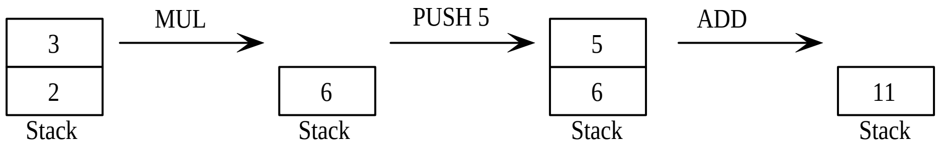


Figure 4.3: **ADD** instruction example, 2 * 3 + 5

Data

There are two kinds of data to be used in expressions, constants and variables. Operators expect this data to be on the stack. To put a constant value on the stack, the **PUSH** instruction is used. This instruction needs one parameter and simply pushes the value of this parameter on the stack.

Variables are different. Sometimes the *value* of the variable is needed, to be used in an expression. And sometimes the *address* of the variable is needed, when a value has to be assigned to a variable for example.

There are two types of variables, global variables and local variables. When the address of a local variable is needed, the local variable address has to be converted to a global address before it is pushed on the stack. This is needed because there may be more than one variable with the same *local* address. For example, when one process with a single local variable i is instantiated twice, both instance variables have the same *local* address. But of course these are separate variables and are stored at different physical addresses in memory (the global address). Thus, when an instruction wants to assign a value to a local variable, the global address of that variable has to be computed first.

There are four possible cases with variables. For each case a specialized instruction exists. These are shown in figure 4.4.

Push(Load) the address *addr* of a global variable	**LDGADDR** *addr*
Push the value of a global variable	**LDGVAR** *addr*
Push the address of a local variable	**LDLADDR** *addr*
Push the value of a local variable	**LDLVAR** *addr*

Figure 4.4: Variable / Stack instructions

The conversion of the address of a local variable to its global address is done automatically by the **LDLADDR** and **LDLVAR** instructions.

4.2 Assignment & Evaluation (*STORE* & *EVAL*)

There are two typical uses of an expression. It can be assigned to a variable, for example $a = 2 * b$. Or it can be used as a condition. The latter is the case, for example, when expressions are used as statements or as part of pre- and post-conditions.

Assignment

Assignment amounts to simply evaluating an expression and storing its value in a variable. The instruction used for assignment is the **STORE** instruction. This instruction pops two values from the stack, the data to be stored and the address where this data has to be stored. Also, this instruction sets the `SUCCESS` flag, indicating the successfull end of a statement, an assignment. The code for an assignment of the form:

<variable> = *<expression>*

is shown in figure 4.5.

```
LDGADDR or LDLADDR address    ; Push address of variable
<expression>                  ; Evaluate the expression
STORE                         ; Perform the assignment
```

Figure 4.5: Translation of an assignment

For example the assignment: a = 4 * b + 5, where *a* and *b* are global variables, is shown in figure 4.6.

```
LDGADDR  0   ; Push address of a
PUSH     4
LDGVAR   1   ; Push value of b
MUL          ; Evaluate 4 * b
PUSH     5
ADD          ; Evaluate 4 * b + 5
STORE        ; Perform the assignment
```

Figure 4.6: Translation of assignment: a = 4 * b + 5

There is one additional instruction which is very convenient when dealing with assignments, the **XCHG** instruction. This instruction swaps the top two values of the stack. This is needed when the top two values of the stack are in the wrong order for a correct assignment.

Evaluation

Sometimes an expression has to be evaluated, but the result of the expression does not have to be stored somewhere. For example when an expression is used as a statement, or in a pre- or post-condition. The instruction responsible for this is the **EVAL** instruction. This instruction removes one value from the stack and sets the `SUCCES` or `FAIL` flag according to this value. If the value is 0, the `FAIL` flag is set, otherwise the `SUCCESS` flag is set. The translation of an evaluation is simply: evaluate the expression, then **EVAL**.

4.3 Atomicity (*SETA* & *CLRA*)

The translation of an atomic statement is simple. The instructions that have to be executed atomically are enclosed by the **SETA** and **CLRA** instructions. **SETA** increases the atomic counter and signals the interpreter that from now on all instructions are atomic. In other words, the interpreter is now in atomic mode. The interpreter stays in atomic mode until the atomic counter is 0. The atomic counter can be decreased with the **CLRA** instruction.

The reason that it is an atomic *counter* instead of a *flag* is that it is possible that multiple **SETA** instructions are encountered before a **CLRA** instruction is encountered, i.e. by nesting of atomic statements.

4.4 Pre & Post conditions (*SETPP* & *CLRPP*)

A statement that has a pre- and/or post condition is only executable when both the pre and post conditions evaluate to a non-negative value. In other words, the `SUCCESS` flag(the flag indicating a statement is enabled) can only be set when:

- The pre-condition is true.
- The statement is executable.
- The post condition is true.

The instructions for a pre-condition are interpreted *before* the instructions of the statement, while the instructions for a post-condition are interpreted *after* the instructions of the statement.

The interpreter must somehow know that certain instructions belong to a pre- or post-condition. When a pre-condition is interpreted, the `SUCCESS` flag may not be set since the statement hasn't been interpreted yet. When the statement is interpreted, the `SUCCESS` flag may also not be set since there *might* be a post-condition which has to be interpreted too.

This problem is solved as follows. When a statement has a pre- and/or post-condition, the *complete* statement, including the pre and post is enclosed by the **SETPP** and **CLRPP** instructions. Both instructions function the same as the atomic instructions **SETA** and **CLRA** in the sense that the prepost flag is also a counter instead of a flag. The **SETPP** instruction signals the interpreter that it is entering pre and post mode. This means that the following statement has *at least* a pre- or a post-condition. The `SUCCESS` flag will only be set when the interpreter is not in pre and post mode. The **CLRPP** instruction is used to "leave" pre and post mode.

The translation of the following pre- and post-statement, assuming `a`, is the only global variable(at global address 0), is shown in figure 4.7. Note that **EVAL** is used to evaluate the pre- and post-conditions.

$$\{a > 4\}\ a = a + 1\ \{a < 6\}$$

```
SETPP           ; enter pre-post mode
LDGVAR    0
PUSH      4
GT
EVAL            ; evaluate a > 4
LDGADDR   0
LDGVAR    0
PUSH      1
ADD             ; evaluate a + 1
STORE           ; a = a + 1
LDGVAR    0
PUSH      6
LT
EVAL            ; evaluate a < 6
CLRPP           ; leave pre-post mode
```

Figure 4.7: Translation of $\{a > 4\}\ a = a + 1\ \{a < 6\}$

4.5 Non-determinism

Non-determinism in the intermediate language is realized by the **CHOICES**, **CHOICE** and **ELSE** instructions.

4.5.1 The *CHOICES* instruction

The **CHOICES** instruction signals the interpreter that it can randomly choose which instructions to interpret next. In other words, using the **CHOICES** instruction, non-determinism can be realized. This instruction has one parameter, the number of choices the interpreter can take. This instruction has to be followed by this number of **CHOICE** instructions, where the last **CHOICE** instruction could also be an **ELSE** instruction. The **CHOICES** instruction determines which **CHOICE** to execute. For example, when the interpreter wants to execute the second choice, the **CHOICES** instruction immediately jumps to the second **CHOICE** instruction, skipping the first one.

4.5.2 The *CHOICE* and *ELSE* instructions

The **CHOICE** and **ELSE** instructions both have one parameter, the *address* of the first instruction belonging to the code of this particular choice. When executed, these instructions simply jump to this address and execution is resumed at this address. The difference between **CHOICE** and **ELSE** is which flags are set when they are interpreted. Both instructions set the `CHOICE` flag, indicating that a non-deterministic choice is in progress. The **ELSE** instruction also sets the `ELSE` flag, indicating that this particular choice is an *else* choice, which has a different interpretation.

4.5.3 Translation

This section describes how non-determinism in the source language is translated into instructions in the intermediate language (see also: Appendix C). Two sources of non-determinism in the source language can be identified, `if` statements and `do` statements. Translation of both statements is basically the same, except the action that is taken *after* a choice has been executed.

An `if` statement will jump to the location *after* the code for the last choice for that `if` statement. A `do` statement will jump back to the start of the statement. A `break` statement inside a `do` statement is translated as a jump to location after the code for the `do` statement (as the jump with an `if` statement). The general translation of non-determinism is depicted in figure 4.8.

```
Start:  CHOICES            Choice_Count
        CHOICE             Address 1
        CHOICE             Address 2
        ...
        CHOICE/ELSE        Address n
1:      <Code for choice>
        JMP                Start(do) or End(if)
2:      <Code for choice>
        JMP                Start(do) or End(if)
        ...
n:      <Code for choice>
        JMP                Start(do) or End(if)
End:    ...
```

Figure 4.8: Translation of non-determinism

An example of the translation of a typical `if` statement is shown in figure 4.9. The **JMP** instructions that jump to the end of the `if` can be replaced by jumps to the **CHOICES** instruction (the start of the `if` statement) , in case this was a `do` statement instead of an `if` statement.

```
if
:: a = a + 1;
:: skip;
:: a = 5 * 6;
fi;
```

(a is the only, global, variable)

is translated to:

```
0   CHOICES   3   ; 3 alternatives
1   CHOICE    4
2   CHOICE    10
3   CHOICE    12
4   LDGADDR   0   ; start first alternative
5   LDGVAR    0
6   PUSH      1
7   ADD           ; evaluate a + 1
8   STORE         ; a = a + 1
9   JMP       18  ; jump to end of if
10  NOOP          ; skip (second alternative)
11  JMP       18  ; jump to end of if
12  LDGADDR   0   ; start third alternative
13  PUSH      5
14  PUSH      6
15  MUL           ; evaluate 5 * 6
16  STORE
17  JMP       18  ; jump to end of if
18  ...
```

Figure 4.9: Translation of an example `if` statement

4.6 Communication

Communication in the intermediate language is realized by the **HS** (HandShake) and **CHAN** (CHANnel, asynchronous) instructions, in combination with the **SEND** and **RECEIVE** instructions.

4.6.1 The *HS* and *CHAN* instructions

HS and **CHAN** are used to initialize the communication. In addition they offer the opportunity to abort a communication attempt as soon as possible, without the need to evaluate possible communication parameters and compare them afterwards.

Both instructions need three parameters:

1. The *Communication Type*. This can be *SEND*, or *RECV* (receive).
2. The *Channel Type*. This can be *LOC* (local) or *GLOB* (global). This parameter is needed because local channel variables are treated differently than global channel variables.
3. The *Channel Id*, to identify the channel which is to be used in the communication.

HS

When the interpreter encounters an **HS** instruction, the following actions take place:

1. If the first parameter is *RECV*, the instruction immediately fails and the *FAIL* flag is set. Handshake receive statements are *always* not executable, they can only be executed in combination with a handshake send. If the first parameter is *SEND*, the interpreter proceeds to step 2.
2. Determine the communication channel. If the second parameter is *GLOB*, the communication channel is simply the value of the third parameter. If the second parameter is *LOC*, the value of the third parameter indicates the *local address* of the local channel variable. The value of this variable is a *pointer* to a *global* channel Id. Local communication channels are

not allowed, because local communication makes no sense. The whole purpose of communication is to send information from one process to another. A process cannot communicate with itself. When a communication channel is declared as a local variable, it is in fact a pointer to an existing channel, which can be assigned to if needed.

This allows communication channels to be passed to processes as arguments, since arguments are simply local variables. Also, using local channel variables, a process can change its communication channel at runtime by simply assigning another value to this variable.

3. Determine whether enough partners are available for the handshake. When the interpreter cannot find enough potential partners for this handshake, the **HS** instruction fails and the *FAIL* flag is set.

When all these steps have been executed successfully, the interpreter continues interpreting. It does *not* set the *SUCCESS* flag.

CHAN

When the interpreter encounters a **CHAN** instruction, the following actions take place:

1. First, the communication channel is determined in exactly the same way as explained in step 2 from the **HS** section.
2. If the first parameter is `SEND`, the interpreter checks whether the communication channel is full. The **CHAN** instruction fails if this is the case.
3. If the first parameter is `RECV`, the interpreter checks whether the communication channel is empty. The **CHAN** instruction fails if this is the case.

4.6.2 The *SEND* and *RECEIVE* instructions

SEND and **RECEIVE** are used to perform the actual communication and data transfer, after the initial communication stage is complete(**HS** and **CHAN** didn't fail).

SEND

When the interpreter encounters an **SEND** instruction, the following actions take place:

1. Determine the type of the current communication channel (**HS** or **CHAN**).
2. Step 2 depends on the type of the current communication:

   HS: Depends on whether this instruction is the *handshake initiator* or not.

   Initiator: n values are expected on the stack, where n is the number of components in the message that is being sent. Those values are removed from the stack and stored in an internal handshake buffer, where they can be accessed by the handshake partners. The `HS` flag is set.

   Non-Initiator: n values are expected on the stack, where n is the number of components in the message that is being sent. Those values are removed from the stack and *compared* to all the values in the internal handshake buffer (value 1 is compared to the first value in the buffer, value 2 is compared to the second value in the buffer, etc). If all the values equal the values in the handshake buffer, this communication is successful and the `SUCCESS` flag is set, otherwise the communication parameters were not compatible and the `FAIL` flag is set.

   CHAN: The **SEND** instruction expects n values on the stack, where n is the number of components in the message that is being sent. Those values are removed from the stack and stored at the appropriate place in the main memory (the channel queue). Communication is complete and the `SUCCESS` flag is set.

RECEIVE

When the interpreter encounters a **RECEIVE** instruction, the following actions take place:

1. Determine the type of the current communication channel (**HS** or **CHAN**).
2. Step 2 depends on the type of the current communication:

   HS: All values in the handshake buffer are pushed on the stack. The `SUCCESS` or `FLAG` are *not* set. These flags are set later, when the values have been successfully compared and/or assigned to variables.

   CHAN: *n* values from the channel queue in main memory are pushed on the stack, where *n* is the number of components in the message that is being received. The `SUCCESS` or `FAIL` flags are *not* set. These flags are set later, when the values have been successfully compared and/or assigned to variables.

4.6.3 Translation

This section describes how communication in the source language is translated into instructions in the intermediate language (see also: Appendix C). The translation for handshake communication and asynchronous communication is the same, except for the **HS** and **CHAN** instructions of course. Sending and receiving are translated differently however. The translations in this section assume a handshake(**HS**) instruction. When an asynchronous communication is needed, simply replace all **HS** instructions with **CHAN** instructions.

Sending

A *send* statement in the source language is typically of the form:

<variable> ! <expression>, <expression>, ...

For example:

A ! 2, 5 + B

This example is translated as depicted in figure 4.10 (assuming B is the only global variable and channel A is the only global channel):

```
HS       SEND  GLOB  0   ; Initiate the communication, channel id 0
PUSH     2               ; First component on the stack
PUSH     5
LDGVAR   0               ; Push variable B, global address 0
ADD                      ; Second component on the stack
SEND                     ; Send!
```

Figure 4.10: Translation of the `send` statement: A ! 2, 5 + B

This translation scheme is simple: initiate, push components on the stack, send.

Receiving

A *receive* statement in the source language is typically of the form:

<variable> ? <expression>, <expression>, ...

for example:

A ? 5 + B, C

```
HS          RECV   GLOB   0   ; Initiate the communication, channel id 0
RECEIVE                       ; Receiving...
SETPP
PUSH        5                 ; Pushing expression 5 + B
LDGVAR      0
ADD                           ; Evaluate 5 + B.
EQ                            ; Compare
EVAL                          ; Equal? Fail or Success for this component
LDGADDR     1                 ; Pushing variable
XCHG
STORE                         ; Store the communication data. Success for this component
CLRPP
```

Figure 4.11: Translation of a typical `receive` statement: A ? 5 + B, C

This is translated as depicted in figure 4.11 (assuming B and C are the only global variables and channel A is the only channel): As seen in figure 4.11, the **RECEIVE** instruction expects additional instructions to compare and/or store the communication data. These additional instructions determine whether a receive is successful. When data is compared, the **EVAL** instruction determines the outcome. When data is stored in a variable the outcome is always a success, which is indicated by the **STORE** instruction that always sets the `SUCCESS` flag.

Since all these different components can set the `SUCCESS` flag, the interpreter might think that the statement has successfully ended after it has compared the first component. To ensure all components are compared and/or stored, the instructions responsible for that are enclosed in a **SETPP** and **CLRPP** pair.

The general translation for a receive statement is thus of the form: Initiate, Receive, **SETPP**, compare and/or store data, **CLRPP**.

4.7 Process creation (*RUN* and *IRET* instructions)

The source language has an init section where all the processes are instantiated and started. The intermediate language has a special instruction to start processes. This is the **RUN** instruction. The **RUN** instruction takes a process definition ID as its only parameter and creates and starts a new process instance of that definition.

A problem with process creation is how process parameters are handled. Parameters to processes should be passed to the processes using the stack. It makes sense to push the parameters onto the stack *before* the **RUN** instruction is executed. Since process parameters are nothing more than local variables for that process, which are initialized upon process creation, the parameters have to be stored in the local variable space somehow.

It was decided that the code responsible for the storage of process parameters, should be a part of the process code, *not* of the init section code. A problem that arises when using this approach is that the execution needs to switch from the init section to the process and back, because the init section has to be completed before the "normal" process code may start.

The solution is to have the parameter storage code at the beginning of each process' code. The parameter storage code is then ended with a special instruction which transfers control back to the init section. This special instruction is the **IRET** instruction. To ensure that the init section is completed before another process starts executing, the init section is enclosed in **SETA** and **CLRA** instructions.

For example consider the process definition and instantiation as shown in figure 4.12.

Here, a process is declared that takes two integers as parameters. The translation of the process code and init section code is shown in figures 4.13 and 4.14. Note that the order in which the parameters are pushed on the stack in the init section is reversed.

```
proctype aProcess(int a; int b) {          init {
    a = b + 5;                                 run aProcess(2, 8);
};                                         }
```

Figure 4.12: Process creation/instantiation

```
LDLADDR  0
XCHG
STORE       ; Store first parameter in local variable
LDLADDR  1
XCHG
STORE       ; Store second parameter in local variable
IRET        ; Return to the init section
LDLADDR  0  ; Start body code
LDLVAR   1
PUSH     5
ADD
STORE       ; a = b + 5
HALT
```

Figure 4.13: Translation of a process with parameters

The general translation of the process creation/instantiation is as follows. First, all parameters for the process are pushed onto the stack. Then the **RUN** instruction is executed, which temporarily transfers control to the process code. The first instructions of the process code then remove the parameters from the stack and store them in local variables. Finally, the **IRET** instruction is executed which transfers control back to the init section.

```
SETA
PUSH  8
PUSH  2
RUN   1  ; process with id 1 (0 = init section)
CLRA
HALT
```

Figure 4.14: Translation of the init section for a process with parameters

4.8 Miscelaneous instructions

A couple of instructions do not fit into any sections above, these are:

- **ASRT**, used to translate an `assert` statement. **ASRT** removes a value from the stack. And signals an *assertion failed* when this value is zero. Always sets the `SUCCESS` flag.
- **GOTO**, goto is used to translate a goto statement. It takes one parameter, the address to jump to. It always sets the `SUCCESS` flag[1].
- **NOOP**, does nothing. Is used to translate the `skip` statement.
- **HALT**, signals the interpreter that the current process has reached the end of its code and should be stopped.

[1]unlike the **JMP** instruction which never sets the `SUCCESS` flag

Chapter 5

Implementation of the abstract machine

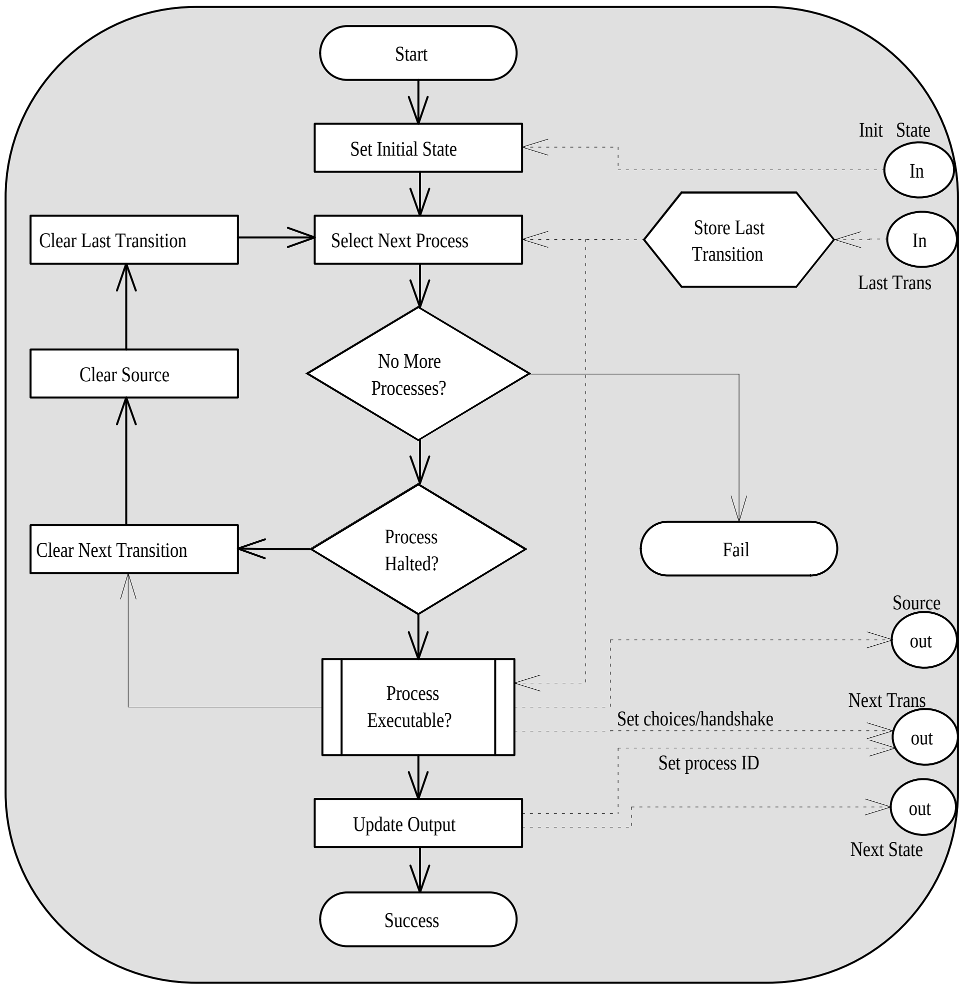


Figure 5.1: Overview of the *next* subroutine.

5.1 The NEXT function

The *next function* simply iterates over all processes, starting with the process that is instantiated first, until it finds a process that is executable. How to determine whether a process is executable? This is is realized by the program block *process executable?* as indicated in figure 5.1. This is a very complex procedure that is described in more detail below.

5.1.1 Determining whether a process is executable

The *executable* function, called by the *next function*, is responsible for deciding whether a process is executable or not. To explain *how* to determine whether a process is executable it is necessary to define *when* a process is considered to be executable. Since a process can be non-deterministic, i.e. it is possible for a process to randomly choose which statement to execute next, we let a process be executable when *at least one* of the possible choices in the process is executable.

Definition 5.1 (Executable) *A process is executable when* at least one *of the statements the process could execute is executable.*

The *next function*, the caller of the *executable function*, is only interested in the first possible next state. Hence the *executable* function only searches for the *first* executable statement too. An overview of the *executable* function is shown in figure 5.2.

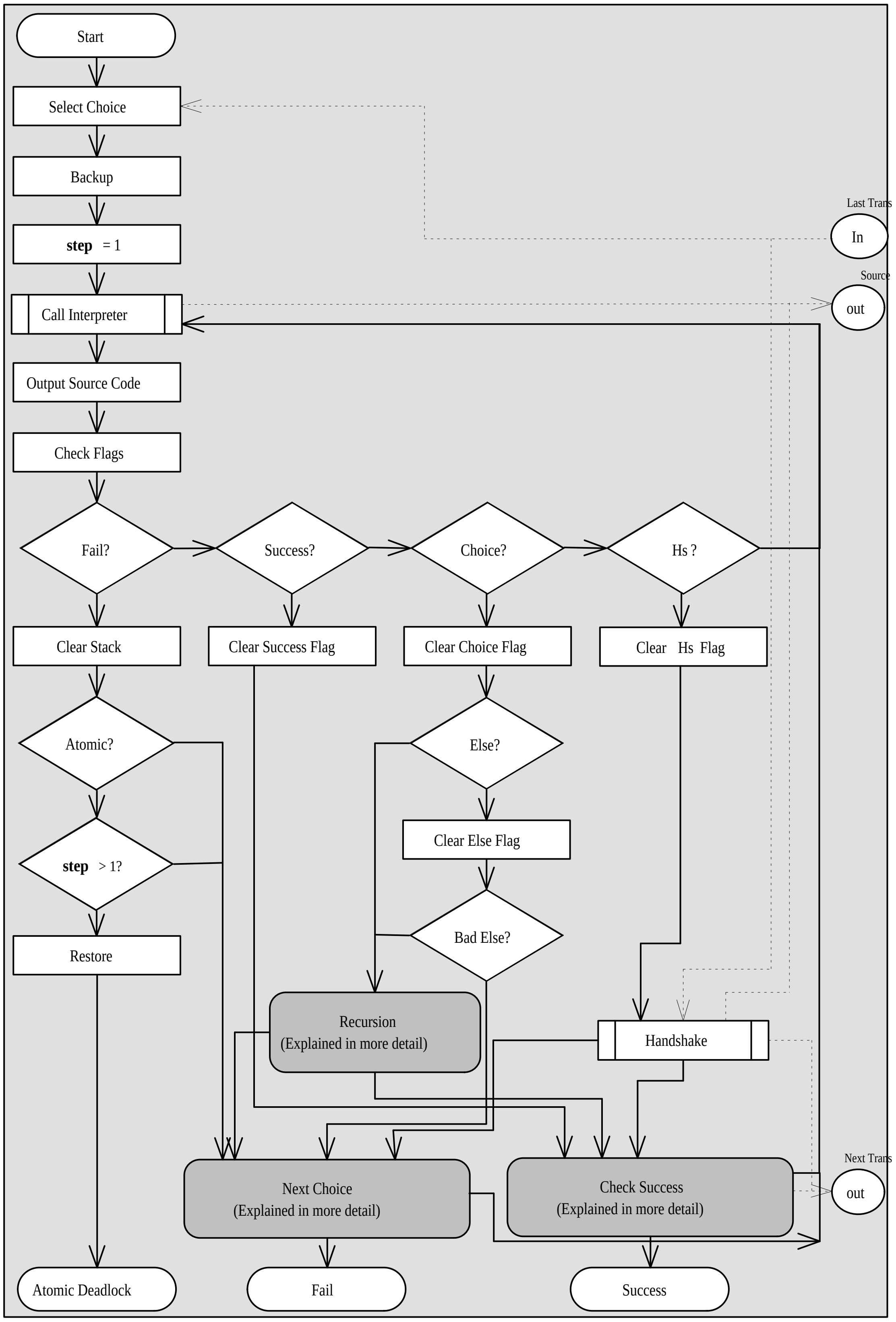


Figure 5.2: Overview of the *executable* subroutine.

The *executable* function first backs up the current state[1] after which it repeatedly calls the interpreter until one of the flags in the flags register is set. (see section 3.3.1 for a definition of the system flags) When a flag is set the following situations can occur:

Fail Flag After the `fail` flag is set, the system checks whether the atomic flag is also set[2]. And if this is not the first statement in the atomic. If both of these conditions are true, an atomic deadlock is detected. Recall that the decision whether an atomic statement is executable, only depends on the *first* statement in the atomic clause. Any other statement that blocks in the atomic will cause an atomic deadlock. If one of the conditions is false the system will simply detect a blocked statement and will try to find another choice.

Success Flag When the `success` flag is set, the system has found an executable choice. After this has been detected, it has to be determined whether this is a next transition or not. This is done in the `Check Success` program block. This will be explained in more detail later on.

Choice Flag The `choice` flag indicates that the interpreter is at a point where it can make a choice between different statements. In case of an *else*, which is also a choice, it has to be determined whether the *else* is executable. This is only the case if and only if *all* the previous choices were unsuccessful. The `executable` function will then call itself recursively. This will be explained in more detail later.

Handshake Flag (Hs) The `Hs` flag indicates that a handshake is in progress. Executing a handshake is a very complicated procedure and is explained in detail in section 5.2

Trying another choice

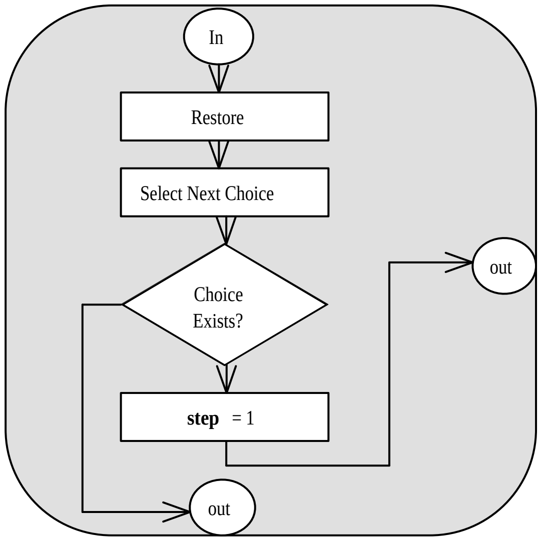


Figure 5.3: *Next Choice* in more detail.

Trying another choice is straightforward (see figure 5.3). First, the *exact* state *before* the last choice was tried is restored. Then the next choice is selected. If no such choice exist, the `executable` function returns false, indicating that no successful choice was found, hence the process is not executable. Otherwise the next executable statement is returned.

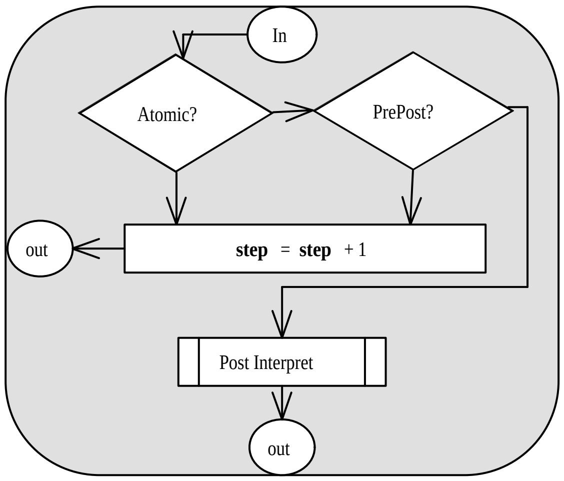


Figure 5.4: *Check Success* in more detail.

Determining whether a successful transition has been found

After the interpreter has set the `success` flag, the conclusion that a statement is executable can not yet be made. The system can still be in an *atomic* state or in a *prepost* state. The `executable` function takes the actions as shown in figure 5.4. If the system is still in one of the aforementioned states, the system continues interpreting, else the current statement is considered executable. After the *post instructions* have been interpreted the *executable* function stops and returns the executable transition. The process is executable.

Post instructions Sometimes, after a successful transition has been found, the interpreter needs to interpret some additional instructions which do not belong to any statement in the source code. These instructions are for example the `HALT` instruction and the `JMP` instruction. The `JMP` instruction is, amongst other things, found at the end of an `if statement` to jump over the other choices. This instruction does not belong to any statement, but *has* to be executed.

Definition 5.2 (Post instruction) *A post instruction is an instruction that does not belong to any statement, but* has *to be executed when it is encountered.*

5.1.2 Handling non-determinism

Non-determinism, or choice, is implemented using a recursive call of the `executable` function when the `choice` flag is set. The reason why recursion is used for non-determinism is the recursive definition of an `if statement`[3].

An `if statement` is executable if and only if *at least one* of its choices is executable.

Also, since the source language allows nesting of `if` statements to an arbitrary level (see figure 5.5), the recursive structure becomes obvious. Consider figure 5.5, it is *one* `if statement` but to determine whether this statement is executable the system has to determine for *each* sub-`if statement` whether it is executable. In other words, `executable` calls itself, thus the most elegant way to handle non-determinism is by using recursion.

Before the recursive call can be made, it is necessary to backup certain (global) information since the recursive call can modify this. Examples of such values are as the *current active choice* and the *current number of choices.* Figure 5.6 shows how the recursive call takes place.

[1]Including the flags register and the *process halted* bit
[2]In case of the atomic flag *set* means: > 0
[3]The same holds of course for a `do statement`

```
if
:: if
   :: if
      :: etc, etc..
      :: ...
      fi;
   :: ...
   fi;
:: ...
fi;
```

Figure 5.5: Nesting of *if statements*

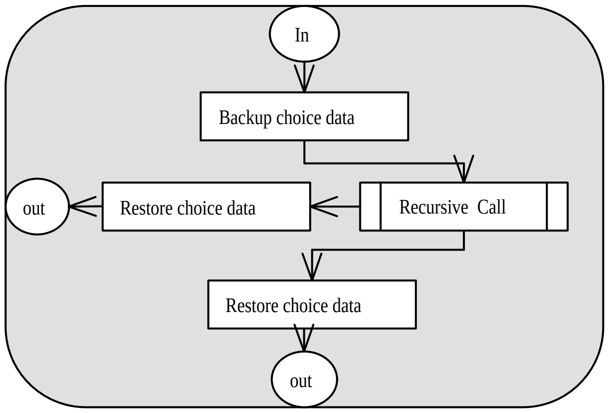


Figure 5.6: *Executable Recurse* in more detail.

5.2 Handshake implementation

A handshake is a complex operation, many conditions have to be met before a handshake can even start. And even when a handshake is in progress, it is not sure whether it will be successful. This section will describe how a handshake works, in detail.

The first step in determining whether a handshake can start is to determine which process *initiates* the handshake, in other words what process is the *handshake initiator*. Note that only processes that want to *send* can initiate a handshake[4].

When a suitable handshake initiator is found, the system locates the potential handshake partners for the initiator i.e. which processes are willing to communicate with the initiator[5]. All potential partners that are found are stored in a potential partner list. What is the format of a potential partner? To be able to describe a potential partner the following information is needed: (See also: 3.2)

- The process ID
- The choice in the process ID that contains the handshake instruction
- A status field that denotes whether this particular partner has
  1. not yet been tried.
  2. been successfully tried.

[4]This implies that processes that want to handshake receive, always block
[5]Note that processes that want handshake receive *will* be included

3. been *un*successfully tried.

The last item is included for optimization and will be discussed in more detail later.

Definition 5.3 (Potential partner format) *The format of a potential partner is* $(status, processID, [choice])$.

This definition automatically yields the next definition.

Definition 5.4 (Potential partner list format) *The format of the list of potential partners is* $[potentialpartner]$ *or* $[(status, processID, [choice])]$.

When all potential partners are found, the system checks whether there are enough partners for the handshake. In other words, if the handshake initiator needs 2 partners, the handshake can continue if the length of the partner list is greater or equal to 2. Note that when there are more potential partners than needed, every combination of partners has to be tried. This is described in more detail later.

5.2.1 Determining the potential partners

To find all partners for a specific process the system has to iterate through all processes and all choices in those processes (recursive) to find a choice that would like to communicate and is compatible with the handshake initiator.

Compatible partners

A potential partner is compatible with the handshake initiator if the following conditions are met:

1. The channels over which to communicate are the same.

2. The process ID of a partner process that wishes to handshake send has to be greater than the process ID of the initiator.

The first condition is obvious, but the second needs further explanation. Suppose process 6 is the handshake initiator and a possible partner is a handshake send statement in process 2. This case has already been examined because all handshake combinations with process 2 as initiator have already been tried. Thus process 6 has already been found to be a partner of process 2 at an earlier stage. In other words, when the combination *2 handshakes with 6* has already been tried, the combination *6 handshakes with 2* may be tried again. Otherwise two distinct handshake combinations will be found which are actually the same.

Note that this is not the case with handshake *receive* statements because they can never initiate a handshake.

5.2.2 Starting the handshake

When enough potential partners are found, the `hs flag` is set(see section 3.3.1) and the handshake can continue. The handshake algorithm will proceed as shown in figure 5.8.

The first step in the handshake procedure is to set the previous handshake state. This is needed when there were more partners than needed in the previous handshake, as described earlier in this section, so that the handshake function can try the next possible handshake combination.

When the previous handshake is identified, the handshake function selects the next possible handshake combination.

Selecting the next handshake combination

Two data structures are used when calculating the next possible handshake combination. These are depicted in figure 5.7.

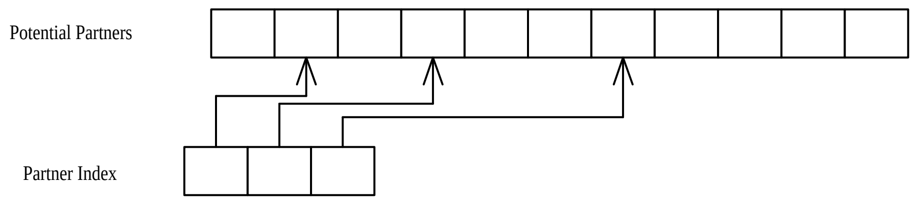


Figure 5.7: Data structures used when calculating the next handshake combination

In this figure, potential partners is the potential partner list, and partner index is a list containing the actual partners used in the handshake combination. This setup is used for the previous handshake and for the next handshake. When this is the first handshake that is being attempted, the values in the partner index list are empty i.e. they do not point to a potential partner.

Selecting a next handshake combination is the process of modifying the partner index so that the new partner index values point to the partners used in the next handshake combination.

The actual process of selecting the next possible handshake is a complicated algorithm. The complete algorithm in pseudo code is depicted in figure 5.9.

When a valid handshake combination has been found, the handshake function tries to execute the handshake to see whether the selected handshake can execute. In other words, are the parameters used in the handshake, the values to be send/received, compatible?

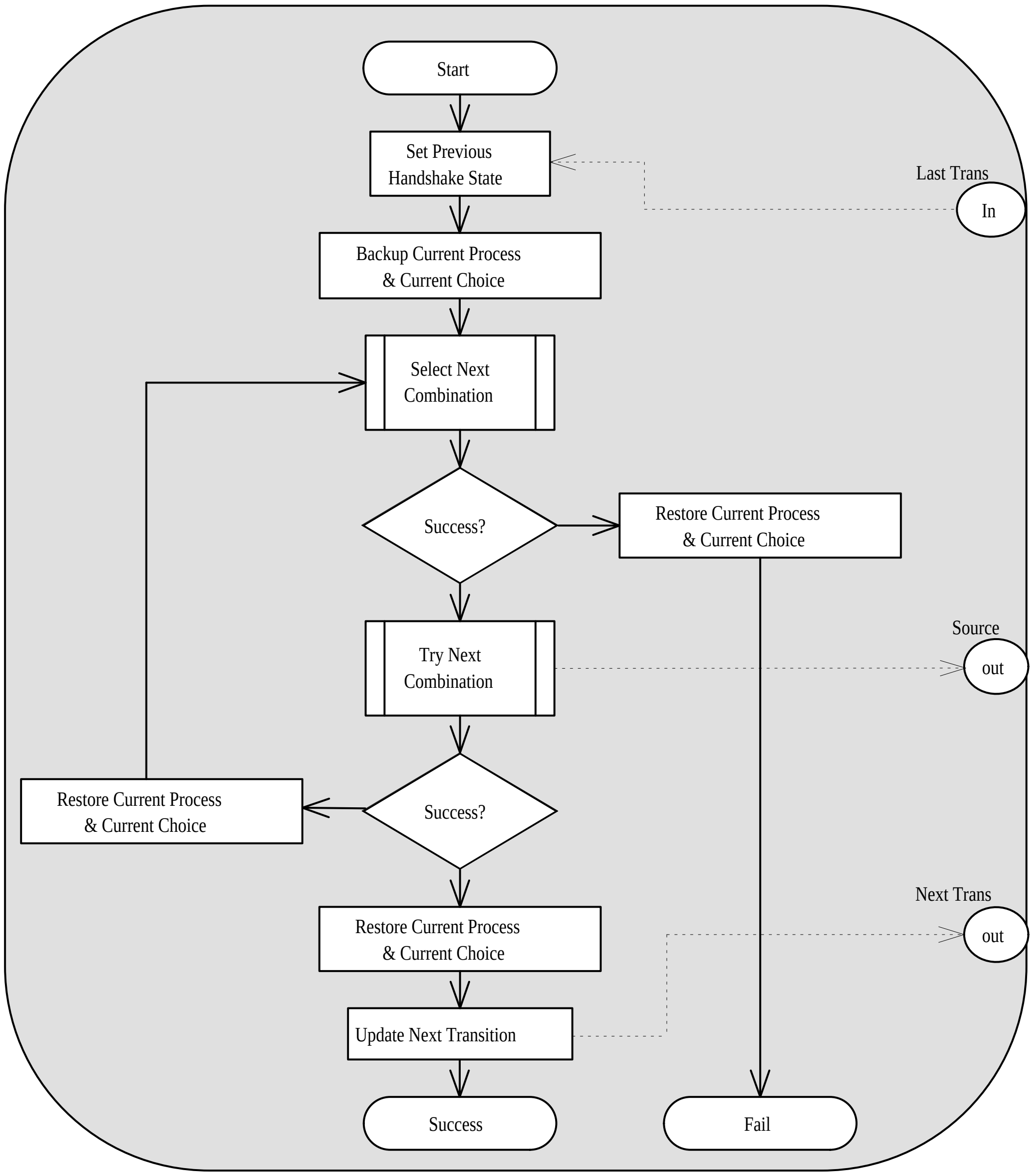

Start
Set Previous Handshake State
Last Trans
In
Backup Current Process & Current Choice
Select Next Combination
Success?
Restore Current Process & Current Choice
Try Next Combination
Source
out
Restore Current Process & Current Choice
Success?
Next Trans
out
Restore Current Process & Current Choice
Update Next Transition
Success
Fail

Figure 5.8: Overview of *handshake*.

5.2.3 Trying a handshake combination

The process of trying a handshake is simply to interpret all instructions belonging to the handshake[6] and to check whether it is enabled. The handshake is enabled when the parameters of the handshake, the variables and constants to be communicated, are compatible.

If a handshake is found to be *not* enabled, the handshake function will try the next possible handshake. When there is no possible handshake, the handshake attempt has failed and the handshake is not executable. (see figure 5.8)

[6]A specialized function is created for this, this is*not* done by the `executable` function

```
for $i = 0$ to sizeof($index$) $- 1$ do
    if $i == 0$ then
        if not $partners[index[i]]$ exists then
            $index[i] = 0$
        else
            if not $partners[index[i]]$ has been successfully tried then
            begin
                $j = index[i] + 1$
                repeat
                    if not $partners[j]$ exists then
                        return no next combination possible
                    else if not $partners[j]$ has been unsuccessfully tried then
                        $break = true$
                    else
                        $j = j + 1$
                until $break$
                $index[i] = j$
            end
    else
        if not $partners[index[i]]$ exists then
            if not $partners[index[i-1]+1]$ exists then
                return no next combination possible
            else begin
                $index[i] = index[i-1] + 1$
                while process id of $partners[index[i]]$ and $partners[index[i-1]]$ are equal do
                begin
                    $index[i] = index[i] + 1$
                    if not $partners[index[i]]$ exists then return no next combination possible
                end
            end
        else
            if $index[i] \leq index[i-1]$ or not $partners[index[i]]$ has been successfully tried then
            begin
                $j = index[i-1] + 1$
                repeat
                    if not $partners[j]$ exists then
                        return no next combination possible
                    else
                        if not $partners[j]$ has been unsuccessfully tried and $j > index[i-1]$ then
                            if process id of $partners[j]$ and $partners[j-1]$ are equal and
                                $partners[j]$ has been successfully tried then
                                $j = j + 1$
                            else $break = true$
                        else $j = j + 1$
                until $break$
                $index[i] = j$
            end
```

Figure 5.9: The *Next Handshake Combination* algorithm

Chapter 6

The Systematic State Explorer

The systematic state explorer uses the *next state generator* (see chapter 3) to generate all possible states of a program, written in the source language. During the exploration, the following errors can be detected:

- Deadlock (as described in section 2.8).
- Atomic Deadlock (as described in section 2.6).
- Assertion failed.
- Division by zero.

The systematic state explorer has three modes of operation:

- An *exploration* mode, in which every reachable state is visited and checked for errors.
- A *simulation* mode, in which a "random run" is performed. This means that whenever there is more than one possible next state, the actual next state is a random choice out of all possible next states. Note that this can lead to a simulation that will run forever, since the simulation may end up in a loop. Also it is not guaranteed that all reachable states are actually reached. However, *simulation mode* may detect an error faster than *exploration* mode.
- An *interactive simulation*. This is the same as *simulation* mode, with the important exception that whenever there is more that on possible next state, *the user* selects the actual next state.

6.1 Exploration mode

Exploration mode systematically generates every possible state that can be reached, and checks them, on-the-fly, for errors. It performs a *depth-first* search to visit the states. When a state is found that has already been visited, that state is ignored. This ensures that eventually all possible states will be checked, since it is not possible that the state explorer enters an infinite loop. The process of generating all possible states, i.e. the depth first search is:

```
procedure dfs(s : state)
    if error(s) then report error fi
    add s to Statespace
    foreach successor t of s do
        if t not in Statespace then dfs(t) fi
    od
end dfs
```

Figure 6.1: The depth first search algorithm

An example of this process is depicted in figure 6.2. The search starts in the `init` state (the state just after the init section has been executed). From this state, the first unexplored next state is state 1 and after that, state 2. When state 2 is the current state, no new states can be found so the state explorer backtracks to to the last successful state, state 1.

This time, the first *unexplored* next state is state 3 and after that state 4. In state 4, the state explorer could go to state 2, but this is not an unexplored state and is ignored. Here, the next unexplored state is state 5. State 5 is again a dead-end and the state explorer backtracks all the way back to the init state. From here state 6 and 7 are generated in the same way as the other states.

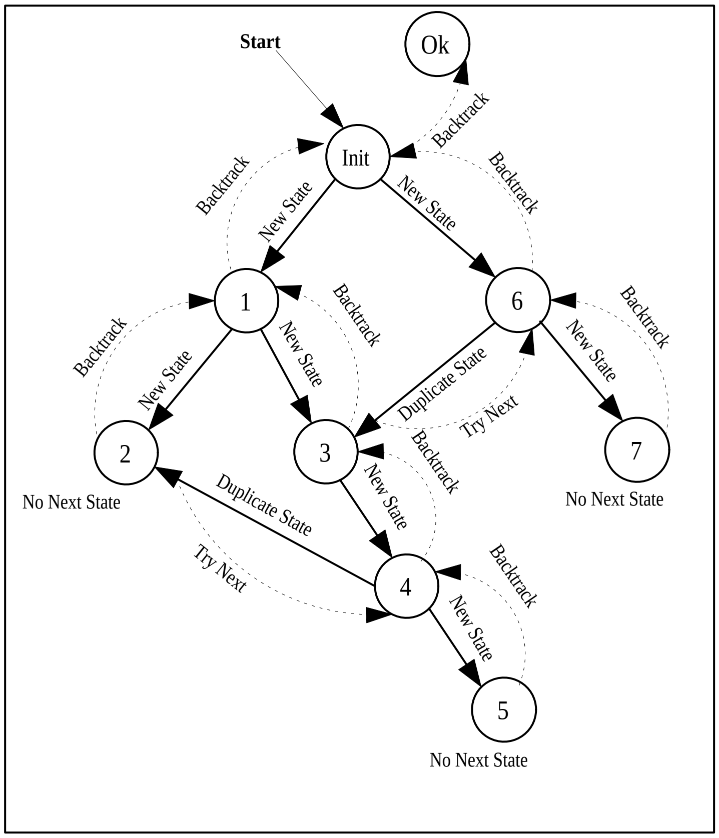


Figure 6.2: An Example: generating all possible states

When the state explorer, again, arrives in the init state, and there are no next unexplored states left, and backtrack is not possible, then all states have been explored (the `OK` state in the picture).

To make this algorithm work, all states that have been found have to be stored somewhere, so that duplicate states can be detected (for every *potential* new state is checked whether this state is in the list of "known" states). In addition, the path up to the current state has to be stored, to allow backtracking to the last successful state. Consider the following example: Suppose, in figure 6.2, the current state is state 5. The path which has to be stored is: "init" `->` "1" `->` "3" `->` "4".

Also, the state explorer has to remember which transition it took to get in a particular state. This allows the state explorer to skip transitions it already took, during backtracking. Finally, the high level statement responsible for a transition has to be stored to be able to generate a trace (the sequence of statements leading to the error) when an error is detected. The previous example would be stored as: ("init", "init", "init") `->` ("1",*first*, *code*) `->` ("3",*second*, *code*) `->`("4",*first*, *code*) This "path" information is stored in a stack. When a new state is found, this state and the other necessary information is pushed on the stack. When backtracking, this information is popped. Figure 6.3 shows what is in the stack and store(hash table).

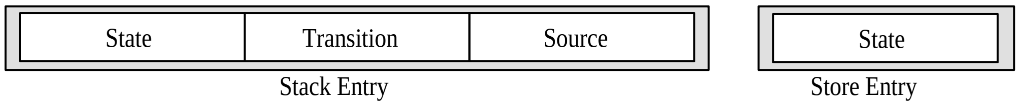


Figure 6.3: Data structures used in the exploration.

6.1.1 The state hash table

For each potential new state that is found, it has to be checked whether this state is already been visited earlier, i.e. is a duplicate state. Since this operation has to be performed so many times it is crucial that the check whether a state is a duplicate state is done very efficient.

An efficient way to store and access the visited states is by using a hash table. What is stored in the hash table are the states that have already been found. After experimenting with different hash functions, the hash function that turned out to be the best (see also: [5], [16]) is shown in figure 6.4:

```
$result = state[0]$
**for** $i = 1$ **to sizeof**$(state) - 1$ **do**
    $result = (result << 3) + state[i]$
**return** $result$
```

Figure 6.4: The hash function

The complete exploration algorithm is shown in figure 6.5. The `NEXT` sub-block in this figure is the next function as described in chapter 3.

6.1.2 Additional validation features

By default, the state explorer searches all states, *until* an error is found. A number features are implemented to change this behavior:

1. The ability to ignore errors and always check all states, counting the number of errors found.
2. The ability to ignore a specific number of errors.
3. The ability to limit the search to a specific depth.
4. The option to detect loops.
5. The option to *not* detect duplicate states.
6. The ability to find the shortest path to an error.

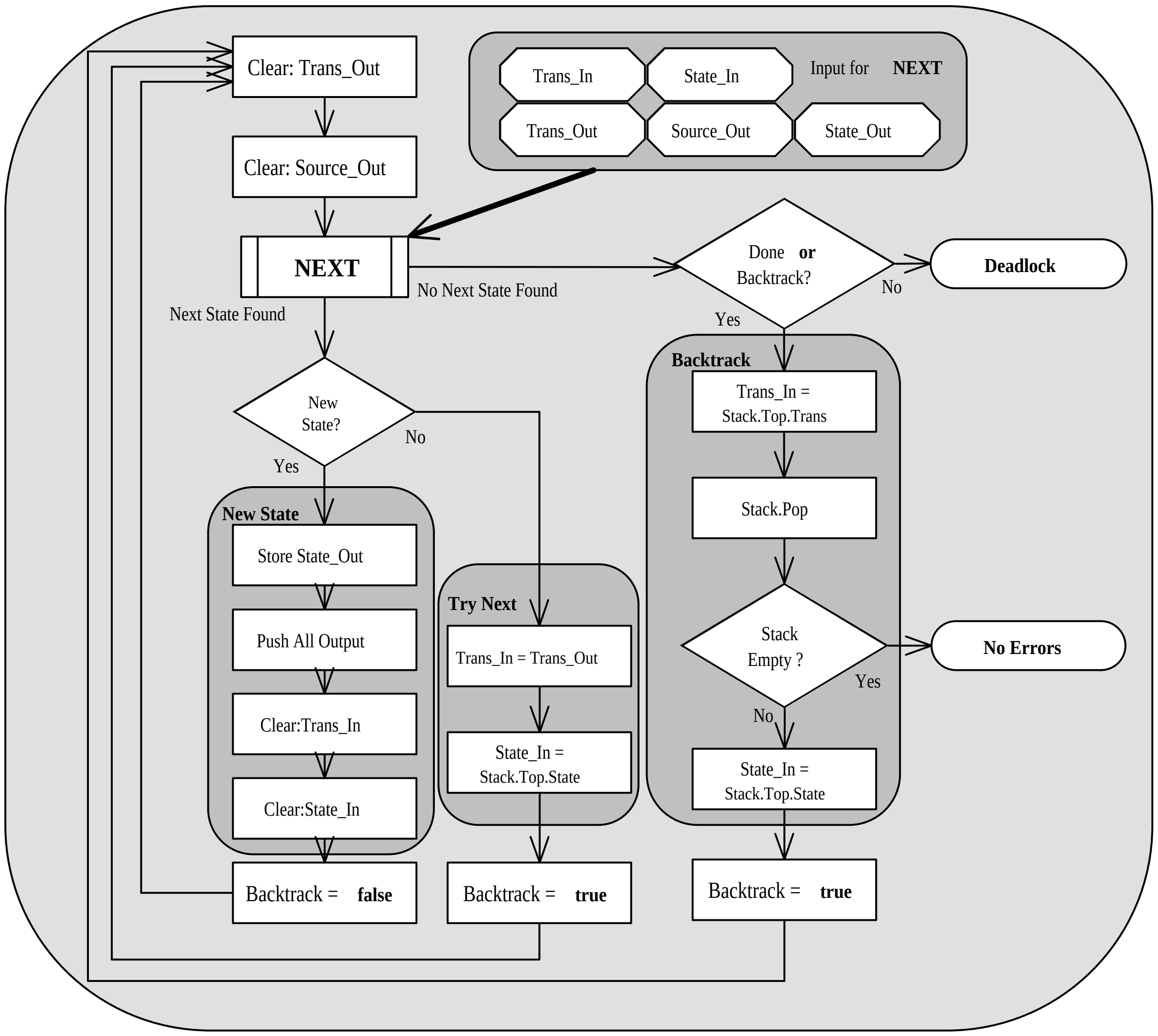


Figure 6.5: The validate/check algorithm

Loops

A loop occurs when the current state is already on the state stack. This means there is a loop in the program. By default, loops are not detected because this is very slow. This is because a linear search over the stack has to be performed.

Duplicate states

A duplicate state is a state that has already been visited once, but is not necessarily a loop. In other words, when the current state is already in the state store (the hash table) a duplicate state has been detected.

The shortest path algorithm

The algorithm to find the shortest path is as follows: The search is started as usual, with duplicate state detection on, and loop detection off. When the first error is detected, the depth at which it is found (and the trace) is remembered. This depth is called the *last error depth*. The search will continue, but will not go beyond depth *last error depth* - 1. After the first error is found, the ”known” duplicate states in the hash table are deleted. Also, duplicate state detection is disabled, to ensure that every possible path is tried. And cycle detection is enabled, to ensure the search will not go into an infinite loop.

When another error is found (at a lower depth) the *last error depth* is set to this value. But duplicate state detection and cycle detection remain unchanged. The search will continue in this fashion until all paths have been tried. The length of the shortest path is the *last error depth.* The trace to this error is returned.

6.2 Simulation mode

Simulation mode performs a simulation of the execution of the model. Whenever a non-deterministic choice *can* be made, a random choice *will* be made. No tests for duplicate states are made, so the simulation can enter an infinite loop. There is no guarantee that every possible state will eventually be reached.

There are two simulation modes:

Random simulation This mode will take a *random* choice when a choice can be made.

Interactive simulation This mode will allow the user to *select* a choice when a choice can be made.

The simulation algorithm is roughly as follows: In the current state, generate *all* possible next states. Select a state from all those states (random or interactive) and let the current state be that state. Repeat this process until an error is found or the program has successfully ended.

6.2.1 Generating all possible next states

The process of generating all possible next states is shown in figure 6.6. The `next` function is

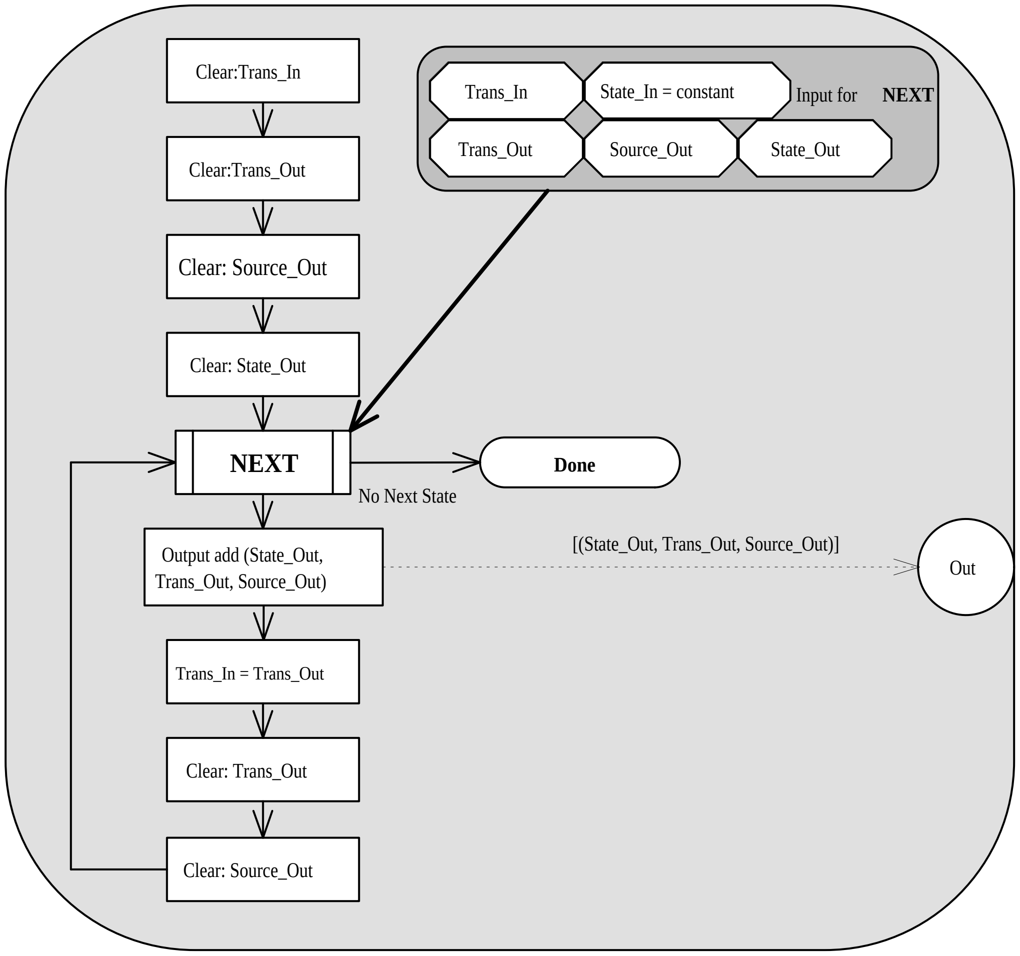


Figure 6.6: Generating all possible transitions

simply called while keeping the input state constant and assigning the output transition to the input transition after each successful next state. This way a list of all possible next states is generated.

Chapter 7

The tool

This chapter describes *MAJoR*, the systematic state explorer tool that was implemented [1]. The tool will be illustrated by examples of models which it can check and statistics about the speed, states visited etc.

7.1 Invoking MAJoR

MAJoR is a command line utility. If no parameters are given, the following output, showing all possible command line options, is given.

```
Syntax: MAJoR <input file> [<option> ...]
Options:
-M<mode>       : 0 = Run a complete model check. (default)
                 1 = Run an interactive simulation.
                 2 = Run a random simulation.
-i             : (Use with -M0) Ignore errors.
-s             : (Use with -M0) Save the shortest path found.
-q             : (Use with -M2) Performs simulation without outputting
                 every step to the screen.
-t             : (Use with -M2) Don't write trace info to trace file.
-S<rseed>      : (Use with -M2) Set the random seed value.
-H<errornr>    : (Use with -M0) Hold at error <errornr>.
-B<bucketsize> : (Use with -M0) Set the initial bucketsize of the hashtable.
                 (default = 100000)
-U<update>     : Set the update value for screen output.
                 Lower is faster update. (default = 1009)
-C<cls>        : Set the clear screen command for the console (-Cclear)
-D<depth>      : (Use with -M0) Set the maximum search depth.
-c             : (Use with -M0) Detect cycles (very slow).
-d             : (Use with -M0) Revisit known states.
                 (do not detect duplicate states).
-L             : (Use with -M0) Limit search depth to depth last error.
-P             : (Use with -M0) Find the shortest path.
```

Figure 7.1: Command-line options of MAJoR

[1]The current name for the tool is *MAJoR*, after the author of the tool. But this name is a working name only and may be subject to change.

The only mandatory parameter is the input file. This file contains the model which has to be checked for errors. All other parameters are optional and are described in the next section.

7.1.1 Command line options

-M<mode> Specifies in what mode the systematic state explorer will run. Use `-M0` to start a complete check (or *validate mode*), `-M1` to start an interactive simulation and `-M2` to start a random simulation.

-i The systematic state explorer will not stop when an error is found. The errors will be counted.

-s The systematic state explorer will remember the trace of the shortest path to an error. This trace is saved to the output file.

q Normally, when running a random simulation, every step that is executed is printed to the screen. This option will not show every step. Instead, a status line is displayed showing the number of seconds the simulation is running, the number of transitions taken and the speed in transitions/second.

t Using this option, no trace information is written to the trace file. This can be convenient when running simulations of very large models. The trace files of such models can be very large (Gigabytes).

-S<rseed> Use the *rseed* value, for example -S1234, to manually initialize the random number generator used by the random simulation mode. When this option is not used, the value is ”randomly” chosen. The *rseed* value can be used to perform the same ”random” simulation twice. The *rseed* value that was used in the last simulation can be found in the trace file.

-H<errornr> When using this option, for example -H3, the systematic state explorer will count and ignore the errors it detects, *until* the number of errors equals *errornr*. This last error will be reported.

-B<bucketsize> Use this option to manually initialize the number of buckets in the hash table, used by the systematic state explorer to store already visited states. Experiment with this value for possible speed optimization. Example: -B100000.

-U<update> The update value means how often status information is written to the screen. New status information is written to the screen every *update* transitions. Use lower values for more accurate status information. Lower values mean faster update but slower performance. Example: -B1000.

-C<cls> Since clearing the screen is not a standard ANSI C command, the console ”clear screen” command can be set using this option. For example -Cclear for UNIX and -Ccls for DOS/WINDOWS.

-D<depth> The search is limited to a depth of *depth*. When this depth is reached, the search will continue, but not beyond this depth.

-c The systematic state explorer will detect cycles.

-d The systematic state explorer will *revisit* known states. Except states that cause a cycle (if the detect cycles option is enabled).

-L When an error is found, the search depth will be limited to the depth of that error.

-P Start the shortest path algorithm as describe in section 6.1.2.

7.1.2 The output files

MAJoR automatically creates two output files, a `.cod` file (code) and a `.trc` file (trace). The name of these files is the name of the input file from which any extensions are removed, plus the new extension (.cod or .trc). For example an input file named `test.mjr` generates `text.cod` and `text.trc`.

The code file (.cod)

The code file contains the intermediate code which was generated from the source language. The code is commented to show which instructions belong to which statement in the source language. An example of a .cod file, taken from the example from section 7.3, is presented in appendix F.

The trace file (.trc)

When an error has been found during a complete check, the complete path to the error, called the *trace* is written to this file. In addition the values of all variables when the error occurred, and what kind of error occurred.

When running a simulation, random or interactive, the current and/or last trace is always written to the file. In addition the last random seed will be written to the trace file in case of a random simulation. An example .trc file is shown in figure 7.7, in section 7.3.

7.2 The status line

When running *MAJoR* in exploration mode, a status line will be displayed which shows statistics about the current search. Figure 7.2 shows an example of the status line.

```
State vector: 30 bytes.
Checking...
Secs  Speed   Depth   States      New      Dup      End     Cycle  Err   Short
   2,  5993,   1458,   11987,    4604,    4236,    3146,        0,   1,   1459
```

Figure 7.2: The status line

Secs How long the validation has been running, in seconds.

Speed The average speed of the validation, States/Secs. The average number of visited states per second.

Depth The current stack depth. Or the length of the current path.

States The number of states tried. This equals *New* + *Dup* + *End* + *Cycle* + *Err*

New The number of new/unique states that have been found.

Dup The number of duplicate states found.

End The number of end states found.

Cycle The number of cycles/loops that have been detected. (This requires the `-c` option).

Err The number of errors that have been found.

Short The length of the shortest path to an error.

7.3 EXAMPLE: Unreliable communication

The code in figure 7.3 shows a model of unreliable communication. Two processes communicate with each other, a sender and a receiver. The sender sends integer values of increasing value (modulo MAX). The receiver receives those values. The assert statement in the receiver process checks whether the value that is sent, is received in the correct order. However, a daemon process is also running. The daemon process can receive the value that is sent by the sender at any time, so the receiver will not receive the correct value.

```
int MAX = 16; chan c = [1] of {byte};

proctype Sender (chan out) {
  byte i; i = 0;
  do
  :: out!i; i = (i+1) % MAX;
  od;
};

proctype Receiver (chan in) {
  byte j; byte k;
  do
  :: in?j; assert(j == k); k = (k+1) % MAX;
  od;
};

proctype Daemon (chan in) {
  byte k;
  do :: in?k; od;
};

init {
    run Sender (c);
    run Receiver (c);
    run Daemon (c);
}
```

Figure 7.3: Source code for unreliable communication example.

7.3.1 Running in validate mode

First the daemon process is not started i.e. the "run Daemon (c)" statement in the init section is commented out. *MAJoR* is invoked like this, assuming the source file is called *communication.mjr*:

```
MAJoR communication.mjr
```

The output is shown in figure 7.4.

As expected, no errors are found. All values send, are received at the correct time. Now the daemon function is started and the validation is started again. The output is shown in figure 7.5.

Again, as expected, an error is found. As seen in the output:

```
Assertion failed: Process: Receiver(2), Statement: assert (j == k)
```

This is the first error the systematic state explorer has detected, and since no other options have been specified the validation is stopped after finding this error. The length of the path is 90, as seen in the output under `Short`. The trace to this error has been written to 'communication.trc'.

```
Parsing...
Warning(Line: 27, Pos: 2) : Identifier 'Daemon' declared at: (Line: 18, Pos: 16)
, but not used.
Done.
State vector: 11 bytes.
Checking...
Secs  Speed   Depth   States     New      Dup      End    Cycle  Err   Short
   0,   505,     -1,     505,     195,     114,     196,       0,   0,    n/a
No errors found.
Closing files...
Freeing memory...
```

Figure 7.4: Unreliable communication example *without* daemon

```
Parsing...Done.
State vector: 14 bytes.
Checking...
Secs  Speed   Depth   States     New      Dup      End    Cycle  Err   Short
   0,    91,     89,      91,      89,       1,       0,       0,   1,     90
Assertion failed: Process: Receiver(2), Statement: assert (j == k)
Writing trace to file 'communication.trc'...Done.
Closing files...
Freeing memory...
```

Figure 7.5: Unreliable communication example *with* daemon

The length of the path to the error is 90 steps. This is a bit long. There is of course a shorter path to an error. The shortest path to an error can be found by using the `-P` command line option. The output after running *MAJoR* :

```
MAJoR communication.mjr -P
```

is shown in figure 7.6.

```
Parsing...Done.
State vector: 14 bytes.
Checking...
Secs  Speed   Depth   States     New      Dup      End    Cycle  Err   Short
   1,  8664,     -1,    8664,    5874,       1,    5875,       8,  68,      7
Assertion failed:
Writing trace to file 'communication.trc'...Done.
Closing files...
Freeing memory...
```

Figure 7.6: Unreliable communication example with daemon, shortest path

The shortest path to an error is now 7 steps. The trace to this error is written to the file 'communication.trc' and is shown in figure 7.7. The trace shows the contents of all variables and the steps the state explorer took to get to the error. The global variable `c` is a channel variable this particular channel has a capacity of one. The reason that two values are displayed is that the first value always denotes the number of elements in the channel. As seen in the trace, the

```
Assertion failed:
Global Variables:
  MAX:16
  c:[0, 0]
Process: Sender(1):
  out:0
  i:1
Process: Receiver(2):
  in:0
  j:1
  k:0
Process: Daemon(3):
  in:0
  k:0
1: Process: Sender(1), Statement: i = 0
2: Process: Sender(1), Statement: out!i
3: Process: Sender(1), Statement: i = (i + 1) % MAX
4: Process: Daemon(3), Statement: in?k
5: Process: Sender(1), Statement: out!i
6: Process: Receiver(2), Statement: in?j
```

Figure 7.7: Trace of unreliable communication example 3

daemon process receives the first value that is send. And thus the assertion will fail the first time the receiver receives something.

7.3.2 Running a random simulation

A random simulation is started by invoking *MAJoR* using the `-M2` option. This will write the resulting trace into the trace file unless the `-t` option is used.

```
MAJoR communication.mjr -M2
```

An example of a possible random trace(taken from the trace file) is shown in figure 7.8. Notice

```
Random seed value: 983796021
1: Process: Sender(1), Statement: i = 0
2: Process: Sender(1), Statement: out!i
3: Process: Receiver(2), Statement: in?j
4: Process: Sender(1), Statement: i = (i + 1) % MAX
5: Process: Sender(1), Statement: out!i
6: Process: Receiver(2), Statement: assert (j == k)
7: Process: Daemon(3), Statement: in?k
8: Process: Sender(1), Statement: i = (i + 1) % MAX
9: Process: Sender(1), Statement: out!i
10: Process: Receiver(2), Statement: k = (k + 1) % MAX
11: Process: Receiver(2), Statement: in?j
Assertion failed:
```

Figure 7.8: Possible random trace of communication.mjr

that the random seed value is included in the trace file so that this exact same random run can be recreated using the `-S<seed>` option.

7.3.3 Running an interactive simulation

The interactive simulation mode (`-M1`) allows the user to step through the program. Possible non-deterministic choices can be made by the user.

```
MAJoR communication.mjr -M1
```

will present the user with the screen as shown in figure 7.9. The user can see the contents of all

```
Global Variables:
  MAX:16
  c:[0, 0]
Process: Sender(1):
  out:0
  i:0
Process: Receiver(2):
  in:0
  j:0
  k:0
Process: Daemon(3):
  in:0
  k:0
--------------------------
-- Possible Transitions --
--------------------------
1: Process: Sender(1), Statement: i = 0
2: [Quit]
--------------------------
Select next transition (1 - 2):
```

Figure 7.9: Interactive simulation, first step

the variables and is able to select the next step from the list of possible transitions. There is only one possible transition in figure 7.9 namely: `1: Process: Sender(1), Statement: i = 0`. The second option, `[Quit]`, quits the interactive simulation.

After selecting the first possible transition *twice*, the list of possible transitions will be of length 3. This is shown in figure 7.10. Note that the option to backtrack, `[Backtrack]`, has appeared. This allows the user to return to a previous state and select another transition.

```
Global Variables:
  MAX:16
  c:[1, 0]
Process: Sender(1):
  out:0
  i:0
Process: Receiver(2):
  in:0
  j:0
  k:0
Process: Daemon(3):
  in:0
  k:0
--------------------------
-- Possible Transitions --
--------------------------
1: Process: Sender(1), Statement: i = (i + 1) % MAX
2: Process: Receiver(2), Statement: in?j
3: Process: Daemon(3), Statement: in?k
4: [Backtrack]
5: [Quit]
--------------------------
Select next transition (1 - 5):
```

Figure 7.10: Interactive simulation, third step

7.4 EXAMPLE: The towers of Hanoi

This example shows how *MAJoR* can be used to solve a puzzle or problem. In this case the puzzle is to solve the ”towers of Hanoi” problem. It also demonstrates the use of pre-post conditions and atomic statements. This puzzle is solved by letting *MAJoR* explore all possible transitions (moving a disc from one tower to another) until an ”error” is detected (an error is generated on purpose in the desired end state. i.e. when all the discs have been moved to the last tower). The source code for this example is shown in figure 7.11. In this example, three discs are used. The source code for examples with 5 and 7 discs can be found in appendix E.

7.4.1 Source code explanation

There is one process, called *tower*, which is instantiated three times. One process for each tower. The *tower* process takes four parameters: *Id* to identify the tower and *s1* to *s3* which are the three discs that can be on the tower. *s1* is the bottom disc and *s3* is the top disc. ”Discs” are integer values from 1 to 3, where large numbers mean a larger disc. A disc can only be placed on a tower when it is empty or the disc below is *larger* than the disc to be placed.

A communication channel, called *Hand* declared. Towers can send discs to the hand and receive discs from the hand. This is the process of moving a disc from tower to tower. When the desired end state is reached, an assertion it thrown.

7.4.2 Results

When running *MAJoR* to solve this problem, the output as shown in fugure 7.12 is produced. The first error is found at depth 19.

When the shortest path has to be found the results are as shown in figure 7.13. The shortest path is of length 15. The trace to this error is shown in figure 7.14(taken from the trace file).

```
chan Hand = [1] of {int};

proctype Tower(int Id; int s1; int s2; int s3) {
    do
    :: {((s3 != 0) && (Id == 3))} assert 0;
    :: {((s3 != 0) && (Id != 3))} atomic{Hand!s3; s3 = 0;};
    :: {((s3 == 0) && (s2 != 0))} Hand?s3 {s3 < s2};
    :: {((s3 == 0) && (s2 != 0))} atomic{Hand!s2; s2 = 0;};
    :: {((s3 == 0) && (s2 == 0) && (s1 != 0))} Hand?s2 {s2 < s1};
    :: {((s3 == 0) && (s2 == 0) && (s1 != 0))} atomic{Hand!s1; s1 = 0;};
    :: {((s3 == 0) && (s2 == 0) && (s1 == 0))} Hand?s1;
    od;
};

init {
    run Tower(1,3,2,1);
    run Tower(2,0,0,0);
    run Tower(3,0,0,0);
}
```

Figure 7.11: ”The towers of Hanoi” source code, three discs

```
Parsing...Done.
State vector: 18 bytes.
Checking...
Secs  Speed   Depth   States      New      Dup      End    Cycle  Err   Short
   0,   121,     18,     121,      48,      42,      30,       0,   1,     19
Assertion failed: Process: Tower(3), Statement: assert 0
Writing trace to file ’Hanoi2.trc’...Done.
Closing files...
Freeing memory...
```

Figure 7.12: Towers of Hanoi result, three discs, first error

One can verify that this is indeed the shortest solution to the Towers of Hanoi problem with three discs.

```
Parsing...Done.
State vector: 18 bytes.
Checking...
Secs  Speed   Depth   States      New      Dup      End    Cycle  Err   Short
   0,   411,     -1,     411,     146,      42,     147,      92,   2,     15
Assertion failed:
Writing trace to file 'Hanoi2.trc'...Done.
Closing files...
Freeing memory...
```

Figure 7.13: Towers of Hanoi result, three discs, shortest path

```
1 : Process: Tower(1), Statement: atomic {Hand!s3;s3 = 0;}
2 : Process: Tower(3), Statement: Hand?s1
3 : Process: Tower(1), Statement: atomic {Hand!s2;s2 = 0;}
4 : Process: Tower(2), Statement: Hand?s1
5 : Process: Tower(3), Statement: atomic {Hand!s1;s1 = 0;}
6 : Process: Tower(2), Statement: Hand?s2
7 : Process: Tower(1), Statement: atomic {Hand!s1;s1 = 0;}
8 : Process: Tower(3), Statement: Hand?s1
9 : Process: Tower(2), Statement: atomic {Hand!s2;s2 = 0;}
10: Process: Tower(1), Statement: Hand?s1
11: Process: Tower(2), Statement: atomic {Hand!s1;s1 = 0;}
12: Process: Tower(3), Statement: Hand?s2
13: Process: Tower(1), Statement: atomic {Hand!s1;s1 = 0;}
14: Process: Tower(3), Statement: Hand?s3
```

Figure 7.14: Shortest path to the Towers of Hanoi problem

7.5 EXAMPLE: Bounded retransmission protocol

In this example the Bounded Retransmission Protocol, which is used in one of the Philips' products (see: [10, 7]) is verified. The source code for this model is found in appendix D. The results are shown in figure 7.15.

```
Parsing...Done.
State vector: 41 bytes.
Checking...
Secs  Speed   Depth   States      New      Dup      End    Cycle  Err   Short
 779, 19164,     -1,14928780, 3868079, 7192621, 3868080,       0,   0,    n/a
No errors found.
Closing files...
Freeing memory...
```

Figure 7.15: Bounded retransmission protocol check results

7.6 EXAMPLE: The dining philosophers problem

The dining philosophers problem is the classic problem of five philosophers who are trying to eat spaghetti with two forks. The philosophers sit at a round table, each having a fork between them. Thus there are five forks. Before a philosopher can eat he needs two forks. Logically it is not possible that all philosophers eat at the same time. And when all philosophers take a fork a deadlock occurs because nobody can take his second fork. The source code for this problem is depicted in figure 7.16.

```
chan chan1 = [1] of {int};
chan chan2 = [1] of {int};
chan chan3 = [1] of {int};
chan chan4 = [1] of {int};
chan chan5 = [1] of {int};

int FORK = 1;

proctype place_forks() {
    atomic {
        chan1!FORK;
        chan2!FORK;
        chan3!FORK;
        chan4!FORK;
        chan5!FORK;
    };
};

proctype philosopher(int left_hand; int right_hand; chan
left_fork; chan right_fork)  {
    if /* Take forks */
    ::left_fork?left_hand; right_fork?right_hand;
    ::right_fork?right_hand; left_fork?left_hand;
    fi;

    skip; /* Eat */

    if /* Drop forks */
    :: left_fork!left_hand; right_fork!right_hand;
    :: right_fork!right_hand; left_fork!left_hand;
    fi;
};

init {
    run place_forks();
    run philosopher(0, 0, chan1, chan2);
    run philosopher(0, 0, chan2, chan3);
    run philosopher(0, 0, chan3, chan4);
    run philosopher(0, 0, chan4, chan5);
    run philosopher(0, 0, chan5, chan1);
}
```

Figure 7.16: Source code for the Dining Philosophers problem

When running MAJoR on this problem, a deadlock is indeed detected. MAJoR is run with the shortest path search option enabled. The output of this run is shown in figure 7.17. The corresponding trace file for the shortest path is shown in figure 7.18. As expected a deadlock occurs when all processes take, for example, their left fork, which is shown in the trace.

```
Parsing...Done.
State vector: 38 bytes.
Checking...
Secs  Speed   Depth   States      New      Dup      End    Cycle  Err   Short
   1, 19910,     -1,   19910,    6464,    9117,    6465,       0,   1,      6
Deadlock detected.
Writing trace to file 'philo.trc'...Done.
Closing files...
Freeing memory...
```

Figure 7.17: MAJoR output for the Dining Philosophers problem

```
   .

Deadlock detected.
Global Variables:
  chan1:[0, 0]
  chan2:[0, 0]
  chan3:[0, 0]
  chan4:[0, 0]
  chan5:[0, 0]
  FORK:1
Process: philosopher(2):
  left_hand:1
  right_hand:0
  left_fork:0
  right_fork:1
Process: philosopher(3):
  left_hand:1
  right_hand:0
  left_fork:1
  right_fork:2
Process: philosopher(4):
  left_hand:1
  right_hand:0
  left_fork:2
  right_fork:3
Process: philosopher(5):
  left_hand:1
  right_hand:0
  left_fork:3
  right_fork:4
Process: philosopher(6):
  left_hand:1
  right_hand:0
  left_fork:4
  right_fork:0
1: Process: place_forks(1), Statement: atomic {chan1!FORK;chan2!FORK;chan3!FORK;
chan4!FORK;chan5!FORK;}
2: Process: philosopher(2), Statement: left_fork?left_hand
3: Process: philosopher(3), Statement: left_fork?left_hand
4: Process: philosopher(4), Statement: left_fork?left_hand
5: Process: philosopher(5), Statement: left_fork?left_hand
6: Process: philosopher(6), Statement: left_fork?left_hand
```

Figure 7.18: Trace file of the shortest path for the Dining Philosophers problem

7.7 Comparison to Spin

Some of the examples presented in the previous sections have also been run using SPIN. Only the examples which could be translated from MAJoR to SPIN without to much hassle are included. These are: the unreliable communication example, the bounded retransmission protocol and the Dining philosophers problem. To make the test more fair, the following SPIN features are disabled:

- Partial Order reduction.
- Dataflow analysis.
- Dead variable elimination.
- Statement merging.

The following changes are also made to the SPIN model code, in order to make the SPIN model as close as possible to the MAJoR model:

- `run` statements in the init section are made atomic to emulate the way MAJoR handles process creation.
- The order of process creation in SPIN is reversed. Again, to emulate the behavior of MAJoR.

The source code and output for the SPIN examples can be found in the following appendices:

- Unreliable communication: Appendix G
- Bounded retransmission protocol (output only): Appendix H
- Dining philosophers: Appendix I

7.7.1 Comparison results

The unreliable communication example

	SPIN	MAJoR
Size State Vector	40 Bytes	14 Bytes
first error		
Number of states	92	91
Stored states	91	89
Matched states	1	1
End states	n/a	0(1 error)
First error depth	93	90
Runtime of verification(sec)	real:0.2 user:0.0 sys:0.0	real:0.37 user:0.10 sys:0.04
shortest path		
Number of states	1064	8664
Stored states	505	5874
Matched states	559	1
End states	n/a	5875(+8 cycles, 68 errors)
Length of shortest path	9	7
Runtime of verification(sec)	real:14.3 user:0.0 sys:0.0	real:13.69 user:13.07 sys:0.04

The bounded retransmission protocol example

	SPIN	MAJoR
Size State Vector	92 Bytes	41 Bytes
Number of states	$1.8214 * 10^7$	$1.4929 * 10^7$
Stored states	$7.4001 * 10^6$	$3.8681 * 10^6$
Matched states	$1.08139 * 10^7$	$7.19262 * 10^6$
End states	n/a	$3.8681 * 10^6$
Runtime of verification(sec)	real:418.9 user:399.3 sys:18.1	real:792.29 user:761.83 sys:27.11
Number of errors found	0	0

The dining philosophers example

	SPIN	MAJoR
Size State Vector	128 Bytes	38 Bytes
Shortest path		
Number of states	23580	19910
Stored states	6216	6464
Matched states	17364	9117
End states	n/a	6465 (1 error)
Length of shortest path	17	6
Runtime of verification(sec)	real:4.7 user:1.8 sys:2.3	real:1.22 user:0.89 sys:0.04

7.7.2 Conclusions

Speed

As seen in the results, SPIN outperforms MAJoR in terms of speed. But the current version of MAJoR is only a prototype, it has not yet been optimized in any way. SPIN, on the other hand, is the results of years of research an improvements. The difference in speed is of course most noticeable in the larger example, the bounded retransmission protocol.

State vector size

In all examples, the state vector size of MAJoR is significantly smaller than the state vector in SPIN. And the current state vector of MAJoR is un-optimized for size. This is very strange. SPIN seems to store more information in the state vector than needed. The only information needed for the state vector(in MAJoR) is all the local and global variables and all the instruction pointers. One explanation for the large state vector size of SPIN is that maybe SPIN needs more information in the state vector for its various optimizations.

Number of states

The number of states in MAJoR has a different meaning than the number of states in SPIN. In SPIN, the number of states equals the number of stored states plus the number of matched states. In MAJoR the number of states is actually the number of times the next function is called. This equals: stored states + matched states + end states + errors. To get MAJoR's total states, add stored states(new) and matched states(dup). MAJoR's end states are in fact points in the traversal of the search tree where the state explorer decides that it has to backtrack.

Comparison of the number of states when using the shortest path algorithm is hard. This is because MAJoR implements the shortest path algorithm in a way which makes it hard to determine the total *new* states visited. The comparison *can* be made when not using the shortest path algorithm though. The examples which do not use the shortest path algorithm are: the unreliable communication example and the bounded retransmission protocol example. The number of states in the former is about equal for SPIN and MAJoR. However when validating the bounded retransmission protocol, SPIN generates significantly more states than MAJoR. This can not be caused by the fact that Spin handles the init section differently since the Spin code was changed to emulate MAJoR's behavior by making the init section atomic.

The source code for this protocol is quite different for Spin and MAJoR. This may cause the difference in the amount of states. MAJoR does not support arrays, while Spin does not support handshake channels.

Another, very simple, model has been verified by MAJoR and Spin. When verifying this model

```
proctype p1() {
    int i = 0;
    do
    :: i < 10000; i=i+1;
    :: i >= 10000; break;
    od;
    (1 > 2);
};

init {
    run p1();
}
```

Figure 7.19: Small example to test state generation.

in Spin and MAJoR, both generate the same amount of states, namely 20003. MAJoR generates one more because the statement `int i = 0` is a state in MAJoR. When optimizations are enabled in Spin, Spin only needs approximately 10000 states because it can merge the statements in the loop. Both verifiers conclude that a deadlock has been detected of course.

Chapter 8

Conclusions

We presented the design and implementation of MAJoR, a tool that can systematically explore all states of a model and check them for errors as described in chapter 6. However, it is not yet a real model checker; it is not possible to perform the range of checks as supported by the CTL and/or LTL languages (see [15]). Also, MAJoR does not optimize things to the same extent as Spin does. Spin uses optimization techniques, such as partial order reduction, to reduce the size of the state space. State vector compression and bitstate hashing are used to allow even more states to be explored.

The work shares similarities to the work done by Jaco Geldenhuys [12]. Apart from the addition of multiway handshake communication and pre- and post-conditions, there are a number of other differences. Unlike Jaco's work, MAJoR's implementation of non-determinism does not put any restrictions on the kind of statements that can be run non-deterministically. MAJoR allows *every* statement to be a part of the non-deterministic if statement, even other if statements. This can lead to nested if-statements which greatly increases the complexity of transitions and the implementation of non-determinism. However, the general approach of MAJoR can also lead to very annoying bugs and problems (see section 8.3) and slower performance.

8.1 Future work

Logically, the first step from here is to add CTL and/or LTL support to MAJoR. Implementing a model checker for CTL and/or LTL can be done without altering the underlying state generation structure of MAJoR. No changes to the next function are necessary.

Optimization techniques such as bitstate hashing and state compression can easily be implemented since these techniques do not require the other parts of the program to be intensely modified. Partial order reduction however, is not so easily implemented. It will probably require serious modifications in the state generator.

8.2 Suggestions for improvement

The following is a list of suggestions, improvements, or features that have not yet been implemented.

8.2.1 Re-design of the intermediate language

Currently, the instructions of the intermediate language are too much specifically designed for the input language. The intermediate language should be made more general to be able to design other high level languages for it more easily.

8.2.2 Minor improvements on the input language

Arrays Currently, arrays are not yet supported. The grammar of the input language can parse them but no code can be generated to allow arrays yet. The lack of arrays severely limits the ease of writing complex models, and they should be added as soon as possible.

Data types Currently, only one data type is supported, the integer. The grammar can parse other data types, but they are still mapped onto an integer. To reduce the size of the state vector, smaller data types such as *byte* and *bit* should be added.

Ranged types The input language grammar already recognizes ranged types. But they are not yet implemented. Ranged types can be used to declare a type of a specific range of numbers, for example [1..15]. This can be used to reduce the size of the state vector.

Type checking The type checking in the parser module is far from completed.

Re-design of the input language grammar The current input language grammar is much like the general available C++ grammar [1]. The current grammar is however too complicated and too ambitious because it allows too many language features that are not supported by MAJoR. It should be rewritten to make it simpler to add rules. This will make the addition of arrays, for example much easier. Also it will improve readability and maintainability of the parser module.

8.2.3 Implementation improvements and additions

Re-design of the parser The current implementation of the parser module, using a class for each grammar rule is not practical and unmaintainable. The parser module should be rewritten.

Optimize cycle detection Currently the cycle detection algorithm uses an extremely slow linear search through the stack. This can be optimized using an additional bit in the hash table for each visited state, which indicates whether that state is in the stack or not.

Addition of breadth first search As an experiment, breadth first search of the state space could be added. This may speed up the shortest path algorithm.

8.2.4 C++ and the STL library

The entire tool has been written in strict ANSI C++ [1] using the STL library [4]. Although the STL library provides a lot of functionality, it may be better to write some parts yourself.

Instead of using the C++ STL hash table, an new specialized hash table could be written. One advantage of this is that with a different hash table, collisions can be detected which is valuable information. The STL hash table does not allow this.

Instead of relying on the default STL memory allocation, it may be better to manage memory allocation yourself. For some reason the standard STL implementation seems to use too much memory. When doing it yourself it is possible to have a more precise idea of how much memory is actually used. Luckily, all the STL data structures allow it to create custom memory allocators for them.

8.3 Known problems

The following is a list of problems or bugs which have not been solved yet.

A `do` statement inside an atomic statement. A `do` statement inside an atomic statement will cause problems when the conditions to enable the choices in the `do` statement are not distinct. This is because an atomic `do` statement may generate an infinite long transition. The result is a stack overflow.

Handshake communication inside an atomic statement. Using a handshake statement inside an atomic statement causes undesired results, unexpected atomic deadlocks or otherwise erratic behavior.

Appendix A

Grammar of the source language

<*specification*>	:	<*global_declaration_list*> <*initialize*>
<*initialize*>	:	**init {** <run_process_list> **}**
<*run_process_list*>	:	<*run_process*>
	:	<*run_process*> <*run_process_list*>
<*run_process*>	:	**run** <*identifier*> **();**
	:	**run** <*identifier*> **(** <*parameter_list*> **);**
<*global_declaration_list*>	:	<*global_declaration*>
	:	<*global_declaration*> <*global_declaration_list*>
<*global_declaration*>	:	<*process_definition*> **;**
	:	<*declaration*> **;**
<*process_definition*>	:	**proctype** <*identifier*> **(** <*declaration_list*> **) {** <*statement_list*> **}**
	:	**proctype** <*identifier*> **() {** <*statement_list*> **}**
<*declaration*>	:	<*type_specifier*> <*variable_list*>
<*declaration_list*>	:	<*declaration*>
	:	<*declaration*> **;** <*declaration_list*>
<*statement_list*>	:	<*statement*>
	:	<*statement*> <*statement_list*>
<*statement*>	:	<*identifier*> **:** <*guarded_statement*> **;**
	:	<*guarded_statement*> **;**
<*guarded_statement*>	:	<*guardable_statement*>
	:	<*pre_guard*> <*guardable_statement*>
	:	<*guardable_statement*> <*post_guard*>
	:	<*pre_guard*> <*guardable_statement*> <*post_guard*>
<*guardable_statement*>	:	<*complex_statement*>
	:	<*simple_statement*>
	:	<*assignment_statement*>
<*complex_statement*>	:	<*if_statement*>
	:	<*do_statement*>
	:	<*expression*>
	:	<*communication_statement*>
<*simple_statement*>	:	<*compound_statement*>
	:	<*atomic_statement*>
	:	<*goto_statement*>
	:	<*skip_statement*>
	:	<*assert_statement*>
	:	<*declaration*>

<assignment_statement>	:	*<unary_expression>* = *<expression>*
<atomic_statement>	:	**atomic {** *<statement_list>* **}**
<goto_statement>	:	**goto** *<identifier>*
<skip_statement>	:	**skip**
<assert_statement>	:	**assert** *<expression>*
<compound_statement>	:	**{** *<statement_list>* **}**
<if_statement>	:	**if** *<if_choice_list>* **fi**
<do_statement>	:	**do** *<do_choice_list>* **od**
<communication_statement>	:	*<send_statement>*
	:	*<receive_statement>*
<send_statement>	:	*<postfix_expression>* **!** *<parameter_list>*
<receive_statement>	:	*<postfix_expression>* **?** *<parameter_list>*
<parameter_list>	:	*<expression>*
	:	*<expression>* **,** *<parameter_list>*
<variable_list>	:	*<init_variable>*
	:	*<init_variable>* **,** *<variable_list>*
<init_variable>	:	*<variable>*
	:	*<variable>* = *<expression>*
	:	*<variable>* = *<channel_init>*
<pre_guard>	:	**{** *<expression>* **}**
<post_guard>	:	**{** *<expression>* **}**
<channel_init>	:	[*<expression>*] **of {** *<type_list>* **}**
<variable>	:	*<postfix_expression>*
<type_list>	:	*<type_specifier>*
	:	*<type_specifier>* **,** *<type_list>*
<type_specifier>	:	**bit**
	:	**byte**
	:	**int**
	:	**chan**
	:	**hs**
	:	**char**
	:	**int** *<ranged_type>*
	:	**byte** *<ranged_type>*
	:	**char** *<ranged_type>*
<ranged_type>	:	[*<literal>* **..** *<literal>*]
<if_choice_list>	:	*<if_choice>*
	:	*<if_choice>* *<if_choice_list>*
	:	*<if_choice>* *<if_choice_else>*
<do_choice_list>	:	*<do_choice>*
	:	*<do_choice>* *<do_choice_list>*
	:	*<do_choice>* *<do_choice_else>*
<do_statement_list>	:	*<in_do_statement>*
	:	*<in_do_statement>* *<do_statement_list>*
<if_choice>	:	**::** *<statement_list>*
<if_choice_else>	:	**:: else ;** *<statement_list>*
<do_choice>	:	**::** *<do_statement_list>*
<do_choice_else>	:	**:: else ;** *<do_statement_list>*
<in_do_statement>	:	*<statement>*
	:	*<break_statement>*
<break_statement>	:	**break;**

<expression>	:	*<logical_and_expression>*
	:	*<expression>* || *<logical_and_expression>*
<logical_and_expression>	:	*<inclusive_or_expression>*
	:	*<logical_and_expression>* **&&** *<inclusive_or_expression>*
<inclusive_or_expression>	:	*<exclusive_or_expression>*
	:	*<inclusive_or_expression>* | *<exclusive_or_expression>*
<exclusive_or_expression>	:	*<and_expression>*
	:	*<exclusive_or_expression>* ˆ *<and_expression>*
<and_expression>	:	*<equality_expression>*
	:	*<and_expression>* **&** *<equality_expression>*
<equality_expression>	:	*<relational_expression>*
	:	*<equality_expression>* *<equality_operator>* *<relational_expression>*
<relational_expression>	:	*<shift_expression>*
	:	*<relational_expression>* *<relational_operator>* *<shift_expression>*
<shift_expression>	:	*<additive_expression>*
	:	*<shift_expression>* *<shift_operator>* *<additive_expression>*
<additive_expression>	:	*<multiplicative_expression>*
	:	*<additive_expression>* *<additive_operator>* *<multiplicative_expression>*
<multiplicative_expression>	:	*<unary_expression>*
	:	*<multiplicative_expression>* *<multiplicative_operator>* *<unary_expression>*
<unary_expression>	:	*<postfix_expression>*
	:	*<unary_operator>* *<unary_expression>*
<postfix_expression>	:	*<primary_expression>*
	:	*<postfix_expression>* [*<number>*]
	:	*<postfix_expression>* *<postfix_operator>*
<primary_expression>	:	*<literal>*
	:	*<identifier>*
	:	*<array_init>*
	:	**(** *<expression>* **)**
<literal>	:	**NUMBER**
	:	**CHARACTER**
<array_init>	:	**{** *<parameter_list>* **}**
<identifier>	:	**IDENTIFIER**
<equality_operator>	:	**==**
	:	**!=**
<relational_operator>	:	<
	:	>
	:	<=
	:	>=
<shift_operator>	:	<<
	:	>>
<additiv_operator>	:	+
	:	−
<multiplicative_operator>	:	*
	:	/
	:	%
<unary_operator>	:	++
	:	−−
	:	+
	:	−
	:	!
	:	~
<postfix_operator>	:	++
	:	−−

Appendix B

The instruction set

B.1 Mathematical instructions

Name	Description	Effect
ADD	Addition	POP B; POP A; PUSH A + B
SUB	Subtraction	POP B; POP A; PUSH A - B
MUL	Multiplication	POP B; POP A; PUSH A * B
DIV	Division	POP B; POP A; PUSH A / B
MOD	Remainder	POP B; POP A; PUSH A % B
INC	Increment	POP A; PUSH A + 1
DEC	Decrement	POP A; PUSH A - 1
NEG	Negate	POP A; PUSH -A
BAND	Bitwise AND	POP B; POP A; PUSH A & B
BNOT	Bitwise NOT	POP A; PUSH ~A
BOR	Bitwise OR	POP B; POP A; PUSH A | B
BXOR	Bitwise XOR	POP B; POP A; PUSH A ^ B
LAND	Logical AND	POP B; POP A; PUSH A && B
LOR	Logical OR	POP B; POP A; PUSH A || B
LNOT	Logical NOT	POP A; PUSH !A
SHR	Shift right	POP B; POP A; PUSH A >> B
SHL	Shift left	POP B; POP A; PUSH A << B

B.2 Relational instructions

Name	Description	Effect
EQ	Equal	POP B; POP A; PUSH A == B
NE	Not equal	POP B; POP A; PUSH A != B
GT	Greater than	POP B; POP A; PUSH A > B
GE	Greater than or equal	POP B; POP A; PUSH A >= B
LT	Less than	POP B; POP A; PUSH A < B
LE	Less than or equal	POP B; POP A; PUSH A <= B

B.3 Jump instructions

Name	Description	Flags
JMP x	Jump to code address x	
GOTO x	Jump to code address x	Success

B.4 Stack modifying instructions

Name	Description	Effect	Flags
STORE	Store data from stack in memory	POP B; POP A; MEM[A] = B	Success
EVAL	Evaluate expressions	POP A; if A = 0 Fail else Success	Success *or* Fail
LDGADDR x	Push address of global variable	PUSH x	
LDGVAR x	Push value of global variable	PUSH MEM[x]	
LDLADDR x	Push address of local variable	PUSH local2global(x)	
LDLVAR x	Push value of local variable	PUSH MEM[local2global(x)]	
PUSH x	Push constant value	PUSH x	
XCHG	Exchange two values on the stack	POP B; POP A; PUSH B; PUSH A	

B.5 Non-determinism instructions

Name	Description & Effect	Flags
CHOICES x	Initiate non-determinism, x choices. Followed by x **CHOICE** instructions	
CHOICE x	Code for a choice is located at address x	Choice
ELSE x	Same as **CHOICE**, but this is an else choice	Choice & Else

B.6 Communication instructions

Name	Description & Effect	Flags
CHAN (*send/receive*) (*local/global*) *channel*	Initiate asynchronous communication. Can fail if communication is not possible	Fail
HS (*send/receive*) (*local/global*) *channel*	Initiate handshake communication. Can fail if communication not possible	Fail
RECEIVE	Executes & Verifies the actual receive operation	
SEND	Executes & Verifies the actual send operation	Fail *or* Success

B.7 Flags instructions

Name	Description & Effect	Flags
CLRA	Decrements the atomic flag	Atomic
CLRPP	Decrements the prepost flag	PrePost
SETA	Increments the atomic flag	Atomic
SETPP	Increments the prepost flag	PrePost

B.8 Miscellaneous instructions

Name	Description	Effect	Flags
ASRT	Assert	POP A; ASSERT A	Success
HALT	Halts the current process		Success
IRET	Returns control to the init section after parameters for a proctype have been assigned		Success
NOOP	Does nothing		Success
RUN x	Starts process with ID x		Success

Appendix C

Translation of the source language

<*specification*>	:	<*global_declaration_list*>
		<*initialize*>
<*initialize*>	:	**SETA**
		<run_process_list>
		CLRA
		HALT
<*run_process_list*>	:	<*run_process*>
		[<*run_process_list*>]
<*run_process*>	:	<*identifier(+function_context)*>
		[<*parameter_list*>]
		RUN *process_id*
<*global_declaration_list*>	:	<*global_declaration*>
		[<*global_declaration_list*>]
<*global_declaration*>	:	<*process_definition*>
	:	<*declaration(+global_context)*>
<*process_definition*>	:	<*identifier(+function_context)*>
		[<*declaration_list(+param_decl_context)*>]
		IRET
		<*statement_list*>
		HALT
<*declaration*>	:	**if** *type* == *communication* **then** *(+chan_id_context)*
		<*type_specifier(+declare_context)*>
		<*variable_list(+declare_context)*>
<*declaration_list*>	:	<*declaration*>
		[<*declaration_list*>]
<*statement_list*>	:	<*statement*>
		[<*statement_list*>]
<*statement*>	:	[<*identifier(+label_context)*>]
		<*guarded_statement*>
<*guarded_statement*>	:	**if** *communication* **then CHAN** *params* or **HS** *params*
		SETPP
		[<*pre_guard*>]
		<*guardable_statement*>
		[<*post_guard*>]
		CLRPP
<*guardable_statement*>	:	<*complex_statement*>
	:	<*simple_statement*>
	:	<*assignment_statement*>

```
<complex_statement>          :  <if_statement>
                             :  <do_statement>
                             :  <expression>
                                EVAL
                             :  <communication_statement>
<simple_statement>           :  <compound_statement>
                             :  <atomic_statement>
                             :  <goto_statement>
                             :  <skip_statement>
                             :  <assert_statement>
                             :  <declaration>
<assignment_statement>       :  <unary_expression(+assign_context)>
                                <expression>
                                STORE
<atomic_statement>           :  SETA
                                <statement_list>
                                CLRA
<goto_statement>             :  <identifier(+label_context)>
                                GOTO label adress
<skip_statement>             :  NOOP
<assert_statement>           :  <expression> ASRT
<compound_statement>         :  <statement_list>
<if_statement>               :  CHOICES choice_count
                                for i = 1 to choice_count - 1 do
                                    CHOICE choice_addr
                                if lastchoice = else then
                                    ELSE choice_addr
                                else
                                    CHOICE choice_addr
                                <if_choice_list>

<do_statement>               :  CHOICES choice_count
                                for i = 1 to choice_count - 1 do
                                    CHOICE choice_addr
                                if lastchoice = else then
                                    ELSE choice_addr
                                else
                                    CHOICE choice_addr
                                <do_choice_list>

<communication_statement>    :  <send_statement>
                             :  <receive_statement>
<send_statement>             :  <postfix_expression(+chan_id_context)>
                                <parameter_list(+chan_send_context)>
                                SEND
<receive_statement>          :  RECEIVE
                                SETPP
                                <postfix_expression(+chan_id_context)>
                                <parameter_list(+chan_receive_context)>
                                CLRPP
```

```
<parameter_list(chan_receive_context)>  :  if expression = variable then
                                                <expression(+chan_receive_variable_context)>
                                           else
                                                <expression>
                                                EQ
                                                EVAL
                                           [<parameter_list>]
<parameter_list>                        :  <expression>
                                           [<parameter_list>]
<variable_list>                         :  <init_variable>
                                           [<variable_list>]
<init_variable>                         :  <variable>
                                        :  <variable(+assign_context, +init_var_context)>
                                           <expression(-declare_context, +init_var_context)>
                                           STORE
                                        :  <variable(+init_var_context, +chan_declare_context)>
                                           <channel_init(+init_var_context)>
<pre_guard>                             :  <expression>
                                           EVAL
<post_guard>                            :  <expression>
                                           EVAL
<channel_init>                          :  <expression(+chan_id_context)>
                                           <type_list>
<variable>                              :  <postfix_expression>
<type_list>                             :  <type_specifier>
                                           [<type_list>]
<type_specifier>                        :  no code
<ranged_type>                           :  no code
<if_choice_list>                        :  <if_choice>
                                           [<if_choice_list>]
                                        :  <if_choice>
                                           <if_choice_else>
<do_choice_list>                        :  <do_choice>
                                           [<do_choice_list>]
                                        :  <do_choice>
                                           <do_choice_else>
<do_statement_list>                     :  <in_do_statement>
                                           [<do_statement_list>]
<if_choice>                             :  <statement_list>
                                           JMP end_of_if
<if_choice_else>                        :  <statement_list>
                                           JMP end_of_if
<do_choice>                             :  <statement_list>
                                           JMP start_of_do
<do_choice_else>                        :  <statement_list>
                                           JMP start_of_if
<in_do_statement>                       :  <statement>
                                        :  <break_statement>
<break_statement>                       :  JMP out_of_do
```

<expression>	:	*<logical_and_expression>*
	:	*<expression>*
		<logical_and_expression>
		LOR
<logical_and_expression>	:	*<inclusive_or_expression>*
	:	*<logical_and_expression>*
		<inclusive_or_expression>
		LAND
<inclusive_or_expression>	:	*<exclusive_or_expression>*
	:	*<inclusive_or_expression>*
		<exclusive_or_expression>
		BOR
<exclusive_or_expression>	:	*<and_expression>*
	:	*<exclusive_or_expression>*
		<and_expression>
		BXOR
<and_expression>	:	*<equality_expression>*
	:	*<and_expression>*
		<equality_expression>
		BAND
<equality_expression>	:	*<relational_expression>*
	:	*<equality_expression>*
		<relational_expression>
		EQ|NE
<relational_expression>	:	*<shift_expression>*
	:	*<relational_expression>*
		<shift_expression>
		LT|GT|LEQ|GEQ
<shift_expression>	:	*<additive_expression>*
	:	*<shift_expression>*
		<additive_expression>
		SHR|SHL
<additive_expression>	:	*<multiplicative_expression>*
	:	*<additive_expression>*
		<multiplicative_expression>
		ADD|SUB
<multiplicative_expression>	:	*<unary_expression>*
	:	*<mulitplicative_expression>*
		<unary_expression>
		MUL|DIV|MOD
<unary_expression>	:	*<postfix_expression>*
	:	*<unary_expression>*
		INC|DEC|NEG|LNOT|BNOT
<postfix_expression>	:	*<primary_expression>*
	:	*<postfix_expression>*
		<number>
	:	*<postfix_expression>*
		<postfix_operator> (not yet implemented)
<primary_expression>	:	*<literal>*
	:	*<identifier>*
	:	*<array_init>*
	:	*<expression>*
<array_init>	:	*<parameter_list>* (not yet implemented)
<literal>	:	**PUSH** *literal*

```
<identifier(chan_id_context)>                   : no code
<identifier(label_context)>                     : no code
<identifier(param_decl_context)>                : LDLADDR identifier
                                                  XCHG
                                                  STORE
<identifier(assign_context)>                    : if $identifier = local$ then
                                                      LDLADDR identifier
                                                  else
                                                      LDGADDR identifier

<identifier(chan_receive_variable_context)>     : if $identifier = local$ then
                                                      LDLADDR identifier
                                                  else
                                                      LDGADDR identifier
                                                  XCHG
                                                  STORE
<identifier(declare_context)>                   : no code
<identifier(function_context)>                  : no code
<identifier>                                    : if $identifier = local$ then
                                                      LDLVAR identifier
                                                  else
                                                      if $identifier = communication$ then
                                                          PUSH channel_id
                                                      else
                                                          LDGVAR identifier
```

Appendix D

Bounded retransmission protocol source code

```
hs Sin = [2] of {bit} ;
hs Sout = [2] of {byte } ;
hs Rout = [2] of {byte , byte} ;
hs ChunkTimeout = [2] of {bit} ;
hs SyncWait = [2] of {bit} ;

chan K  = [1] of {bit, bit, bit, byte} ;
chan L  = [1] of {bit} ;

byte    n ;

byte    d1 ;
byte    d2 ;
byte    d3 ;
byte    sInd ;
byte    k ;

byte    e1_ind ;
byte    e2_ind ;
byte    e3_ind ;
byte    e1_val ;
byte    e2_val ;
byte    e3_val ;

bit checknow ;

proctype Environment()
{
  byte i ;
  byte v ;

  do
  ::  Sin! 1  ;
      if ::  (n>0) -> checknow = 1 ;
                  checknow = 0 ;
      :: else -> skip ; fi  ;
      atomic { d1=0 ; d2=0 ; d3=0 ; } ;
      if
      :: n = 1 ;
      :: n = 2 ;
      :: n = 3 ;
      fi ;
    atomic{
      if
      :: n==1 -> if
                 :: d1 = 1 ;
                 fi ;
      :: n==2 -> if
                 :: d1 = 1 ;
                 :: d2 = 1 ;
                 fi ;
      :: n==3 -> if
                 :: d1 = 1 ;
                 :: d2 = 1 ;
                 :: d3 = 1 ;
                 fi ;
      fi ;
    };
      k = 0 ;
      Sin! 1  ;

  ::  Sout?sInd ;

  ::  Rout?i,v -> atomic {
        k=k+1;
        if
        :: k==1 -> e1_ind = i ;
                   e1_val = v ;
        :: k==2 -> e2_ind = i ;
                   e2_val = v ;
        :: k==3 -> e3_ind = i ;
                   e3_val = v ;
        :: else -> assert(0) ;
        fi ;
        i=0 ;
        v=0 ;
      } ;
  od ;
} ;

proctype Sender()
{
  bit     ab ;
  byte    rc ;
  byte    i ;


  ab = 5 ;
  goto idle ;

idle:
  Sin?1  ;
  Sin?1  ;
  i = 1 ;
  goto next_frame ;

next_frame:
  atomic {
    skip ;
    if
    ::  i==1 -> K!(i==1),(i==n),ab,d1 ;
    ::  i==2 -> K!(i==1),(i==n),ab,d2 ;
    ::  i==3 -> K!(i==1),(i==n),ab,d3 ;
    ::  else -> assert(0) ;
    fi ;
  } ;
  rc = 0 ;
  goto wait_ack ;

wait_ack:
  if
  ::  L?1  ->
        ab = 1-ab ;
        goto success ;

  ::  ChunkTimeout?1  ->
        if
        ::  (rc < 2 )  -> rc=rc+1;
                          atomic {
                            if
                            ::  i==1 -> K!(i==1),(i==n),ab,d1 ;
                            ::  i==2 -> K!(i==1),(i==n),ab,d2 ;
                            ::  i==3 -> K!(i==1),(i==n),ab,d3 ;
                            ::  else -> assert(0) ;
                            fi ;
                          } ;
                          goto wait_ack ;
        ::  (rc >= 2 ) -> goto error ;
        fi ;
  fi ;

success:
  if
  ::  (i == n) -> Sout! 3  ;
                  goto idle ;
  ::  (i <  n) -> i=i+1;
                  goto next_frame ;
  fi ;

error:
  if
  ::  (i == n) -> Sout! 4  ;
  ::  (i != n) -> Sout! 5  ;
  fi ;
  SyncWait! 1  ;
  SyncWait?1  ;
  ab = 0 ;
  goto idle ;
} ;

proctype Receiver()
{
  bit    b1 ;
  bit    bN ;
  bit    ab ;
  byte   v ;
  bit    exp_ab ;

new_file:
  if
  ::  K?b1,bN,ab,v   ->
      goto first_safe_frame ;
  ::  SyncWait?1  ;
      SyncWait! 1  -> goto new_file ;
  fi ;

first_safe_frame:
  exp_ab = ab ;
  goto frame_received ;

frame_received:
  if
  ::  (ab != exp_ab) ->
        L! 1  ;
        goto idle ;

  ::  (ab == exp_ab) ->
        if
        ::  ( b1 && !bN) -> Rout! 1 ,v ;
        ::  (!b1 && !bN) -> Rout! 2 ,v ;
        ::  (        bN) -> Rout! 3 ,v ;
        fi ;
        goto frame_reported ;
  fi ;

frame_reported:
  L! 1  ;
  exp_ab = 1-exp_ab ;
  goto idle ;

idle:
  if
  ::  K?b1,bN,ab,v ->
        goto frame_received ;

  ::  SyncWait?1  ->
        if
        ::   bN -> skip ;
        ::  !bN -> Rout! 5,0  ;
        fi ;
        SyncWait! 1  ;
        goto new_file ;
  fi ;
} ;

proctype Daemon()
{
  bit   b ;
  bit   b1 ;
  bit   bN ;
  bit   ab ;
  byte  v ;

  do
  ::  K?b1,bN,ab,v -> ChunkTimeout! 1  ;
  ::  L?b          -> ChunkTimeout! 1  ;
  od ;
} ;

proctype Invariant()
{
  assert(((! checknow )||(  (((! (k>0)  )||(   (((! (k>=1) )||(  ((! (e1_ind != 5 ) )||
  (  (e1_val == d1) ))  ))  && ((! (k>=2) )||(  ((! (e2_ind != 5 ) )
  ||(  (e2_val == d2) ))  ))  && ((! (k>=3) )|| (  ((! (e3_ind != 5 ) )||
  (  (e3_val == d3) ))  ))   && ((! (n>1) )||(  (e1_ind == 1 ) ))   && ((! (k>2) )||
  (  (e2_ind == 2 ) ))   && (( ( (k == 1) && ((e1_ind == 3 ) || (e1_ind == 5 )) ) ||
  ( (k == 2) && ((e2_ind == 3 ) || (e2_ind == 5 )) ) || ( (k == 3) && ((e3_ind == 3 ) ||
  (e3_ind == 5 )) ) )  && ( ( (k == 1) && ((! (e1_ind == 3 ) )|| (  (1==n) ))  ) ||
  ( (k == 2) && ((! (e2_ind == 3 ) )||(  (2==n) ))  ) || ( (k == 3) && ((! (e3_ind == 3 ) )||
  (  (3==n) ))  ) )  && ( ( (k == 1) && ((! (e1_ind == 5 ) )||(  (1>1) ))  ) ||
  ( (k == 2) && ((! (e2_ind == 5 ) )||(  (2>1) ))  ) || ( (k == 3) && ((! (e3_ind == 5 ) )||
  (  (3>1) ))  ) ) )  && ( ( (k == 1) && ((! (sInd == 3 ) )|| (   (e1_ind == 3 ) ))  ) ||
  ( (k == 2) && ((! (sInd == 3 ) )||(   (e2_ind == 3 ) ))  ) || ( (k == 3) &&
  ((! (sInd == 3 ) )||(   (e3_ind == 3 ) ))  ) )  && ( ( (k == 1) && ((! (sInd == 5 ) )||
  (  (e1_ind == 5 ) ))  ) || ( (k == 2) && ((! (sInd == 5 ) )||(  (e2_ind == 5 ) ))  ) ||
  ( (k == 3) && ((! (sInd == 5 ) )|| (  (e3_ind == 5 ) ))  ) )  && ((! (sInd == 4 ) )||
  (   (k==n) ))  ) ))   && ((! (k==0) )|| (   (( ((sInd == 4 )  && (n == 1)) ||
  ((sInd != 4 )  && (n != 1)) )  && ( ((sInd == 5 ) && (n > 1))  ||
  ((sInd != 5 ) && (n <= 1)) ) ) ))  )  ))  ) ;
} ;

init {
  run Environment() ;
  run Sender() ;
  run Receiver() ;
  run Daemon() ;
  run Invariant() ;
}
```

Appendix E

Towers of Hanoi source code

E.1 Three discs

```
chan Hand = [1] of {int};

proctype Tower(int Id; int s1; int s2; int s3) {
    do
    :: {((s3 != 0) && (Id == 3))} assert 0;
    :: {((s3 != 0) && (Id != 3))} atomic{Hand!s3; s3 = 0;};
    :: {((s3 == 0) && (s2 != 0))} Hand?s3 {s3 < s2};
    :: {((s3 == 0) && (s2 != 0))} atomic{Hand!s2; s2 = 0;};
    :: {((s3 == 0) && (s2 == 0) && (s1 != 0))} Hand?s2 {s2 < s1};
    :: {((s3 == 0) && (s2 == 0) && (s1 != 0))} atomic{Hand!s1; s1 = 0;};
    :: {((s3 == 0) && (s2 == 0) && (s1 == 0))} Hand?s1;
    od;
};

init {
    run Tower(1,3,2,1);
    run Tower(2,0,0,0);
    run Tower(3,0,0,0);
}
```

E.2 Five discs

```
chan Hand = [1] of {int};

proctype Tower(int Id; int s1; int s2; int s3; int s4; int s5) {
    do
    :: {((s5 != 0) && (Id == 3))} assert 0;
    :: {((s5 != 0) && (Id != 3))} atomic{Hand!s5; s5 = 0;};
    :: {((s5 == 0) && (s4 != 0))} Hand?s5 {s5 < s4};
    :: {((s5 == 0) && (s4 != 0))} atomic{Hand!s4; s4 = 0;};
    :: {((s5 == 0) && (s4 == 0) && (s3 != 0))} Hand?s4 {s4 < s3};
    :: {((s5 == 0) && (s4 == 0) && (s3 != 0))} atomic{Hand!s3; s3 = 0;};
    :: {((s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 != 0))} Hand?s3 {s3 < s2};
    :: {((s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 != 0))} atomic{Hand!s2; s2 = 0;};
    :: {((s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 == 0) && (s1 != 0))} Hand?s2 {s2 < s1};
    :: {((s5 == 0) && (s4 == 0) && (s3 == 0) &&
        (s2 == 0) && (s1 != 0))} atomic{Hand!s1; s1 = 0;};
    :: {((s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 == 0) && (s1 == 0))} Hand?s1;
    od;
};

init {
    run Tower(1,5,4,3,2,1);
    run Tower(2,0,0,0,0,0);
    run Tower(3,0,0,0,0,0);
}
```

E.3 Seven discs

```
chan Hand = [1] of {int};

proctype Tower(int Id; int s1; int s2; int s3; int s4; int s5; int
s6; int s7) {
    do
    :: {((s7 != 0) && (Id == 3))} assert 0;
    :: {((s7 != 0) && (Id != 3))} atomic{Hand!s7; s7 = 0;};
    :: {((s7 == 0) && (s6 != 0))} Hand?s7 {s7 < s6};
    :: {((s7 == 0) && (s6 != 0))} atomic{Hand!s6; s6 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 != 0))} Hand?s6 {s6 < s5};
    :: {((s7 == 0) && (s6 == 0) && (s5 != 0))} atomic{Hand!s5; s5 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 != 0))} Hand?s5 {s5 < s4};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 != 0))} atomic{Hand!s4; s4 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 != 0))} Hand?s4 {s4 < s3};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) &&
        (s3 != 0))} atomic{Hand!s3; s3 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 == 0) &&
        (s2 != 0))} Hand?s3 {s3 < s2};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 == 0) &&
        (s2 != 0))} atomic{Hand!s2; s2 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 == 0) &&
        (s1 != 0))} Hand?s2 {s2 < s1};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 == 0) &&
        (s1 != 0))} atomic{Hand!s1; s1 = 0;};
    :: {((s7 == 0) && (s6 == 0) && (s5 == 0) && (s4 == 0) && (s3 == 0) && (s2 == 0) &&
        (s1 == 0))} Hand?s1;
    od;
};

init {
    run Tower(1,7,6,5,4,3,2,1);
    run Tower(2,0,0,0,0,0,0,0);
    run Tower(3,0,0,0,0,0,0,0);
}
```

Appendix F

Example .cod file of the unreliable communication example

```
0   LDLADDR     0   ; Start of proctype: Sender
1   XCHG
2   STORE
3   IRET            ; Parameter init done
4   LDLADDR     1   ; Start of assignment to: 'i'
5   PUSH        0
6   STORE           ; End of assignment to: 'i'
7   CHOICES     1   ; Start of DO statement
8   CHOICE      9
9   CHAN        SEND    LOC 0   ; Start CHOICE (do)
10  LDLVAR      1   ; Start of channel send (out)
11  SEND            ; End of channel send (out)
12  LDLADDR     1   ; Start of assignment to: 'i'
13  LDLVAR      1
14  PUSH        1
15  ADD
16  LDGVAR      0
17  MOD
18  STORE           ; End of assignment to: 'i'
19  JMP         7   ; Repeat (jump to start of do); End of DO statement
20  HALT            ; End of proctype: Sender
21  LDLADDR     0   ; Start of proctype: Receiver
22  XCHG
23  STORE
24  IRET            ; Parameter init done
25  CHOICES     1   ; Start of DO statement
26  CHOICE      27
27  CHAN        RECV    LOC 0   ; Start CHOICE (do)
28  RECEIVE         ; Start of channel receive (in)
29  SETPP
30  LDLADDR     1
31  XCHG
32  STORE           ; Communication to variable: 'j'
33  CLRPP           ; End of channel receive (in)
34  LDLVAR      1   ; Start of assert statement
35  LDLVAR      2
```

```
36  EQ
37  ASRT             ; End of assert statement
38  LDLADDR     2    ; Start of assignment to: 'k'
39  LDLVAR      2
40  PUSH        1
41  ADD
42  LDGVAR      0
43  MOD
44  STORE            ; End of assignment to: 'k'
45  JMP         25   ; Repeat (jump to start of do); End of DO statement
46  HALT             ; End of proctype: Receiver
47  LDLADDR     0    ; Start of proctype: Daemon
48  XCHG
49  STORE
50  IRET             ; Parameter init done
51  CHOICES     1    ; Start of DO statement
52  CHOICE      53
53  CHAN        RECV    LOC 0   ; Start CHOICE (do)
54  RECEIVE          ; Start of channel receive (in)
55  SETPP
56  LDLADDR     1
57  XCHG
58  STORE            ; Communication to variable: 'k'
59  CLRPP            ; End of channel receive (in)
60  JMP         51   ; Repeat (jump to start of do); End of DO statement
61  HALT             ; End of proctype: Daemon
62  SETA             ; Start of @Init
63  LDGADDR     0    ; Start of global variable init
64  PUSH        16
65  STORE
66  PUSH        0    ; End of global variable init; Push channel ID of GLOBAL channel : 'c'
67  RUN         1
68  PUSH        0    ; Push channel ID of GLOBAL channel : 'c'
69  RUN         2
70  PUSH        0    ; Push channel ID of GLOBAL channel : 'c'
71  RUN         3
72  CLRA
73  HALT             ; End of @Init
```

Appendix G

Unreliable communication: Spin output

G.1 Code

```
int MAX = 16;
chan c = [1] of {byte};

proctype Daemon (chan in) {
  byte k;
  do :: in?k; od;
};

proctype Receiver (chan in) {
  byte j; byte k;
  do
  :: in?j; assert(j == k); k = (k+1) % MAX;
  od;
};

proctype Sender (chan out) {
  byte i; i = 0;
  do
  :: out!i; i = (i+1) % MAX;
  od;
};

init {
    atomic {
        run Daemon (c);
    run Receiver (c);
        run Sender (c);
    }
}
```

G.2 Result: First error

```
pan: assertion violated (j==k) (at depth 93)
```

```
pan: wrote pan_in.trail
(Spin Version 3.4.2 -- 28 October 2000)
Warning: Search not completed

Full statespace search for:
    never-claim             - (not selected)
    assertion violations    +
    cycle checks        - (disabled by -DSAFETY)
    invalid endstates   +

State-vector 40 byte, depth reached 92, errors: 1
      91 states, stored
       1 states, matched
      92 transitions (= stored+matched)
       2 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

4.302   memory usage (Mbyte)


real        0.2
user        0.0
sys         0.0
```

G.3 Result: Shortest path

```
pan: assertion violated (j==k) (at depth 93)
pan: wrote pan_in.trail
pan: reducing search depth to 92
.
.
pan: wrote pan_in.trail
pan: reducing search depth to 10
pan: wrote pan_in.trail
pan: reducing search depth to 9
(Spin Version 3.4.2 -- 28 October 2000)

Full statespace search for:
    never-claim             - (not selected)
    assertion violations    +
    cycle checks        - (disabled by -DSAFETY)
    invalid endstates   +

State-vector 40 byte, depth reached 92, errors: 68
     505 states, stored
     559 states, matched
    1064 transitions (= stored+matched)
       2 atomic steps
hash conflicts: 3 (resolved)
(max size 2^19 states)

4.302   memory usage (Mbyte)
```

```
real      14.3
user       0.0
sys        0.0
```

G.3.1 Shortest path trace

```
preparing trail, please wait...done
spin: dataflow optimizations turned off
spin: dead variable elimination turned off
spin: statement merging turned off
spin: warning -o[123] option ignored in simulations
  1:    proc  0 (:init:) line  25 "pan_in" (state 1)    [(run Daemon(c))]
  2:    proc  0 (:init:) line  26 "pan_in" (state 2)    [(run Receiver(c))]
  3:    proc  0 (:init:) line  27 "pan_in" (state 3)    [(run Sender(c))]
  4:    proc  3 (Sender) line  17 "pan_in" (state 1)    [i = 0]
  5:    proc  3 (Sender) line  19 "pan_in" (state -)    [values: 1!0]
  5:    proc  3 (Sender) line  19 "pan_in" (state 2)    [out!i]
  6:    proc  3 (Sender) line  19 "pan_in" (state 3)    [i = ((i+1)%MAX)]
  7:    proc  1 (Daemon) line   6 "pan_in" (state -)    [values: 1?0]
  7:    proc  1 (Daemon) line   6 "pan_in" (state 1)    [in?k]
  8:    proc  3 (Sender) line  19 "pan_in" (state -)    [values: 1!1]
  8:    proc  3 (Sender) line  19 "pan_in" (state 2)    [out!i]
  9:    proc  2 (Receiver) line  12 "pan_in" (state -)  [values: 1?1]
  9:    proc  2 (Receiver) line  12 "pan_in" (state 1)  [in?j]
spin: line  12 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((j==k))
#processes: 4
 10:    proc  3 (Sender) line  19 "pan_in" (state 3)
 10:    proc  2 (Receiver) line  12 "pan_in" (state 2)
 10:    proc  1 (Daemon) line   6 "pan_in" (state 2)
 10:    proc  0 (:init:) line  29 "pan_in" (state 5)
4 processes created
Exit-Status 0
```

Appendix H

Bounded retransmission protocol: Spin output

H.1 Results

```
Depth=  213260 States=   1e+06 Transitions= 2.03187e+06 Memory= 344.860
Depth=  213260 States=   2e+06 Transitions= 4.1921e+06 Memory= 400.875
Depth=  213260 States=   3e+06 Transitions= 6.26591e+06 Memory= 459.041
Depth=  213260 States=   4e+06 Transitions= 8.48133e+06 Memory= 519.049
Depth=  213260 States=   5e+06 Transitions= 1.06532e+07 Memory= 579.160
Depth=  213260 States=   6e+06 Transitions= 1.36607e+07 Memory= 639.169
Depth=  213260 States=   7e+06 Transitions= 1.68512e+07 Memory= 699.178
(Spin Version 3.4.2 -- 28 October 2000)

Full statespace search for:
    never-claim            - (not selected)
    assertion violations    +
    cycle checks        - (disabled by -DSAFETY)
    invalid endstates   +

State-vector 92 byte, depth reached 213260, errors: 0
7.4001e+06 states, stored
1.08139e+07 states, matched
1.8214e+07 transitions (= stored+matched)
2.18253e+07 atomic steps
hash conflicts: 1.19363e+07 (resolved)
(max size 2^26 states)

Stats on memory usage (in Megabytes):
710.410 equivalent memory usage for states (stored*(State-vector + overhead))
434.603 actual memory usage for states (compression: 61.18%)
    State-vector as stored = 55 byte + 4 byte overhead
268.435 memory used for hash-table (-w26)
20.000  memory used for DFS stack (-m1000000)
723.141 total actual memory usage

real     6:58.9
user     6:39.3
sys        18.1
```

Appendix I

Dining philosophers: Spin output

I.1 Code

```
/* Dining Philosophers Problem (MAJoR & PROMELA Compatible) */
/* Michel Rosien */

chan chan1 = [1] of {int};
chan chan2 = [1] of {int};
chan chan3 = [1] of {int};
chan chan4 = [1] of {int};
chan chan5 = [1] of {int};

int FORK = 1;

proctype place_forks() {
    atomic {
        chan1!FORK;
        chan2!FORK;
        chan3!FORK;
        chan4!FORK;
        chan5!FORK;
    };
};

proctype philosopher(int left_hand; int right_hand; chan left_fork; chan right_fork)  {
    if /* Take forks */
    ::left_fork?left_hand; right_fork?right_hand;
    ::right_fork?right_hand; left_fork?left_hand;
    fi;

    skip; /* Eat */

    if /* Drop forks */
    :: left_fork!left_hand; right_fork!right_hand;
    :: right_fork!right_hand; left_fork!left_hand;
    fi;
};

init {
    atomic{
    run philosopher(0, 0, chan5, chan1);
    run philosopher(0, 0, chan4, chan5);
    run philosopher(0, 0, chan3, chan4);
    run philosopher(0, 0, chan2, chan3);
    run philosopher(0, 0, chan1, chan2);
    run place_forks();
    }
}
```

I.2 Output

```
pan: invalid endstate (at depth 17)
pan: wrote pan_in.trail
pan: reducing search depth to 16
(Spin Version 3.4.2 -- 28 October 2000)

Full statespace search for:
    never-claim            - (not selected)
    assertion violations    +
    cycle checks        - (disabled by -DSAFETY)
    invalid endstates   +

State-vector 128 byte, depth reached 43, errors: 1
    6216 states, stored
   17364 states, matched
   23580 transitions (= stored+matched)
       9 atomic steps
hash conflicts: 0 (resolved)
(max size 2^26 states)

289.357 memory usage (Mbyte)


real        4.7
user        1.8
sys         2.3
```

I.3 Trace of shortest path

```
preparing trail, please wait...done
spin: dataflow optimizations turned off
spin: dead variable elimination turned off
spin: statement merging turned off
spin: warning -o[123] option ignored in simulations
  1:    proc  0 (:init:) line  38 "pan_in" (state 1)    [(run philosopher(0,0,chan5,chan1))]
  2:    proc  0 (:init:) line  39 "pan_in" (state 2)    [(run philosopher(0,0,chan4,chan5))]
  3:    proc  0 (:init:) line  40 "pan_in" (state 3)    [(run philosopher(0,0,chan3,chan4))]
  4:    proc  0 (:init:) line  41 "pan_in" (state 4)    [(run philosopher(0,0,chan2,chan3))]
  5:    proc  0 (:init:) line  42 "pan_in" (state 5)    [(run philosopher(0,0,chan1,chan2))]
  6:    proc  0 (:init:) line  43 "pan_in" (state 6)    [(run place_forks())]
  7:    proc  6 (place_forks) line  14 "pan_in" (state -)   [values: 2!1]
  7:    proc  6 (place_forks) line  14 "pan_in" (state 1)   [chan1!FORK]
  8:    proc  6 (place_forks) line  15 "pan_in" (state -)   [values: 5!1]
  8:    proc  6 (place_forks) line  15 "pan_in" (state 2)   [chan2!FORK]
  9:    proc  6 (place_forks) line  16 "pan_in" (state -)   [values: 4!1]
  9:    proc  6 (place_forks) line  16 "pan_in" (state 3)   [chan3!FORK]
 10:    proc  6 (place_forks) line  17 "pan_in" (state -)   [values: 3!1]
 10:    proc  6 (place_forks) line  17 "pan_in" (state 4)   [chan4!FORK]
 11:    proc  6 (place_forks) line  18 "pan_in" (state -)   [values: 1!1]
 11:    proc  6 (place_forks) line  18 "pan_in" (state 5)   [chan5!FORK]
 12: proc 6 terminates
 13:    proc  5 (philosopher) line  24 "pan_in" (state -)   [values: 2?1]
 13:    proc  5 (philosopher) line  24 "pan_in" (state 1)   [left_fork?left_hand]
 14:    proc  4 (philosopher) line  24 "pan_in" (state -)   [values: 5?1]
 14:    proc  4 (philosopher) line  24 "pan_in" (state 1)   [left_fork?left_hand]
 15:    proc  3 (philosopher) line  24 "pan_in" (state -)   [values: 4?1]
 15:    proc  3 (philosopher) line  24 "pan_in" (state 1)   [left_fork?left_hand]
 16:    proc  2 (philosopher) line  24 "pan_in" (state -)   [values: 3?1]
 16:    proc  2 (philosopher) line  24 "pan_in" (state 1)   [left_fork?left_hand]
 17:    proc  1 (philosopher) line  24 "pan_in" (state -)   [values: 1?1]
 17:    proc  1 (philosopher) line  24 "pan_in" (state 1)   [left_fork?left_hand]
spin: trail ends after 17 steps
#processes: 6
 17:    proc  5 (philosopher) line  24 "pan_in" (state 2)
 17:    proc  4 (philosopher) line  24 "pan_in" (state 2)
 17:    proc  3 (philosopher) line  24 "pan_in" (state 2)
 17:    proc  2 (philosopher) line  24 "pan_in" (state 2)
 17:    proc  1 (philosopher) line  24 "pan_in" (state 2)
 17:    proc  0 (:init:) line  45 "pan_in" (state 8)
7 processes created
Exit-Status 0
```

Appendix J

Statistics

Total nr or files:	30
Total lines of code:	10770
Total bytes of code:	313089
Size of executable:	1166390

J.1 Software

Compiler:	GNU G++ version 2.95.3
Make:	GNU Make version 3.76.1
Text Editor:	GNU Emacs 20.4.1
Operating system:	SunOS Release 5.6 Version Generic_105181-16 [UNIX(R) System V Release 4.0]

J.2 Hardware

Manufacturer:	Sun (Sun Microsystems)
System Model:	Enterprise E3500
Main Memory:	4.0 GB
ROM Version:	OBP 3.2.21 1999/02/19 14:33
Number of CPUs:	6
CPU Type:	sparc

J.2.1 Files

```
-rw-r--r--   1 rosien   stud        8022 Jan 26 13:47 code_vector.hpp
-rw-r--r--   1 rosien   stud         629 Jan 26 11:18 defines.hpp
-rw-r--r--   1 rosien   stud       27635 Mar  4 21:23 emit_code.cpp
-rw-r--r--   1 rosien   stud         164 Jan 26 11:19 emit_code.hpp
-rw-r--r--   1 rosien   stud        3506 Mar  1 10:47 grammar.h
-rw-r--r--   1 rosien   stud       19979 Jan 26 11:18 grammar.y
-rw-r--r--   1 rosien   stud        6221 Jan 26 11:19 instructions.cpp
-rw-r--r--   1 rosien   stud       18585 Jan 26 11:18 instructions.hpp
-rw-r--r--   1 rosien   stud         329 Jan 26 11:17 lexer.h
-rw-r--r--   1 rosien   stud        4368 Jan 26 11:18 lexer.y
-rw-r--r--   1 rosien   stud       14922 Mar  4 21:23 main.cpp
-rw-r--r--   1 rosien   stud         206 Jan 26 11:19 mem_debug.hpp
-rw-r--r--   1 rosien   stud       23659 Mar  6 09:35 model_check.cpp
-rw-r--r--   1 rosien   stud        1689 Mar  4 21:23 model_check.hpp
-rw-r--r--   1 rosien   stud         186 Jan 26 11:16 parser.h
-rw-r--r--   1 rosien   stud        5075 Jan 26 11:19 parsetree.cpp
```

```
-rw-r--r--    1 rosien   stud        55402 Jan 26 11:17 parsetree.hpp
-rw-r--r--    1 rosien   stud         1648 Jan 26 11:17 symbol.hpp
-rw-r--r--    1 rosien   stud         3160 Jan 26 11:17 symtable.hpp
-rw-r--r--    1 rosien   stud        45651 Mar  1 16:32 system.cpp
-rw-r--r--    1 rosien   stud         7596 Jan 29 17:02 system.hpp
-rw-r--r--    1 rosien   stud         1554 Jan 30 13:59 temp.cpp
-rw-r--r--    1 rosien   stud         2644 Jan 26 11:19 tree_check_declare.cpp
-rw-r--r--    1 rosien   stud         1724 Jan 26 11:19 tree_check_label.cpp
-rw-r--r--    1 rosien   stud         1832 Jan 26 11:19 tree_check_use.cpp
-rw-r--r--    1 rosien   stud        19276 Jan 26 13:02 tree_execute.cpp
-rw-r--r--    1 rosien   stud         9717 Jan 26 11:19 tree_size.cpp
-rw-r--r--    1 rosien   stud         7176 Jan 26 11:19 tree_type_check.cpp
-rw-r--r--    1 rosien   stud        14171 Jan 26 11:19 types.cpp
-rw-r--r--    1 rosien   stud         6363 Jan 26 11:17 types.hpp
```

Bibliography

[1] *Borland C++ Programmer's Guide. Version 4.5.* Borland International, Inc., 100 Borland Way. P.O. Box 660001, Scotts Valley, CA 95067-0001.

[2] *ON-THE-FLY, LTL MODEL CHECKING with SPIN.* http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[3] *PROMELA Language reference.* http://cm.bell-labs.com/cm/cs/what/spin/Man/promela.html.

[4] *Standard Template Library Programmer's Guide.* http://www.sgi.com/tech/stl/.

[5] *String hash function.* http://www-scf.usc.edu/ csci201/notes/hashing/sld007.htm.

[6] Rajive Bagrodia. Synchronization of Asynchronous Processes in CSP. University of Califonia at Los Angeles *http://dev.acm.org/pubs/articles/journals/toplas/1989-11-4/p585-bagrodia/p585-bagrodia.pdf.*

[7] Ed Brinksma, editor. *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science (LNCS)*, University of Twente, Enschede, The Netherlands, April 1997. Springer-Verlag, Berlin.

[8] Edmund M. Clarke and E. Allen Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In Dexter Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science (LNCS)*, pages 52–71. Springer-Verlag, Berlin, 1981.

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

[10] Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and G. Jan Tretmans. The Bounded Retransmission Protocol must be on Time! In Brinksma [7], pages 416–431.

[11] E. Dijkstra. *A Discipline of Programming.* 1976.

[12] Jaco Geldehuys. Efficiency Issues in the Design of a Model Checker. Master's thesis, University of Stellenbosch.

[13] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[14] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[15] Joost-Pieter Katoen. Concepts, Algorithms and Tools for Model Checking. Technical Report Volume 32, Number 1, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich-Alexander-Universität, Erlangen, Nürnberg, Germany, June 1999.

[16] Don Knuth. *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley Publishing Company, 1998.

[17] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science (LNCS)*, pages 337–351. Springer-Verlag, Berlin, 1982.

Index