Ricardo João Besteiro e Silva

A Behavioral Analysis Tool for Models of Software Systems

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Lisboa, 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

A Behavioral Analysis Tool for Models of Software Systems

Ricardo João Besteiro e Silva (aluno nº 28025)

1º Semestre de 2009/10

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

A Behavioral Analysis Tool for Models of Software Systems

Ricardo João Besteiro e Silva (aluno nº 28025)

Orientador: Prof. Doutor Luís Caires

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

1º Semestre de 2009/10

Acknowledgements

First and foremost I would like to thank Professor Luís Caires for his support and patience throughout the elaboration of this thesis. I would also like to thank my colleagues Bernardo Toninho and Hugo Vieira for all the times they helped me by providing further insight in the tool they are also familiar with.

Last, but in any means not least, I would like to thank all my friends, family and specially Luísa Lourenço for the not so technical but still important support they provided which made this thesis possible.

Resumo

Cálculos de processos são linguagens simples que permitem a modelação de sistemas concorrentes de forma a poder ser verificada a sua correcção. É possível usar cálculos de processos para analisar sistemas concorrentes comparando a representação de uma implementação com uma especificação mais geral usando alguma forma de equivalência, ou através da verificação se algumas propriedades descritas numa lógica adequada são verdadeiras.

Bisimulação forte é uma de muitas relações de equivalência definidas para comprar cálculos de processos. Esta relação de equivalência considera processos que têm o mesmo comportamento, i.e. realizam as mesmas transições, como equivalentes sem olhar para os detalhes de implementação.

Há vários tipos de lógicas para raciocinar sobre processos que vão desde lógicas temporais – que descrevem como é que as propriedades evoluem ao longo da vida de um processo – lógicas comportamentais – que descrevem que transições é que um processo pode fazer – e lógicas espaciais – que descrevem a estrutura dos componentes de um sistema e como estão ligados.

Verificação de modelos trata-se de verificar se um modelo, que no nosso caso são processos, satisfaz uma dada propriedade lógica. A verificação de modelos em conjunção com cálculos de processos são meios bastante populares para a verificação da correcção de sistemas concorrentes.

Nesta tese endereçamos o problema de verificar a bisimilaridade entre processos usando fórmulas características, que são fórmulas que descrevem na totalidade o comportamento de um processo. Implementámos a possibilidade de verificar bisimilaridades na ferramenta Spatial Logic Model Checker. Como resultado desta implementação também estendemos a lógica da ferramenta com uma modalidade extra. Outras funcionalidades que adicionámos à ferramenta como resultado deste trabalho foram a possibilidade de definir propriedades recursivas e a adição de uma *cache* ao algoritmo de verificação para evitar redundância.

Palavras-chave: algoritmo de bisimulação, verificação de modelos, cálculos de processos, fórmula característica

Abstract

Process calculi are simple languages which permit modeling of concurrent systems so that they can be verified for correctness. We can analyze concurrent systems based on process calculi by either comparing a representation of the actual implementation with a simpler specification for equivalence, or by verifying whether desired properties described in an adequate logic hold.

Strong bisimulation equivalence is one of many equivalence relations defined on process calculi to aid in the verification of concurrent software. This equivalence relation relates processes which exhibit the same behavior, i.e. perform the same transitions, as equivalent regardless of internal implementation details.

Logics to reason about processes range from those which describe temporal properties – how properties evolve during the course of a process' life – behavioral properties – which actions a process is capable of performing – and spatial properties – what components compose a process and how are they connected.

Model checking consists of verifying if a model, in our case a process, satisfies a given property. Model checking techniques are quite popular in conjunction with process calculi to aid in the verification of the correctness of concurrent systems.

In this thesis we address the problems of checking bisimilarity between processess using characteristic formulae, which are formulae used to fully describe a process' behavior. We implement some facilities to allow bisimilarity verification in the Spatial Logic Model Checker tool. As a result of adding these facilities we also extend the SLMC tool with an extra modality in the logic it uses to reason about processes. We have also added the possibility to define mutually recursive properties in the tool and enhanced the model checking algorithm with a cache to prevent redundant, time-consuming checks to be performed.

Keywords: bisimulation algorithm, model checking, process calculi, characteristic formula

Contents

1 Introduction **1**
1.1 Motivation and Contributions 3
1.2 Document Structure 3

2 Context **5**
2.1 Motivation 5
2.2 Process Calculi 6
2.2.1 CCS 7
2.2.2 π-calculus 8
2.2.3 Bisimulation 10
2.3 Logics 12
2.3.1 Hennesy-Milner Logic 12
2.3.2 μ-calculus 13
2.4 Model Checking 14
2.5 Techniques for bisimulation checking 14
2.6 Tools which support bisimulation checking 15
2.7 Spatial Logic Model Checker 16

3 Verifying Bisimilarity using Model Checking **17**
3.1 Building Characteristic Formulae for CCS Processes 17
3.1.1 Finite Processes 18
3.1.1.1 Examples 19
3.1.2 Finite State Processes 20
3.1.2.1 Examples 21
3.2 Challenges 24

4 Spatial Logic Model Checker **25**
4.1 The Spatial Logic Model Checker 25
4.1.1 Syntax of Processes 25
4.1.2 Syntax of Formulae 25
4.1.3 Top Level Commands 27
4.1.3.1 Defining Processes 27
4.1.3.2 Defining Properties 29
4.1.3.3 Checking Properties of Processes 30
4.1.3.4 Additional Commands 30
4.2 Extensions to the Tool 31
4.2.1 Process Stepping 31
4.2.2 Verifying Bisimilarity 33
4.2.3 Extensions to the Logic 33

4.2.3.1 The `proc` operator 33
4.2.3.2 The <[]> modality 34
4.2.3.3 Recursive Formulae 34
4.3 Implementation Details 35
4.3.1 Checking the <[]> modality 35
4.3.2 Recursive Formulae 37
4.3.3 Caching in the Model Checker 38
4.3.4 Building a Characteristic Formula 40
4.4 Evolution 41

5 The Jobshop Example 44
5.1 Description 44
5.2 Implementation 44
5.3 Verifications 46
5.3.1 Positive Example 46
5.3.2 Negative Example 47

6 Closing Remarks 49
6.1 Future Work 50

List of Figures

2.1 Language Constructs of Milner's CCS 8
2.2 Semantics for composition in CCS 8
2.3 LTS for a simple CCS process 9
2.4 The π-calculus 9
2.5 A simple example of mobility 10
2.6 Example of weak bisimulation 11
2.7 Hennessy-Milner logic 13
2.8 Hennessy-Milner Logic's semantics 13

3.1 The LTS for process `a|b`. 19
3.2 Formulae for characterizing `a|b`. 19
3.3 Semaphores 21
3.4 LTS for `SemSpec`. 22
3.5 Characteristic formula for `SemSpec`. 22
3.6 LTS for process `S`. 23
3.7 Erroneous formula for `S`. 24

4.1 Syntax of SLMC processes. 26
4.2 Syntax of Formulae in SLMC. 28
4.3 Example usage of the `step` command. 32

1. Introduction

Computers are a kind of general purpose machine which is nowadays present in most of the aspects of our daily life. This general purpose machine can be used to fulfil many tasks be it the creation of a text document like this thesis, analyzing big sets of numerical data, creating pictures or even fireworks choreographies. The reason computers are so versatile is because we can make custom sets of instructions, which we call programs, which direct what the computer is performing. Most computers are only able to understand a very simple set of instructions which algebraically manipulate a small amount of numerical resources. Although it seems unlikely, this simple language the computer uses is the ultimate responsible for the wide array of things which can be accomplished using a computer.

While this language has the potential to be able to describe all actions and tasks which can be performed by a computer, it is not a language with which most programmers are comfortable using. This language is very low level and to increase the productivity of programmers (the rate at which they are able to make the computer do what they need it to do) several other higher level languages were developed. The first programming languages developed simply abstracted the way to perform mathematical operation by allowing the expressions to be expressed in a mathematical manner instead of by the steps taken to perform the computation. Since then several languages have emerged which deal with computation in their own way. Languages today range from iterative languages, which describe which steps to take in order to get a job done, to functional programming languages where we describe the problem we are facing using the abstraction of a mathematical function, and also logical languages where it is the problem which is described and not how to solve it. All these languages focus on making it easy for the programmer to express his intentions in varying degrees of abstraction, and come equipped with some way of translating what the programmer said in that language to the language the computer understands. We call the programs which transform one language the programmer understands to the language the computer understands a compiler.

Computer programming is a task where many mistakes occur. Even experienced programmers can make mistakes which given the complexity of the system they are developing are hard to spot. Since errors can occur, programs need to be tested for correctness before they are deployed to do their task. Most errors have a relatively small impact or are just annoying, other errors in more critical system can have dire consequences and there is really no way they can be afforded. There are many ways in which we can analyze a program and try to provide some safety about its correctness. We can classify the techniques used to achieve this goal in two major categories: dynamic and static analysis.

Dynamic analysis occurs when we actually run the program and observe its behavior in search of abnormal situations. One example of this kind of analysis is unit testing where the parts of the program are tested individually by other programs which attempt to certify that the program is running correctly, or that certain misbehaviors are not present.

Static analysis happens when we try to analyze a program or a computer system without

actually having to run it. The most pervasive use of static analysis comes in the form of type systems, which are present in almost all mainstream programming languages. What type systems do is trying to figure out the kind of each expression in the program and assert that it conforms to certain rules. Most commonly type systems are used to make simple lightweight checks on a program's source code to identify errors like using a string variable in an arithmetic expression or calling a non-existent method from an object. There are, however, more complex type systems which can, for example, verify that a software component has all the external dependencies it needs to operate or that a service in a web-application is communicating correctly with the other services in that application.

With the advent of the Internet and multi-core processors being a widespread commodity we have witnessed a great deal of interest in what are called concurrent and distributed systems. The main difference between these systems and the sequential systems which were in place before is that these systems are compromised of one or more sequential systems running seemingly, or actually, at the same time and communicating with each other. This interleaving of the program execution can introduce errors which are not easily caught just by analyzing each program individually. In order to catch errors which derive from the concurrent nature of these systems we need to be able to reason about the system as a whole. This new programming environment introduces errors which did not occur so often in sequential programs like communications being skewed from a delay in message transmission or some parts of the system entering a deadlocked state where it cannot progress.

As argued, the techniques used to reason about the correctness of sequential programs cannot help with all the issues involved in analyzing a concurrent system. While the standard techniques for program analysis will help with the elimination of errors of individual components in the system, asserting the correctness of the system as a whole is a much more delicate matter which required the creation of new analysis techniques.

One of the kind of tools which was introduced to help with this problem are the so called process calculi like Calculus of Communicating Systems and the π-calculus. These calculi provide small languages in which to represent the core aspects of the concurrent system (usually, the communications performed) in such a way that a precise analysis can be made on the system's properties. We use these languages to build a model of the actual system we are analyzing and then we can either compare the behavior of the system we described with the description of the ideal system we want to implement to assert its correctness. Another way we can use these calculi to verify the correctness of a system is by having an appropriate logic which we can use to define interesting properties over the model of our system. The latter technique is called Model Checking.

The Spatial Logic Model Checker is a model checking tool developed by Hugo Vieira and Luís Caires which uses a π-calculus like language to define the model of the software system and a spatial logic to verify the existence or absence of desired or undesired properties in the system. The logic used in this tool is capable of not only reasoning about the behavior of the system (what actions it can perform) but also about the composition of the system (the properties of the various parts which compose it).

1.1 Motivation and Contributions

The SLMC tool provides a powerful logic to reason about the behavior and composition of concurrent systems. With this logic we can check if systems are composed of the right components and that these components are doing what they should be doing. If we need to check very intricate and complex behavior, we will be faced with large formulae which are hard to reason about for themselves, so we can end up having errors in our approach to eliminate errors. For some kinds of properties it is a much more direct approach to specify some other process with the intended behavior and have some way of saying that the system we are analysing should behave like that process for all practical reasons.

The main focus of the work in this thesis is granting SLMC with the capacity to perform the kind of behavioral analysis we just described. We wish to be able to check if a process exhibits the exact same behavior as some other process. Instead of using the more classic approach to this issue which is using a partition refinement algorithm to sort out which states are equivalent in both processes, we took a different route. We use the notion of a characteristic formula to implement our process equivalence test. A characteristic formula is a property in the model checker which is capable of identifying processes which exhibit the same behavior as the one which originated the formula and distinguishing processes which do not. The main advantage of using these characteristic formulae is that, unlike the method based on a special algorithm for comparing the processes, it can be used as a way of enhancing the logic's expressiveness which will allow us to express properties about systems in a more meaningful and natural manner.

To sum up, the contributions of this work are:

- We introduced the possibility for SLMC to relate two processes using strong bisimilarity
- SLMC now allows the use of a process in a formula to mean the process' full behaviour
- We have implemented an equation system which allows the definition of mutually recursive formulae, enhancing the tool's expressiveness and usability (you no longer have to use fixed points to define certain kinds of formulae)
- We have extended the logic with a new behavioral modality which provides a set of properties that a process' continuation by some action must be able to fulfil
- We have implemented a cache in our model checking allowing the speedup of the verification of big repetitive formulae

1.2 Document Structure

On the remainder of this document we will be introducing the motivation for static analysis and some techniques used for it in chapter 2, where we will also present some tools which also do what we propose to do with this work, only in a different way. In chapter 3 we will describe the

process of how to build a characteristic formula and provide some discussion on the limitations and challenges presented by this approach to equivalence checking. On chapter 4, we provide an overview of the SLMC tool as it was and also discuss the changes made to it in the course of the work presented in this thesis. We also discuss some of the implementation details of the new features and the evolution of the solution throughout the duration of this thesis. Chapter 5 contains a more elaborate example usage of our system than the examples we use to illustrate in the chapters describing the system. In chapter 6 we draw out some conclusions taken from our work and outline some ideas about work which can be made to improve upon what we done.

2 . Context

In this chapter we introduce motivation for our work by relating it with its practical uses. We will describe methods for checking concurrent systems and elaborate on process calculi and logics for process calculi. We introduce the notion of model checking and bisimulation, and describe algorithms used for the practical implementation of checking whether or not processes are bisimilar. Also we will list a few tools which support bisimulation . Finally we will briefly describe the tool where we will implement the ideas developed in this thesis.

2.1 Motivation

Software verification consists of checking whether or not an implementation of a system by a computer program does what it is supposed to do or, in other words, if it conforms to its specification. In this work we are mainly concerned with the verification of concurrent systems which because of their inherent non-determinism are notably harder to reason about, and thus verify, than sequential systems.

Techniques for verification of concurrent processes, which we will describe later, range from practical to formal. Formal methods analyze a system by translating it to a mathematical description and reasoning about its properties, possibly using specially tailored logics, with strict mathematical rigor. Practical approaches are primarily based on running the system and attempting to detect abnormal behavior. As concurrent systems are non-deterministic by nature, the practical approach may not be able to detect problems which only arise at very specific interleavings of communications or situations. On the other hand reasoning formally about the system will provide with an in-depth analysis of what problems may arise. Not only this analysis identifies problems, but it will also provide with a starting point for solving them.

Sequential systems are usually tested using a series of programs which run the system testing whether the program responds as intended in several desired situations. It is obvious that this approach can also be used to verify concurrent systems simply by creating test programs which interact with the system and check if the result is the expected correct one. By extending this approach to concurrent systems it is desirable that the tests be run several times because some problems may not manifest in a single run of the system but rather when certain, rarer, conditions arise. Devising tests for concurrent systems is usually a very painstaking enterprise since the test must be able to account for a very large number of potential interactions as well as their interleavings. The usual procedure for detecting problems that arise from unaccounted for interleavings is to run all the tests a great number of times and expect that in some of them the problems arise so developers are able to identify and correct them.

A more modern approach to program testing is by using a technique called Unit Testing[2] where a program is divided in several relatively small *units* which are tested individually for correctness. A unit can be a function, a procedure, a class or whatever seems appropriate in the language, model or framework being used. Unit testing has the advantage of, when

done properly, being able to more easily identify on which module a program has problems. Unit Testing encourages to test as often as possible, since tests are usually small, in order to detect early cases when changes affect existing code. This approach can also be used to verify concurrent systems using the same cares as in the previously described approach.

A more formal approach involves using some kind of automatic validity checking for desired properties in the program. The Floyd-Hoare Logic[17, 12] was defined for sequential programs but can be applied to verifying concurrent systems. A Hoare triple, the basis for the logic, is a statement of the sort {P}C{Q}, where P and Q are logical formulae and C is a program statement. A Hoare triple like the one presented above states that after completing statement C, starting from an environment satisfying P, we will reach an environment satisfying Q. We call P the preconditions for and Q the post-conditions of the application of C. Using Hoare logic one can reason formally about the flow of a program and check whether, given certain preconditions, some conditions are right to be expected to hold. This approach can be used to reason about concurrent processes if enough care is taken in the definition of the conditions and if the programmers code in a disciplined manner. This approach usually requires a large amount of annotations in the source code in order to guide the checking algorithm through the proof of the desired properties. These extensive annotations in the source code are required so that the algorithm is able to verify the program's correctness in a reasonable amount of time, making this approach very dependent on human intervention.

We can verify if a system exhibits interesting properties by translating it to a formal language and checking whether the desired properties hold in the simplified model. Formal languages are more concise than programming languages because they are more focussed on the essential aspects of the paradigm in use and thus contain less clutter, which makes them easier to reason about than directly using the source language as in the Floyd-Hoare Logic approach. These simple languages used for modeling are usually algebras which attempt to capture the essence of the programming paradigm at hand. When we wish to reason about concurrent systems we use process calculi like the ones discussed in the next section. These calculi are capable of expressing the interactions between the system's components and, by comparing the system's representation with an ideal model or checking if certain properties are satisfied by it, one is able to reason about the system's correctness. The comparison with an ideal model, a specification, is done by assessing whether the interactions possible in the specification are the same the implementation is capable of performing, normally using a notion of equivalence called bisimulation. One can also verify whether certain properties hold by checking if the structure of the system sustains properties described in a logic defined for describing the behavior of processes, a modal logic.

2.2 Process Calculi

Analyzing a system using formal methods usually consists of translating the system to a formal language for which interesting properties can be defined and checked against. In the case when

we wish to model concurrent systems we turn to process algebras or calculi like CCS [21], CSP [18], or the π-calculus [22], to serve as our modeling languages.

After having a representation of our system in a process calculus it is necessary to define properties to reason about it. In process algebras it is customary to prove systems to be correct by comparing them to "ideal" specifications of their behavior. To perform this comparison we use appropriate equivalence relations like structural congruence (a process can be rewritten by algebraic manipulation of the other) or bisimilarity [24] (a process exhibits the same observable behavior).

Another way of verifying the correctness of a system is by defining some interesting properties a system should have by using an appropriate logic, and checking that the system verifies them. We describe logics used for this purpose in section 2.3. The model checking algorithms which are used to verify whether a property holds in a given process are discussed in section 2.4.

In the remainder of this section we will describe two process calculi, CCS and π-calculus. We will use CCS throughout the rest of this report to argue about the soundness of our approach, but the language on which this thesis' work will ultimately focus on is more closely related to the π-calculus.

2.2.1 CCS

Milner proposed one of the first process calculi which he named Calculus of Communicating Systems (CCS) and consists of the constructs depicted in figure 2.1. CCS captures the essence of concurrent programming by focussing on communication and synchronization by message passing. In figure 2.1 α stands for a *label* which is either a message identifier, represented by a name like `a` or `coin`, or τ which stands for an internal action; L is a set of labels where τ may not be included; and f is a renaming function, defined as $[a_1/b_1, ..., a_n/b_n]$ to state that each a_i is to be replaced by the corresponding b_i. Communication in CCS is synchronous so an action must wait for a complementary action to be available in the environment in order to proceed. The complementary action to α is $\overline{\alpha}$ and the interpretation is usually that α is receiving a message while $\overline{\alpha}$ is sending one. τ has no complementary action. (inaction) denotes a process which can do nothing; (prefix) denotes a process which can make action α and act like process P; (choice) can act either like P or Q; (composition) composes processes P and Q in parallel so that they can act independently or by exchanging messages between themselves; (restriction) disallows process P to communicate using any label in L; and (relabelling) acts as process P where labels are changed according to renaming function f.

The semantics of CCS is defined by using a labelled transition system where we have transition rules with a possibly empty set of preconditions. A transition rule is possible whenever all preconditions are possible, in the likeness of type rules. As an example in figure 2.2 we present the rules for (composition): the first two rules state that either subprocess can evolve independently while the third captures the possibility of the two processes communicating with each other and evolving without an external observable action.

$$
\begin{array}{rclr}
\text{P,Q} & ::= & 0 & \text{(inaction)} \\
 & | & \alpha.\text{P} & \text{(prefix)} \\
 & | & \text{P} + \text{Q} & \text{(choice)} \\
 & | & \text{P} \mid \text{Q} & \text{(composition)} \\
 & | & \text{P} \setminus \text{L} & \text{(restriction)} \\
 & | & \text{P}[f] & \text{(relabelling)}
\end{array}
$$

Figure 2.1 Language Constructs of Milner's CCS

$$
\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \quad \frac{Q \xrightarrow{\alpha} Q' \qquad P \xrightarrow{\overline{\alpha}} P'}{P|Q \xrightarrow{\tau} P'|Q'}
$$

Figure 2.2 Semantics for composition in CCS

We can analyze a process and build a graph depicting all states a process can evolve into and connecting them by edges labeled by the labels which are used to evolve from one state to the other. Using this representation we can analyze a process' possible states and transitions in a very convenient manner. These transition graphs are called Labelled Transition Systems, or LTS, and are used as basis for many analysis on concurrent processes. The LTS is built by checking which transition rules are applicable in the initial state, building an edge for each of them, and repeating the process for the resulting states from each transition.

As an example lets take the process defined as `a.0|b.0`, which we will abbreviate by omitting the empty processes at the end to `a|b`. This process is capable of evolving through two possible transitions by using any of the two first rules in figure 2.2 which are possible because the subprocesses `a.0` and `b.0` are capable of performing a transition using the rule for prefix, by performing an action, `a` or `b` respectively, and evolving to the empty process. Figure 2.3 depicts the LTS built by fully analyzing process `a|b`. The first, top-most, tier shows the originating process with the two transitions we just described; in every state in the second tier one of the subprocesses of the composition is not capable of performing any action, and thus each state has only one edge leading from it to the final state of the system, where none of the subprocesses is capable of performing any action, and as a consequence neither is their composition.

2.2.2 π-calculus

CCS processes have a static nature in the sense that they only communicate through previously known channels, represented as unique message identifiers. The calculus we presented above does not allow the dynamic creation of new channels and the reconfiguration of the process network. The π-calculus, later proposed by Milner, Parrow and Walker in [22], addresses this issue

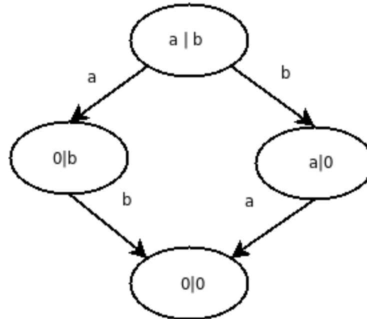

Figure 2.3 LTS for a simple CCS process

π	::=	x(y)	receive y along channel x
		x<y>	send y along channel x
		τ	unobservable action

P,Q	::=	$\sum \pi_i.P_i$	(summation)
	|	P | Q	(composition)
	|	(ν c)P	(restriction)
	|	!P	(replication)

Figure 2.4 The π-calculus

by allowing fresh names to be generated and for those names to be passed along in communications where they can be used as channel names. Figure 2.4 shows the syntax of the resulting calculus.

What this calculus adds in terms of expressiveness is the fact that names can not only be generated by the (restriction) construct, but they can also be passed along and received by other processes. The act of receiving a name through a channel binds the receiving variable to the sent name which can also be used as a channel.

Figure 2.5 shows an example of a system which makes use of the mobility of channels. There are three types of participants in the system: A, which sends and receives a message through a single channel; Talk, which receives and sends a message through a channel plus is capable of receiving another channel which will start being used in the receive-send cycle; and C, which decides whether or not it is time for Talk to switch channels. The mobility in this example is present in how Talk can change the connection it is using by "order" of process C. This example illustrates, in a very simple setting, how process names can be passed along with messages in the π-calculus. These language mechanics are used in [22] to model the handover protocol between mobile phone stations and the mobile devices while they are moving.

Although it may not seem that there is much difference between CCS and the π-calculus,

$$
\begin{array}{lll}
C(x1,x2) & \triangleq & \tau.C(x1,x2) \\
 & & +switch{<}x1{>}.C(x1,x2) \\
 & & +switch{<}x2{>}.C(x1,x2) \\
Talk(c,m) & \triangleq & c(m).c{<}m{>}.Talk(c,m) \\
 & & +switch(c1).Talk(c1,m) \\
A(x,m) & \triangleq & x{<}m{>}.x(m).A(x,m) \\
Sys & \triangleq & (\nu\ x1,x2,m) \\
 & & (Talk(x1,m)|C(x1,x2)|A(x1,m)|A(x2,m))
\end{array}
$$

Figure 2.5 A simple example of mobility

it happens that the added power the possibility of exchanging channel names brings to the language allows it to model the λ-calculus thus making it Turing Complete, which is interesting to note.

2.2.3 Bisimulation

One of the verifying techniques mentioned before is the use of a relatively simple specification process and comparing it to a process modeling the actual implementation we want to verify to check if the behavior is the expected one. Equivalences like structural congruence are not expressive enough to compare processes for this purpose as it is possible for a non-congruent processes to behave in the same manner. One such example are the processes `a|b` and `a.b+b.a` which are not structurally congruent but exhibit the same behavior: can either read `a` and then `b` or vice-versa. In order to compare these processes we need to define a broader sense of equivalence where we take into account the behavior of processes instead of just their syntactic similarities. The behavioral equivalence between CCS processes is elegantly captured by the notion of bisimulation where processes are equivalent if and only if they can mutually simulate each other.

Formally, bisimulation is a relation between labelled transition systems which deems systems equivalent if and only if they are only able to perform equivalent transitions. This means that two processes, or sub-processes, related by a bisimulation can be interchangeably switched without any difference in the overall behavior of the system. Two processes P and Q are in a bisimulation if and only if the following two conditions hold:

1. If $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and Q' is bisimilar to P'.
2. If $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and P' is bisimilar to Q'.

This relation allows to identify, for example, processes `a|b` and `a.b+b.a` as equivalent whereas structural congruence cannot. If we also build the LTS for the latter process it will

$$\begin{array}{lcl} \text{In} & \triangleq & in.\overline{m}.In \\ \text{Out} & \triangleq & m.\overline{out}.Out \\ \text{Sys} & \triangleq & (In|Out)\backslash\ \{m\} \\ \text{Spec} & \triangleq & in.\overline{out}.Spec \end{array}$$

Figure 2.6 Example of weak bisimulation

have a similar structure to the one depicted in figure 2.3. The bisimulation relating the two processes makes a one to one correspondence between the identical nodes of the LTS.

The bisimulation relation defined above forces processes to have the exact same behavior in order for it to be applicable, which may not be entirely desirable as internal behavior must also match. This makes specifications to be used with this kind of equivalence relation be more complicated than needed whereas simpler specifications should be able to describe the intended behavior.

To get simpler specifications we may abstract from the internal details of the systems and compare them using only their observable behavior, from where we exclude τ transitions. For an observer a process which does $\tau.\tau.a.\tau$ behaves in exactly the same manner as a process which only performs a. For that observer there is no discernible difference in the behavior of the two processes (regarding the mentioned transition), making them equivalent to said observer. The notion of bisimilarity mentioned above does not consider two processes with these transitions equivalent but we may want to consider them as if they were.

It is possible to define a different notion of bisimilarity where processes are equivalent if and only if they are capable of making the same *observable* transitions to equivalent states. The bisimilarity characterization for this notion of equivalence can be obtained from the definition of bismilarity given above by exchanging the transitions with a different notion of transition where non-observable actions are not taken into consideration. A transition $P \overset{\alpha}{\Longrightarrow} P'$ is possible whenever a process P can evolve to a process P' by performing a possibly empty series of τ transitions, followed by a transition by α and another possibly empty series of τ transitions. We define *weak* bisimilarity by replacing the $\overset{\alpha}{\longrightarrow}$ transitions with $\overset{\alpha}{\Longrightarrow}$ transitions. With the notion of *weak* bisimilarity in place, we call *strong* bisimilarity to the previously defined equivalence.

Figure 2.6 shows an example of a system, Sys, being weakly bisimilar to a specification, Spec. Sys is composed of two subprocesses which communicate with each other using the restricted label `m`. The idea is that In receives a message, passes it along to Out using a private channel and Out outputs the message. As message `m` is restricted in the system, it can only be used within the system by the communication between In and Out, and gives rise to an unobservable action after `in` is received. Between receiving `in` and sending `out`, Sys performs only the τ transition from the synchronization of `m` between In and Out, which means the Sys has the same observable behavior as Spec, making Sys and Spec weakly bisimilar, and thus equivalent using this notion of equivalence.

2.3 Logics

When reasoning about a concurrent system described in some calculus like CCS or the π-calculus, one cannot derive all interesting properties from just structural properties or behavioral equivalence. In order to reason about higher level properties of systems it is often convenient to express them in specially tailored logics and devise an algorithm for checking if a process satisfies a formula in that logic. These algorithms are called model checking algorithms and we will discuss them in section 2.4.

Special logics have been formulated to express temporal and behavioral properties of systems. Temporal logics, like LTL[25] and CTL[7], allow us to reason about the timing of the satisfaction of properties. Some examples of temporal logic predicates include assessing whether a property will hold eventually in the lifetime of the system, or if it always holds. Behavioral logics, like Hennessy-Milner's and the μ-calculus, allow us to reason about behavioral properties of systems. One example of a behavioral predicate is the possibility for performing an action to reach a state satisfying a given property, which can be used to assess whether a process is in a deadlock state, for instance.

On the remainder of this section we will describe two behavioral logics, Hennessy-Milner's and the μ-calculus. We focus on the behavioral kind of logics because they are best suited for describing the behavior of concurrent systems, which makes them of particular interest for the purposes of this thesis.

2.3.1 Hennesy-Milner Logic

Matthew Hennessy and Robin Milner proposed a logic for reasoning about labelled transition systems in [15]. The logic they proposed consists of the propositions described in figure 2.7 accompanied by the satisfaction rules in figure 2.8. This logic, besides the usual logical operators, adds the modal behavioral operators which allow reasoning about a process' transitions. (diamond) states that a process must evolve to a state satisfying Φ by doing a transition with label α, and (box) states that all transitions by label α must reach a state satisfying Φ. One interesting aspect of these operators is that even though (box) can be defined in terms of (diamond), we cannot define (diamond) in terms of (box): $[\alpha]\Phi$ is equivalent to $\neg\langle\alpha\rangle\neg\Phi$, which intuitively states that it is not possible to do a transition using α which does not lead to a state satisfying Φ; but it is not possible to write a property equivalent to $\langle\alpha\rangle\Phi$ using only the other operators in the logic. Even though the logic could be smaller, we keep the (box) operator because its expressiveness is very useful in defining properties.

The example process we used in section 2.2.1 satisfies the following property which states that it can perform either an a transition or a b transition: $\langle a\rangle\top\wedge\langle b\rangle\top$. Another property we can define is $[a](\langle b\rangle\top\wedge [a]\bot)$ which states that if it is possible to perform a transition with label a, after doing it it is possible to do a transition by reading b but not by reading a.

A quite intuitive property, which was proven by Milner in [16], of this logic is that bisimilar processes satisfy exactly the same formulae. In other words, we cannot devise an HML formula

$$
\begin{array}{lccr}
\Phi, \Psi & ::= & \top & \text{(true)} \\
 & | & \bot & \text{(false)} \\
 & | & \Phi \vee \Psi & \text{(or)} \\
 & | & \Phi \wedge \Psi & \text{(and)} \\
 & | & \neg\Phi & \text{(negation)} \\
 & | & <\alpha>\Phi & \text{(diamond)} \\
 & | & [\alpha]\Phi & \text{(box)}
\end{array}
$$

Figure 2.7 Hennessy-Milner logic

$$
\begin{array}{lll}
\text{E} \models \top & & \\
\text{E} \not\models \bot & & \\
\text{E} \models \Phi \wedge \Psi & \text{iif} & \text{E} \models \Phi \text{ and } \text{E} \models \Psi \\
\text{E} \models \Phi \vee \Psi & \text{iif} & \text{E} \models \Phi \text{ or } \text{E} \models \Psi \\
\text{E} \models \neg\Phi & \text{iif} & \text{E} \not\models \Phi \\
\text{E} \models <\alpha>\Phi & \text{iif} & \exists F \in \{E' : E \xrightarrow{\alpha} E'\} \quad F \models \Phi \\
\text{E} \models [\alpha]\Phi & \text{iif} & \forall F \in \{E' : E \xrightarrow{\alpha} E'\} \quad F \models \Phi
\end{array}
$$

Figure 2.8 Hennessy-Milner Logic's semantics

capable of distinguishing two strongly bisimilar CCS processes.

2.3.2 μ-calculus

The Hennessy-Milner Logic (HML) which we described in the previous section is only capable of reasoning about finitely many transitions, as all have to be explicit. This restriction makes it unsuitable to reason about eventual (something will happen in some future state) or permanent (something always happens) properties. To address this lack of expressiveness an extension of HML, called the μ-calculus, was devised by de Bakker in [1].

The μ-calculus extends HML by adding fixed point operators which allow recursive property definitions. The added operators are $\mu.X(\Phi)$ and $\nu.X(\Phi)$ for minimal and maximal fixed point on formula X, respectively.

Using these new operators one can define properties like "it is always possible to perform a transition by reading a", $\nu.X(<a>\top\wedge[-]X)$, or "sometime in the future, this process will be able to read a", $\mu.X(<a>\top\vee<->X)$. In both examples we use - to denote all possible transitions.

2.4 Model Checking

Model checking, introduced by Clarke, Emerson and Allen in [6], consists of checking whether a model, formulated in some algebraic language, satisfies a given logical formula and originally appeared as a technique for verifying if digital circuits corresponded to their specification. Model checking has the restriction of only being applicable to finite state systems which confers it the property of being able to be performed automatically. A bonus of being applicable only to finite systems is that we gain the certainty that, given enough resources, the recursive model checking algorithm will eventually terminate. However, when used in the context of verifying large concurrent systems model checking suffers from the problem of state explosion where there may be simply so many states to check, due to their exponential growth from non-determinism, that in practice the algorithm will take too much time to execute or not have enough resources to deal with the problem at hand.

In order to cope with the state explosion problem and be able to verify larger systems there are several techniques available to both the implementers of model checking algorithms and to the people specifying systems to be model checked. Some of these techniques are used to reduce the resource usage of the algorithm, by using a more succinct representation for the system states (ordered binary decision diagrams as in [3], for example), or the running time of the algorithm, by checking only a possible subset of interleaving actions built from partial orderings[14] of independent actions. Apart from the algorithmic improvements which can be done, system specifiers can also abstract some parts of the system to smaller subsystems with identical properties in order to minimize the number of states in the whole system. There are many other aspects of the system, like symmetry for one, which can be exploited to allow this technique to be employed on even larger systems.

Using model checking on a system has three stages: Modeling the system by converting it to a formalism accepted by the model checking tool; Specifying, using an appropriate logic, which properties are desired that the system verifies; and the actual Verification which is, ideally, done automatically by a tool.

2.5 Techniques for bisimulation checking

In order to compare two processes to check if there is a bisimulation relating them, i.e. they are bisimilar, it is possible to create the full transition graph and attempt to build the bisimulation relation by using its definition directly. This however is very time consuming and impractical for most but the smallest processes. To be able to cope with the number of states a reasonably larger system has, it is necessary to use better techniques for bisimulation checking.

There are a couple ways which have proven to be successful in dealing with creating a bisimulation relation: partition refinement algorithms and "on the fly" verification techniques are the most prominent.

As suggested by Kanellakis and Smolka in [19], looking for a bisimulation relating two

processes can be seen as a partition refinement problem. A simplistic view of this approach is that we can start with all the states in a single partition, considering them all equivalent, and then iteratively refine the partitions based on which transitions are possible in a state until a fixed point is met. Once no more refinements are possible, the bisimilarity test is simply a matter of checking whether the starting points of the processes being compared ended up in the same partition, i.e. the same class relating to the notion of equivalence used in the partitioning. There are several optimizations which can be employed in order to reduce the running time of the algorithm, being that currently the most efficient are the ones due to Paige and Tarjan in [23].

The above mentioned technique deals with the problem of exhaustively searching the state space to build a bisimulation relation in an efficient manner. Another way to allow verification of larger systems is by reducing the space complexity of the algorithm, which Fernandez and Mounier showed to be possible in [11]. The algorithm they propose does a Depth-First Search of the space state instead of expanding it fully, making it require a lesser amount of memory resources.

This technique was employed in the Aldébaran tool in order to improve the efficiency of its bisimilarity checker, but the former technique is by far the most widely spread being used in the majority of tools which support equivalence checking of some sort.

2.6 Tools which support bisimulation checking

Bisimilarity checking is implemented in a reasonably large number of tools. Most of these tools support the specification of processes based on some process calculus and incorporate bisimilarity checking using one of the above discussed algorithms. In this section we discuss some well known tools.

Victor and Moller's Mobility Workbench[29] (MWB) is much like the Concurrency Workbench (CWB) of Cleaveland, Parrow and Steffen[9] in terms of functionality, but they differ in the specification language used. They work on different base languages, CWB uses CCS while MWB is based on the π-calculus, but both provide a fairly wide range of equivalence relations to compare processes with. CWB provides strong and weak bisimilarity while MWB can be used to compare processes with strong, weak, and open bisimilarity. They also differ in what algorithm is used when checking for equivalence between processes. CWB uses partition refinement but due to the dynamic nature of the π-calculus, the authors of MWB cannot use partition refinement algorithms for bisimilarity checking and use the "on-the-fly" techniques to implement the equivalence test.

The SIGREF tool[31] employs Binary Decision Diagrams (BDDs) to shorten the representation of the transition graphs used in the partition refinement algorithm used to check for branching bisimilarity between CCS-like processes.

Construction and Analysis of Distributed Processes[13] (CADP) is a recent tool which stems from the older Aldébaran[8]. In this tool the adopted language is called LOTUS which is an ISO

standard based on CCS and CSP. This tool allows for strong and weak bisimilarity comparison between processes and implements almost every mentioned technique. The tool uses a partition refinement algorithm with BDDs but can also use "on-the-fly" techniques when appropriate.

2.7 Spatial Logic Model Checker

The Spatial Logic Model Checker (SLMC) developed by Hugo Vieira, Luís Caires and Ruben Viegas[30] is a model checking tool written in OCAML which incorporates a process definition language based upon the π-calculus, and a logical specification language which includes behavioral and spatial properties[4].

The behavioral subset of the logical language is based on the μ-calculus, while the spatial predicates allow for reasoning about the structure of the system. Some interesting properties which can be seen as spatial include connectivity, resource availability and even some security properties, like secrecy, can be considered spatial properties of the system.

3. Verifying Bisimilarity using Model Checking

In this chapter we introduce the possibility to use a logical formula to assert whether or not a process is bisimilar to another and provide some algorithms to build such formulae based on CCS processes. We provide some examples of application of this technique and discuss its limitations and how we can overcome them.

3.1 Building Characteristic Formulae for CCS Processes

The equivalence relation induced by a given logic over two programs is usually given in the terms of whether or not it is possible to devise a formula in the logic which can differentiate the two processes. In simpler terms, two processes are equivalent in regard to a certain logic if for any formula one of the processes satisfies, the other also satisfies the same formula.

Using this result directly would involve going though all possible formulae and checking if the two processes provided the same result for each formula (although search space can be reduced, it would always be rather big). Another, more suitable, approach to using this result to verify process equivalence is by attempting to build a distinguishing formula. Such a formula would be satisfied by one process and not by the other therefore the logic would be capable of revealing that the processes are not equivalent. How to build a distinguishing formula is well known, and is used in several tools, like the Concurrency Workbench, to provide better feedback on the user in understanding why a process is not equivalent to another.

Building a formula capable of distinguishing two processes is interesting but requires that both processes to be compared be known. It would be even more interesting to be able to build a formula, named a characteristic formula, from one of the processes which could be used to assert whether or not any other process is equivalent to the one which originated it. This formula would attest to the complete behavior of the process up to bisimulation given certain conditions on the compared process were satisfied.

The first step towards building a characteristic formula from a CCS process was given by Graf and Sifakis in [26] where they describe a function between a process and a formula which provides characterization up to bisimilarity. The logic used is very similar to Henessy-Milner's while the process calculus is basically CCS without the parallel composition operator or recursion. The expressive power of the model is basically the same as CCS since it is always possible to rewrite a process with parallelism into a process with only choice by enumerating all possible interleavings of the parallel compositions' transitions. Another way to interpret the process calculus they used in this paper is by treating it as a calculus which describes an LTS, since it describes the different actions a process can do at any given point.

In [27] Steffen extends Graf and Sifakis' approach to also be able to deal with recursively defined processes, as long as the process is finite state. Steffen's way to build characteristic formulae is based on the LTS of a process and uses a fixed point operator from the modal μ-calculus to deal with recursion.

In the next two sections we will describe how to build characteristic formulae based on a Labelled Transition System given by a quadruple (S, Act, T, s) where:

- S is a (finite) set of states.
- Act is a (finite) set of actions.
- T is a function $S \times Act \times S$ relating how a state can evolve to another state by performing an action.
- s is the initial state.

On each section we will also show how a characteristic formula looks like for a few simplistic examples.

3.1.1 Finite Processes

We can describe the generation of the characteristic formula for an LTS through a function Tr applied to the initial state of the LTS which is defined for any state in the LTS as follows:

$$\bigwedge_{a \in B} < a > Tr(s') \quad \wedge \bigwedge_{a \in Act} [a] \bigvee_{s' \in C(a)} Tr(s') \tag{3.1}$$

Where

- B is $\{a : a \in Act \wedge \exists s' : (s, a, s') \in T\}$
- $C(x)$ is $\{s : s \in S \wedge \exists (s, x, s') \in T\}$

Following the usual convention that $\bigwedge \emptyset$ is $true$ and $\bigvee \emptyset$ is $false$, this method of building a formula equates to having for each state a formula which is a logical conjunction of:

1. $<\alpha>s$ for each transition α leading to the state satisfying s.
2. $[\alpha](s1 \vee s2 \vee ... \vee sn)$ for all states ($s1$ to sn) lead to by label α, for every α with a transition in that state.
3. $[\alpha]\bot$ for all transitions α not possible in that state.

The conditions introduced for each state in the characteristic formula force a conforming process to be able to perform transitions to states which satisfy the same formula as the state the original process transitioned to, from the conditions introduced by 1. The conditions in 3 force that the process does not engage in a transition which was not present in the originating process. The conditions introduced by 2 force that the process does not perform any transition to a state not mentioned in 1 using the actions not forbidden by 3. These conditions basically equate to the process satisfying this formula having to perform the same transitions (meaning, going through the same action to an equivalent state), perform only those transitions and not get any added transition, which is how we intuitively see bisimilarity.

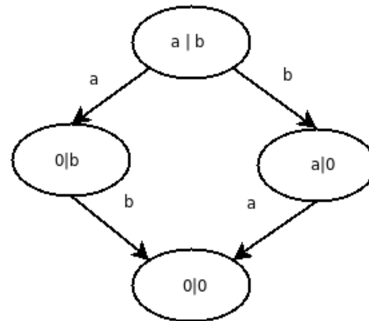

Figure 3.1 The LTS for process a|b.

```
s3 = [a]false and [b]false

s1 = <a>s3 and [a]s3 and [b]false
s2 = <b>s3 and [b]s3 and [a]false

s0 = <a>s2 and <b>s1 and [a]s2 and [b]s1
```

Figure 3.2 Formulae for characterizing a|b.

3.1.1.1 Examples

To illustrate the construction of a characteristic formula for a finite process we will see how it is built for process `a | b`, and then reason that process `a.b + b.a`, which is well known to be bisimilar to the first process, satisfies the introduced formula. We will also exemplify how can non-bisimilar processes be identified as such by the built formula.

The process `a | b` has the LTS depicted in figure 3.1, where from the initial state it is possible to perform a or b, on the final state no actions are possible and there is a state for each transition in the initial state where it is possible to do a transition to the final state using the action which was not the one leading to that state.

Figure 3.2 shows the characteristic formulae for the above mentioned process. Property s3 describes the behavior of the bottom state in the LTS of figure 3.1, where no action is possible. The formulae s1 and s2 describe the intermediate states where b or a were performed, respectively, from the initial state. In these states it is possible to perform a, respectively b, leading to a state satisfying s3, and b, respectively a, cannot be performed in states satisfying these formulae. The initial state for the process yields formula s0 from figure 3.2. This formula states what transitions can be done, namely one using label a to a state satisfying s2 and one using label b to a state satisfying s1. The formula also asserts that all transitions by label a must lead to a state satisfying s2 and similarly to b and s1.

For the remainder of this section we will analyze how processes `a.b + b.a`, `a.b` and

a.b + b.a + a.a satisfy or not the characteristic formula we came up with for a|b.

Process a.b + b.a is known to be bisimilar to a|b, so it should satisfy the formula. On its initial state it can perform a or b. If it performs a it will arrive in a state where it can do b and become inactive, which satisfies formula s2. If it performs a, it transforms into a process which satisfies s1 as it can, and can only, perform b after which it becomes inactive.

Process a.b should not satisfy the characteristic formula for a|b, and it indeed fails to satisfy s0 from the fact that it cannot perform b on its initial state.

Process a.b + b.a + a.a should not satisfy s0. It perfectly satisfies <a>s2, <b>s1 and [b]s1, but fails to satisfy [a]s2 as there is a transition by a to a state which does not satisfy s2, but instead satisfies s1.

The three examples in the previous discussion try to illustrate how the characteristic formulae devised by Graf and Sifakis detect deviant behavior from the process which originated the formula. We exemplified what happens when the process does behave correctly, how the formula detects some transition is missing and how it detects that some transition is there when it should not be.

This method for building characteristic formulae is the basis for the method we explain in the next section which extends it in order to add support to infinite behavior as long as the process has a finite number of states.

3.1.2 Finite State Processes

Building a characteristic formula for finite state processes which may have recursive, and therefore infinite, behavior requires that the logic in which we express the formula also allows the definition of recursive formulae. For this purpose we use the modal μ-calculus to make use of its fixed point operators to construct the desired formula.

Dealing with infinite behavior using the approach described in the previous chapter would result in formulas with infinite size due to the fact that recurrent states would always be revisited. In order to cope with this issue we need to be able to represent infinite behavior in a finite fashion and that is where the novel operators in the μ-calculus come into play, by allowing some form of recursion in logical formulae.

In [27] Steffen was the first to provide an algorithm which allows translating a CCS-like process into a characteristic formula in the μ-calculus. Steffen's approach is based on the "rewriting" of the LTS to turn it into a regular tree so that it is possible to do the depth-first application of an algorithm which introduces fixed-point calls when they are needed. His algorithm has two steps, one initialization step and one depth-first association between each node and a formula for that node:

1. **Initialization**: Assign a fresh variable to every node.
2. **Formula**: In depth-first order do, for every node:
    - build a formula as in the previous section, however, use the node's currently assigned formula as the continuation of each action.

```
SemSpec = p.Sem1                        Sem     = p.v.Sem
   Sem1 = p.Sem0 + v.SemSpec            SemGood = Sem | Sem
   Sem0 = v.Sem1
                                        SemBad  = p.v.SemBad +
                                                  p.p.v.v.SemBad
```

Figure 3.3 Semaphores

- replace the currently assigned formula with the formula built in the previous step with a fixed point "around" it using the node's current formula as the fixed-point variable.

Instead of analysing the LTS and building the equivalent regular tree, it is possible to achieve the same result by depth-first searching the LTS and maintaining a list of nodes in the path to the current node. Whenever a transition leads to a node in the path, it is a back-arc and a variable should be used instead of further building the formula.

3.1.2.1 Examples

We will now provide two examples of how to use this method to build characteristic formulae. In the first example we will try to distinguish a good and a bad implementation of a two unit semaphore by providing a specification of the behavior of the semaphore, a good implementation by composing two one unit semaphores in parallel and a bad choice based implementation. On our second example we will show why the transformation of the LTS into a regular tree is necessary in order for the algorithm provided by Steffen to work.

On our first example we use the processes defined in figure 3.3 where we can see the definition of four processes: a specification for a two-unit semaphore (`SemSpec`), the implementation of a one unit semaphore (`Sem`), the implementation of a two-unit semaphore by parallel composition of two one-unit semaphores (`SemGood`), and the failed attempt at the definition of a two-unit semaphore using only choice (`SemBad`). We will build the characteristic formula for `SemSpec` and check whether `GoodSem` and `BadSem` satisfy it so as to identify which of them correctly implements a two-unit semaphore.

A semaphore is a computing unit from which one can obtain a resource and, having it, can release it. The operations which are possible over a semaphore are `p` to acquire the resource and `v` to relinquish it. When the resource is held by someone, no one else can obtain it until it is released. A two-unit semaphore holds two resources, which means two `p` operations can be performed before the semaphore is unable to give out the resource. As we can see by the specification, after performing one `p` it is possible to perform another `p`, but it is also possible to immediately perform `v` returning the semaphore to its initial state, and after performing `p` twice without a `v` in between the only possible action becomes `v`, which leads back to the state where both `p` and `v` can be performed.

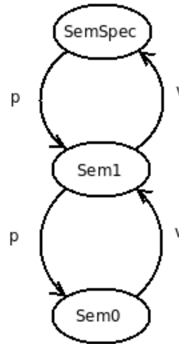

Figure 3.4 LTS for SemSpec.

```
semspec = maxfix SemSpec.(
              <p>sem1 and [p]sem1 and
              [v]false)
   sem1 = maxfix Sem1.(
              <p>sem0 and <v>SemSpec and
              [p]sem0 and [v]SemSpec)
   sem0 = maxfix Sem0.(
              <v>Sem1 and
              [v]Sem1 and [p]false)
```

Figure 3.5 Characteristic formula for SemSpec.

Figure 3.4 depicts the LTS for SemSpec, based on which we will build the characteristic formula for that process. The LTS is already a regular tree, which means no transformation is necessary in order to correctly apply Steffen's algorithm. Using Steffen's algorithm we come up with the formula depicted in figure 3.5, where uncapitalized variables are to be taken as full text substitutions by the appropriate formula while capitalized variables are fixed point "calls" and are to be taken as is.

Process GoodSem, which is bisimilar to SemSpec, satisfies this property while process BadSem, which is not bisimilar to SemSpec, does not. GoodSem can evolve in two ways by p and cannot perform v, so in order to satisfy semspec, both transitions must lead to processes which satisfy sem1. By p, GoodSem can become either v.Sem|Sem or Sem|v.Sem, which are structurally congruent and will hence satisfy the same formulae, for that reason we only analyze what happens with v.Sem|Sem with regard to it satisfying sem1. This process can perform both p and v as required by the formula, and by performing v it will become GoodSem, which by the fixed point hypothesis satisfies SemSpec. This hypothesis is only valid if the process which we get from performing p satisfies sem0. The process v.Sem|Sem evolves by p to v.Sem|v.Sem, and if that process satisfies sem0, GoodSem satisfies semspec. v.Sem|v.Sem cannot perform

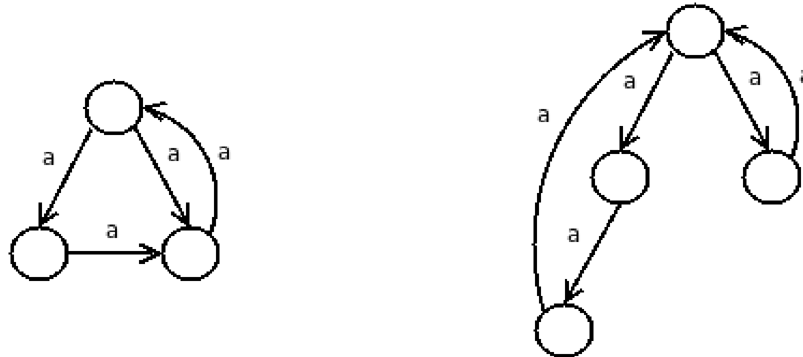

Figure 3.6 LTS for process S.

p, and is able to perform v by reacting with either of the processes in the parallel composition, where it becomes v.Sem|Sem or Sem|v.Sem, both of which are structurally congruent and, by the fixed point hypothesis, satisfy Sem1. From this we can conclude that GoodSem satisfies semspec and is therefore correctly identified as bisimilar to SemSpec by the characteristic formula.

On the other hand, BadSem might also seem to have a behavior compliant with SemSpec, as it can either perform p and then v, or p twice and then v twice before returning to the starting state. However this process has the problem of making a premature "decision" about what case it is expecting. After performing the first reaction by action p this process transforms into either v.BadSem or p.v.v.BadSem which means in one case it can only perform v, and in the other it can only perform p, negating the possibility for the environment to choose which action to perform. This misbehavior is caught on by the characteristic formula by forcing all states after performing a p from the original one to satisfy sem1. Neither of the possible continuations for BadSem is capable of satisfying sem1 because it will either not allow p or v.

We will use the recursive process S = a.a.S + a.a.a.S to illustrate the need to turn the LTS into an equivalent regular tree so that Steffen's algorithm can be used. This process' LTS has three states as depicted in the left side of figure 3.6. If we were to apply Steffen's algorithm before transforming the LTS into the regular tree depicted in the right side of figure 3.6, we would end up with a reference to a fixed point variable which would not be in the path to that expression (from the transition in the bottom of the LTS). The formula we would end up with using this approach is the one on the left side of figure 3.7 where we also highlight, on the right side of the same figure, the problematic substitution. By adopting the equivalent regular tree LTS this problem is no longer posed, as the first back reference is to the root of the tree, as it should. The algorithm for building the formula which uses a set of nodes in the path to the node can be applied directly to the original LTS as it will only introduce the reference to the fixed point variable when the transition leads to the root of the LTS, yielding the correct desired formula.

```
s0 = maxfix S0.(
       <a>s1 and <a>s2 and
       [a](s1 or s2))              maxfix S0.(
s1 = maxfix S1.(                     <a>maxfix S1.(<a>S2 ... )
       <a>S2 and [a]S2)              ... )
s2 = maxfix S2.(
       <a>S0 and [a]S0)
```

Figure 3.7 Erroneous formula for S.

3.2 Challenges

The main issue with the approach described in this chapter for the generation of characteristic formulae is the amount of space which the formula itself requires. We shall illustrate this matter using the semaphore example in figure 3.5.

Formula `sem0` appears twice in formula `sem1` which in turn also has two copies in formula `semspec`, which means formula `sem0` ends up repeated four times in the final formula used to characterize the behavior of the `SemSpec` process. More complicated processes will suffer the same problem and exhibit an even greater blowup in size if there is the possibility of reaching a state by several possible paths in the LTS, as that whole state's characteristic formula needs to be repeated in all the possible paths in the formula.

Even if we can somehow eliminate the space requirement for the formula, the underlying reason for the issue still exists and will manifest itself when we attempt to analyze the new process. The deeper problem is that we need to traverse all possible paths and assert that all of them are in conformity with the behavior of the system which originated the formula, even if we get to that a similar state from a different path. There is a lot of redundancy in these formulae which needs to be dealt with in order for this approach to be viable.

In our implementation we overcome the space challenge by using a characteristic equation system instead of a single formula. The use of this equation system means that the formula for each state is only defined once and an abbreviation of it is used in all the places where it would appear. This eliminates redundancy in the formula, reducing its size.

The above solution covers the need to reduce the formula size but does nothing to aid the complexity of the actual verification of the new formula against a process. Without further measures taken, checking this formula will still have many redundancy. We combat this redundancy by remembering which processes satisfy each formulae and reuse this information when it is needed without having to recalculate.

4. Spatial Logic Model Checker

The work performed for this thesis was based on SLMC, an already existing tool which supports Spatial Model Checking on a π-calculus like language using a logic which extends the μ-calculus with spatial properties. In this chapter we describe the tool as it was and how it has become as a result of this work. We also describe the inner workings of the new functionalities and provide an in-depth discussion of the choices made.

4.1 The Spatial Logic Model Checker

The Spatial Logic Model Checker is a tool which allows the automatic verification of behavioral and spatial properties of processes described in a process language similar to the π-calculus. Besides the operators from the μ-calculus SLMC introduces operators capable of reasoning about the structure of processes. In the following subsections we will present the language for processes and the language used to express properties of those processes. We will also provide some insight on the meaning of the formulae.

4.1.1 Syntax of Processes

Processes in SLMC are defined using the language shown on figure 4.1. This language is basically the π-calculus where sending a message with arity two in channel `a` is written `a!(m,n)`, and receiving one is written `a?(x,y)`. We also have the choice operator which is here written using the `select` construct instead of a + operator. As the language accepted by SLMC is based on the polyadic π-calculus, when sending or receiving messages in channels, they can exchange any number of names. In this language it is also possible to represent the internal silent (unobservable) action, and compare names by their equality (or inequality). As we will see, it is possible to define processes based on others, so there is a construct which allows using another process' definition to aid in the construction of a given process. It is also possible, using the `new` construct, to declare fresh names private to the enclosed process.

4.1.2 Syntax of Formulae

Formulae in the SLMC have the typical logical connectives and constants, to which they add the Hennessy-Milner modalities for possibility and necessity of an action. We can also define recursiveness in formulae by using the maximum and minimum fixed point operators which are also able to be parametrized. In addition to this there are also a few quantifiers over names: the existence quantifier, `exists`; the universal quantifier `forall`; and quantifiers for fresh and hidden names, respectively `fresh` and `hidden`. The temporal modality `always` expresses that a formula is satisfied in all configurations of a system with regard to its internal evolution. The `eventually` modality states that a property will be true after some unspecified number of

```
name      ::=  lower (letter | digit | _)*
CapsId    ::=  upper (letter | digit | _)*
namelist  ::=  ϵ | name (,name)*
prefix    ::=  name!(namelist)
            |  name?(namelist)
            |  [ name  = name ]
            |  [ name != name ]
            |  tau
process   ::=  0
            |  process | process
            |  new namelist in process
            |  prefix.process
            |  select { prefix.process (; prefix.process)* }
            |  CapsId(namelist)
            |  (process)
```

Figure 4.1 Syntax of SLMC processes.

internal steps.

It is important to be aware of what the various possible labels for the behavioral modalities of the logic mean:

- `tau` or an empty label, denote an internal computation step.
- *name* denotes any action, either input or output, with subject *name*.
- `?` denotes any input action.
- `!` denotes any output action.
- *name*`?` denotes any input action with subject *name*.
- *name*`!` denotes any output action with subject *name*.
- *name*`?(`*namelist*`)` denotes a particular input action.
- *name*`!(`*namelist*`)` denotes a particular output action.

The spatial operators of the logic include the `void` primitive which is true if the process is the inactive process; the composition connective (`|`) which is satisfied when we can separate a process into two parallel processes where one satisfies the property on the right hand side of the operator, and the other satisfies the property on the left hand side; the decomposition connective (`||`) is the de Morgan dual of `|`, meaning `P |= A || B` if and only if `P |= not (not A | not B)`; the numeral property which is true when a process has exactly *number* parallel components; the free operator, `@`, which tests whether a process has a determined free name; the revelation operator, `reveal`, which is true if a process has a restricted name and satisfies the following property; and its dual, `revealall`.

Finally, when defining a property there is also the possibility of using another, already defined property, by using the property's name, which can be parametrized in both names and other formulae.

The logic presented here induces a form of equivalence which is stronger than bisimilarity as it is capable of distinguishing processes based on their structure, instead of just on their behavior like what happens with the μ-calculus. An example of this property is that the processes `a.b+b.a` and `a|b`, can be distinguished by the formula `not void | not void` while they cannot be distinguished by any formula in the μ-calculus. The complete characterization of the equivalence induced by the logic the SLMC tool uses is further described in [5]. As the main interest in this work is in characterizing bisimilarity we will not be making use of the spatial operators present in the logic.

4.1.3 Top Level Commands

The SLMC tool is used by issuing one or more of several top level commands. The basic functionality provided by these commands includes the definition of both processes and properties and the possibility to check if a process satisfies a property.

4.1.3.1 Defining Processes

To define a process in SLMC we use the `defproc` command. The `defproc` command is issued in the following manner:

```
defproc ID[(n1,...)] = <process>
(and ID[(n1,...)] = <process>)*;
```

We can create a process by specifying an identifier for it, a possibly empty list of parameters and its definition using the process language described previously. Processes can be recursive and, more interestingly, mutually recursive as we can define several processes at once, using the `and` connective. The process identifiers need to start with a capital letter. We now define the simple process `a|b` in SLMC:

```
defproc Simple = a?() | b?() ;
```

To illustrate the more advanced use of SLMC's process defining command, we will now define the two-unit semaphore process we used in the previous chapter where the channels to be used for communicating with the semaphore can be parametrized.

```
formula ::= true
          | false
          | formula and formula
          | formula or formula
          | formula => formula
          | formula <=> formula
          | not formula
          | ( formula )
          | <label> formula
          | [label] formula
          | minfix CapsId.formula
          | (minfix CapsId(namelist).formula) (namelist)
          | maxfix CapsId.formula
          | (maxfix CapsId(namelist).formula) (namelist)
          | CapsId
          | CapsId(namelist)
          | name == name
          | name != name
          | @ name
          | exists name . formula
          | forall name . formula
          | hidden name . formula
          | fresh name . formula
          | always formula
          | eventually formula
          | reveal name . formula
          | revealall name . formula
          | inside formula
          | void
          | formula | formula
          | formula || formula
          | number
          | Id(namelist,formulalist)

label   ::= tau
          | name
          | ?
          | !
          | name?
          | name!
          | name?(namelist)
          | name!(namelist)
          | *
```

Figure 4.2 Syntax of Formulae in SLMC.

```
defproc SemSpec(p,v) =
    p?().Sem1(p,v)

and        Sem1(p,v) =
    select {
       p?().Sem0(p,v) ;
       v?().SemSpec(p,v)
    }

and        Sem0(p,v) =
    v?().Sem1(p,v) ;
```

4.1.3.2 Defining Properties

In SLMC it is possible to define a named property through the use of the `defprop` command. This command is very similar to the `defproc` command except that property identifiers need to start with a lowercase letter and we cannot define mutually recursive properties. In order to define recursive properties we have to rely on the logic's fixed point operators instead of being able to use a formula's own identifier to recursively refer to it. The concrete syntax for defining a formula is given bellow, where a property can be parametrized in a set of names and/or in a set of other properties.

```
defprop id[(n1,...,P1,...)] = <formula>;
```

An example of the definition of a property in SLMC could be property `semstart` which states that it is possible to perform `get` but not `put`. This property can be defined in SLMC as

```
defprop semstart =
   <get>true and [put]false ;
```

However, as our two unit semaphore is parametrized on the names of the channels to be used for communication, we might want to also parametrize our property on the names to be used. In addition we might want that the behavior after the first get also be given as a parameter in order to be able to use the same definition to check whether the first state is correct or to be able to check more deeply the process' behavior. One possible definition for the new `semstart` property could be the following

```
defprop semstart(p,v,C) =
   <p>C and [v]false ;
```

It is important to note that the name of the property which is given as a parameter starts with an uppercase letter and the names of parametrized channels start with lowercase ones.

4.1.3.3 Checking Properties of Processes

Verifying if a process satisfies a certain property is done via the `check` command, which is used in the following manner:

```
check <process id> [(n1,...)] |= <formula>;
```

In this command one check whether an already defined process satisfies a formula. Unlike the process which must have already been defined via the `defproc` command, the formula can be defined *in-loco*. We can check that the `Simple` process is capable of performing both an `a` and a `b` transitions by issuing the following command:

```
check Simple |= <a>true and <b>true;
```

We can also check properties which were already defined, for instance that `SemSpec` satisfies `semstart` (with the proper parameters).

```
check SemSpec(get,put) |= semstart(get,put,true);
```

As was mentioned earlier, we can also use the `semstart` property to use more complex properties in its third parameter. For example, we can check that the process following a `get` can perform another `get`, and also a `put`.

```
check SemSpec(get,put) |=
      semstart(get,put,<get>true and <put>true);
```

4.1.3.4 Additional Commands

To facilitate the development and verification of more complex systems or examples, it is possible in SLMC to issue a command which reads a file containing other commands. The `load` command takes a file name enclosed in double quotes and reads commands from it. The filename can be relative to the current working directory or absolute in the file system, as expected. To allow the user to inspect and change the current working directory, SLMC provides the top level commands `pd` and `cd` to respectively perform those actions.

In SLMC there is also a number of variables which control the behavior of the tool. These parameters can be listed, seen and changed their values using the `parameter` command. This command can be called without specifying a parameter, specifying one and specifying a parameter and a new value for it. If no parameter is specified the command shows a list of possible parameters, if a parameter is specified its value is shown, and if a parameter and a new value are specified the parameter's value is changed to the new value. Currently there are three parameters available in SLMC: `show_time`, `check_counter` and `max_threads`. The boolean parameter `show_time` determines whether or not a calculation of time required to verify a property should be provided when the `check` command is used; the boolean parameter `check_counter` toggles

showing a count of the number of times some property was checked against a process during the checking of a bigger property; the integer parameter max_threads controls how many transitions there must be in a process for it to be considered unmanageable by the system. The max_threads parameter is used to provide a cut-off point when processes seem to belong to the class of processes for which the model checking algorithm does not terminate.

4.2 Extensions to the Tool

We have implemented a few extensions for the SLMC tool to permit its usage as a tool for checking bisimilarity between processes. The implemented extensions include a way to directly check the bisimilarity of two processes and an operator in the logic which allows the construction of a characteristic formula. As some sort of side effects from the development of these functionalities there was also the introduction of a new modality into the logic and the implementation of a feature which allows observing a process' behavior.

In the remainder of this section we will describe the interface to the introduced functionalities, while on the next section we discuss their implementation details.

4.2.1 Process Stepping

We have implemented a feature which allows a user to interact with a process in a step-by-step manner. This feature allows the user to select one of the possible actions a process can perform and evolves the process according to the user's selection. It is capable of detecting when the process cannot perform any action and when a process is at a repeated state.

We can access this functionality by issuing the step command as follows:

```
step P[(n1,...)];
```

Figure 4.3 shows an example usage of the step command, where we interact for a while with process SemSpec(get,put). As can be seen, the tool starts the interaction by enumerating the free names found in the process. For each state we will get information on what restricted names exist in that state and we also get a list of possible actions to choose from. At this point we can either interact with the process by activating one of the actions or quit the stepper. The interaction depicted in figure 4.3 will perform two get actions and then two put actions, which will return the process to its initial state. After the interaction is complete, one can exit the stepping mode by typing "q". Note that after the first put a warning is issued alerting the user that this state has already been visited. Although not pictured in this example, because of its infinite behavior, if the stepping reaches a state where no actions are possible, it will alert the user and exit the stepping mode immediately.

```
> step SemSpec(get,put) ;
Stepping SemSpec(get,put)
free names: put, get

restricted names:
Please select a transition to follow, type 'q' to quit:
1: <get?()>
goto> 1

restricted names:
Please select a transition to follow, type 'q' to quit:
1: <put?()>
2: <get?()>
goto> 2

restricted names:
Please select a transition to follow, type 'q' to quit:
1: <put?()>
goto> 1

YOU HAVE ALREADY BEEN TO THIS STATE
restricted names:
Please select a transition to follow, type 'q' to quit:
1: <put?()>
2: <get?()>
goto> 1

YOU HAVE ALREADY BEEN TO THIS STATE
restricted names:
Please select a transition to follow, type 'q' to quit:
1: <get?()>
goto> q
>
```

Figure 4.3 Example usage of the `step` command.

4.2.2 Verifying Bisimilarity

We have also extended the functionality of the `check` command to allow it to be used to test whether two processes are bisimilar. The check command is extended in the following manner:

```
check Id1[(n1,...)] sim Id2[(n2,...)] ;
```

Just like the version for the `check` command for checking formula satisfaction, the bisimilarity version of the `check` command only accepts the identifiers and possible parameters for already defined processes, not allowing the *in-loco* definition of processes. This command will respond with either "`The processes are bisimilar`" or "`The processes are not bisimilar`" in the corresponding case.

4.2.3 Extensions to the Logic

We have realized two extensions to the logic by introducing a new behavioral modality and a new operator. The new modality is a side effect from the optimizations made to the characteristic formula generation procedure which we will explain in greater detail in the next section. The new operator relates directly to using the tool in the context of verifying bisimilarity using characteristic formulae since it allows the creation of the formula which characterizes a given process.

4.2.3.1 The `proc` operator

The `proc` operator we introduced into the logic receives a process as a parameter and assumes the semantics of a characteristic formula for that process, meaning it is only to be satisfied by a process bisimilar to the one which originated the formula. The most obvious use of such an operator is for directly defining a property which is the characteristic formula for a given process, for example:

```
defprop semspec = proc(SemSpec(get,put)) ;
```

This property can then be used to verify that the process `SemSem` which consists of two parallel one-unit semaphores is indeed bisimilar to `SemSpec`, given the right names are used when parametrizing it. This can be done by using the `check` command:

```
check SemSem |= semspec ;
```

The same property can also be used to verify bisimilarity whether several other processes are bisimilar or not to the originating process without having to generate the characteristic formula again. The fact that the formula does not get generated again is one of the main advantages of using this operator. Another advantage of this operator is the fact that it can be used within any formula to specify that from that point on the process is bisimilar with the process in the operator. One such property could be the property `semafterb` which is satisfied by the process `Example`.

```
defprop semafterb =
  <b>proc(SemSpec(get,put)) ;

defproc Example =
  select {
    b?().SemSpec(get,put) ;
    a?()
  } ;
```

The proc operator has some limitations which might not be obvious. The first such limitation is that recursive processes cannot be directly defined within the operator and need always be used as in the example above by first defining them on the top-level using the defproc command and them simply recalling that definition. The reason for this limitation is that infinite behavior processes are defined by calling the process' own definition recursively instead of having a recursion operator in the process language.

Usage of this operator is the preferred manner of checking for bisimilarity. Instead of using a `check Sys sim Spec` command, we encourage the more general, and equivalent, verification of `check Sys |= proc(Spec)`.

4.2.3.2 The <[]> modality

In the course of our work adding the characteristic formula to the SLMC tool we have introduced a new modality which might be interesting by itself. The modality was introduced to address an issue with the generated formula which we discuss in the next section. The general syntax for this modality is the following:

<[label]>(formulalist)

Where *label* is the same as depicted in figure 4.2 and *formulalist* is a comma separated non-empty list of formulae. This modality has the same meaning as:

<label>f1 and <label>f2 and ... [label](f1 or f2 or ...)

In short, this modality asserts that not only it is possible to do an action which will take the process to a state satisfying one of several formulae, it also asserts that on all transitions done from the process using that label it will necessarily end up in a state satisfying one of said formulae.

4.2.3.3 Recursive Formulae

In the previous version of SLMC one could only use previously defined formulae while defining a new formula. We have extended the aliasing functionality in the definition of formulae to allow creating mutually recursive formulae. For the purpose of defining these mutually recursive

formulae the keyword `andalso` is used. An example of this new functionality is given below, where we define properties `a` and `b` which state that a process alternates between performing an action `a` and an action `b` for all times. The process `AB` satisfies property `a`.

```
defprop a = <a>b
andalso b = <b>a ;

defproc AB = a!().b!().AB ;
```

4.3 Implementation Details

In order to correctly and efficiently implement characteristic formulae in SLMC several issues needed to be addressed. The main issue here is the exponential size of the formula, which unfortunately cannot be avoided. We have proposed a specific modality which makes characteristic formulae more compact, which somehow attenuates the problem into a more manageable form. Even with attenuated size, the formulae are rather large and repetitive and costly to verify using the model checker's previous straightforward algorithm. We optimized the checking algorithm by adding a cache, greatly speeding the verification time of characteristic formulae.

In the remainder of this section we will present in further detail how we verify the `<[]>` modality, how we implemented caching and recursive properties in the model checker and also how we build characteristic formulae.

4.3.1 Checking the `<[]>` modality

As we have argued, the characteristic formulae generated by the algorithm defined in chapter 3 grows exponentially with the number of states in the LTS of the process. This exponential growth is further aggravated by the fact that for each transition in the LTS state, the formula has two copies of the next state's characteristic formula, introduced by the formula being used in the *can* connective to indicate it must be possible to perform a transition to reach that state, and in the *must* connective which states that if a transition is taken it must reach a state bisimilar to one of the states reachable from the original process. The `<[]>` modality we propose combines these two aspects of the characteristic formula to provide a more compact way of representing these kind of formulae where there is no need to repeat the characteristic formulae of the immediate successors of the process. Another interesting aspect of combining the two modalities is that the property can be checked as a whole which may give rise to more efficient implementations for the modality which reflects in a more efficient verification of characteristic formulae.

Characteristic formulae can be seen as having three major components all connected by logical conjunctions. Here is a reminder of what these components are:

1. $<\alpha>s1$ for each transition from the process by action α leading to a state satisfying $s1$;
2. $[\alpha](s1 \vee ... \vee sn)$ for all states $s1 ... sn$ reachable from the process by performing action α;

Algorithm 4.1 <[]> Satisfaction

Require: *Props*: a set of formulae
Require: *lab*: an action label
1: $UnSat \leftarrow Props$
2: $Sat \leftarrow \emptyset$
3: **for all** P' such that $P \xrightarrow{lab} P'$ **do**
4: $\quad T \leftarrow \{x \in UnSat \mid P' \models x\}$
5: $\quad UnSat \leftarrow UnSat \setminus T$
6: $\quad Sat \leftarrow Sat \cup T$
7: $\quad$ **if** $T = \emptyset \wedge \{x \in Sat \mid P' \models x\} = \emptyset$ **then**
8: $\quad\quad$ **return false**
9: $\quad$ **end if**
10: **end for**
11: **return** $UnSat = \emptyset$

3. $[\alpha]\bot$ for every possible action which is not performed by the process.

The <[]> modality takes the form $<\alpha>(s1, ..., sn)$ where $s1, ..., sn$ are other formulae. The semantics for this modality is the same as $<\alpha>s1 \wedge ... <\alpha>sn \wedge [\alpha](s1 \vee ... \vee sn)$. It can be easily seen that this modality can be used in place of the first two components of the characteristic formula to reduce formula size by eliminating the redundancy present there.

The verification of the first component of the characteristic formula requires that all possible states reachable using α be checked in search of one satisfying the desired property. This verification is then repeated to all possible transitions in the original process. The second component of the characteristic formula is checked by going through all possible states reachable by performing α and checking that each one satisfies one of the required formulae. The third component of the formula is verified by checking that no transitions using those actions are possible.

It is not hard to imagine a way one could check the first component of the characteristic formula whilst checking the second component: all we need to do is keep a record of which formulae were used to satisfy the disjunction and in the end verify whether or not all of the formulae were used. This is the idea behind algorithm 4.1, which takes a set of formulae, an action label and a process P and checks whether or not $P \models <[lab]>(Procs)$.

We keep two sets of formulae: *UnSat*, the formulae not yet satisfied by the continuations we visited, and *Sat*, the ones which have been satisfied. *UnSat* is initially equal to all properties to be checked (line 1), while *Sat* is initially empty (line 2). For every process P' resulting from performing a *lab* action we will compute the set T of properties still in *UnSat* which are satisfied by that process (line 4). We exclude these properties from *UnSat* (line 5) and add them to *Sat* (line 6). If T is empty, meaning no property in *UnSat* is satisfied by P', and no property in *Sat* is satisfied by P', this means no property in $Props$ is satisfied by P' and P immediately fails to verify the property. Otherwise we continue to the next iteration. If all continuations by action

lab pass the above test and at the end *UnSat* is empty, this means all properties in the original set *Props* were satisfied by at least one of the P' processes. As all P' processes were able to satisfy at least one of the properties, P satisfies the *must* requirement in the semantics. From the fact that $UnSat$ is empty we know that no property in *Props* was left unsatisfied by at least one of the P' states, which means P also satisfies the *can* component of the `<[]>` modality.

4.3.2 Recursive Formulae

The SLMC tool had a functionality which allowed us to use the definition of another property in the process of making a new property. This is a great feature for property modularity and the fact that properties can be parametrized in names and other even formulae only made this an even more powerful tool.

This system, however, only worked with previously defined formulae in the sense that it was not possible to define two formulae simultaneously and make them depend on each other. These limitations relate to the way this functionality was implemented in SLMC, which was similar to the way macros are implemented in C. In the previous incarnation of the tool whenever a property was used in the definition of another property the alias would be replaced by the defined property with the proper substitutions made to it. This approach obviously denied the possibility of being able to define mutually dependent properties since that would lead to infinitely big formulae, or simply a loop in the algorithm. Syntactically the `defprop` command did not even have a way to define multiple properties in one go like the `defproc` command does. We have added the above mentioned functionality for properties and changed the way the system handles property aliasing.

In order to accommodate this feature in SLMC we needed to change the way the tool works in three fronts. These are the way aliasing is handled in the property syntax tree, the way we maintain property definitions to later access them and also the way we verify properties.

The reason behind changing how the definitions are kept is because we do not want name clashes to break previously defined properties. If we define property B which depends on property A and we later define another property named A, we do not want this change to affect the meaning of property B. We also do not want to repeat the definition of property A everywhere it is referred since that would defeat the space saving purpose of this functionality. The solution we found for the problem of how to store the property definitions so as to avoid the redefinition issue was to use a tiered definition environment inspired in the "spaghetti stack" environment much used in functional programming implementations. An environment consists of a set of local property definitions (pairs associating a name to a property) and a reference to another environment. The topmost environment holds no definitions. Name lookup works its way through the stack by first checking if the definition exists in the first environment, and if it does not it will recursively check the previous environment. The last, empty, environment will not have a parent environment and the lookup will fail should it be reached. With this kind of environment, old definitions are not replaced but rather pushed back on the stack, which means they are still accessible, provided we start the lookup in the same environment they were first defined in.

As was mentioned previously, property aliasing in SLMC was treated as a macro at the syntax level: basically the alias was replaced by a copy of the definition after having all formal parameters replaced by the actual arguments. We changed how this works in order to keep the alias "call" in the syntax tree, but adding some information to it. To avoid loosing the information of exactly which property is mentioned in the alias, we add the current environment to the information kept by the alias in the syntax tree. This way, if the environment kept in the property is used, posterior redefinitions of a property with the same name will not interfere with the meaning of the current property.

Verification of recursive behavior was already present in the SLMC tool. Allowing recursive behavior to be defined using property aliasing was implemented as a sort of syntactic sugar for fixed points. In essence, the way we handle aliasing is by inserting a maximum fixed point with a fresh propositional variable the first time some property alias occurs, and replacing further occurrences of that same alias with a call back to the propositional variable. We do not actually change the formula we are checking, but instead access the environment stored within the alias, obtain the property defined by that alias, do the proper substitutions given the parameters which were passed and proceed with checking whether the process satisfies this updated copy of the aliased property. The matter at hand is a little more delicate than it seems at first, though. In order to identify whether an alias was already used we cannot just use the alias' name, we must use all information used to generate that particular fixed point, which means we need to also take into consideration the names and properties passed as arguments to the alias. In order to maintain the information required to correctly process these recursive formulae we added another parameter to the checking procedure which holds a map relating aliases (including their parameters) to propositional variable names.

4.3.3 Caching in the Model Checker

Combining the `<[]>` modality and the succinct definition of recursive formulae we are able to greatly reduce the space required to represent characteristic formulae so that it scales linearly with the number of LTS states and transitions instead of exponentially. Even with these gains in space we did not eliminate the necessity to actually go through all the possible transitions and states, thus making verification still potentially requiring an exponential amount of time to finish.

This exponential time required to finish checking a characteristic formula stems from the need to verify all paths from all possible transitions. The more ways there are to get to a certain state, the worst this problem manifests itself. This is the same issue we had with exponentially sized formulae but is here translated to the amount of time required to verify a formula instead of manifesting itself on the formula size. Since the main problem is checking whether different paths to the same state are equivalent, we can greatly benefit from not having to verify the (redundant) characteristic formula for that state every time we reach it. In order to remember the result of previous visits to a state, we implemented a cache on top of the model checking algorithm.

The cache we implemented operates using pretty standard memoization [20] techniques. We altered the checking procedure to inquire a cache whether a result is already known for the current process-property pair, if it is use that previously cached result, if it is not check the property normally and then enrich the cache with this new information.

Conceptually our key consists of a process-property pair. In practice, however, the implementation is a bit more complex than that to ensure correctness and efficiency of the caching algorithm. However, our solution includes additional information in the key due to the way quantifiers (`exists` and `forall`, for example) are handled by the verification algorithm. These quantifiers are checked by generating a list of possible names for the variable and checking the following property as if the bound name in it was replaced by the current name being tested. For all practical applications this equates to having a different property for each of the generated names, and our cache needs to be able to handle properties in this manner. Our solution consists in expanding the key used in our cache to a triple consisting of the property, active name binds and process. Without this measure a situation could arise where the validity of an `exists` property could be thwarted simply by the fact that the first tested name substitution happened to not yield a positive result (a similar case could happen with a `forall` statement where the first tested name substitution lead the property to be true while only some other substitution revealed the falsehood of the overall statement).

The cache system is organized in a three-tiered hash table which firstly indexes using the property. For each property we will have an hash table holding results for each name substitution ever encountered for that property which in turn will index an hash table which finally holds the truth value associated with this entry. This final hash table will be indexed by the process.

Looking up a cache entry requires hashing and comparing three entities: a property, a list of pairs of names (associations instantiating the first with the second) and the internal representation of the current state of the process being analyzed. Since the name associations which will be used as well as the process states which are visited depend greatly on how the state space is visited to check the validity of a property in regards to a process, there is little use in pre-processing these items since we could end up doing a lot more processing than the necessary to verify a given property. The properties are of a different, more static, nature and can be pre-processed in order to avoid deep structural comparison between them. Instead of having the system do an in-depth traversal of the property to calculate their hash value and then compare properties (in-depth again) for equality, we can assign an identifier to each sub-formula in the property and use the much lighter integer comparison and hashing to do our work for us. Since at this stage of the cache system properties are static, we incur in no complications by doing this simplification. These integer identifiers for sub-formulae are generated in a linear time pass through the formula structure and each abstract syntax node is tagged with their respective identifier. To avoid cluttering the model checker with global information about which formulae have been tagged and which are their integer identifiers, we have chosen to perform this tagging every time a property is about to be checked. This enables that the metadata required for tagging the formula abstract syntax tree can be of a temporary nature. The attribution of integer numbers to the formula being checked is a similar approach to the one used by Steffen and

Cleaveland in [10] to optimize the performance of the model checker for the alternation-free modal mu-calculus.

4.3.4 Building a Characteristic Formula

The description of how to build a characteristic formula on chapter 3 is based on the LTS of a process and not on the process itself. In our implementation of the algorithm for building the characteristic formula we adopt the same approach by working on an abstract representation of the process' states instead of working directly on top the actual process representation. The gains we get from this approach are not only that we get a generic implementation which could work with any other process system (provided we are able to build the LTS from that system), but also that we can work the LTS in order to obtain a more succinct characteristic formula for the process.

In our implementation we use three phases to build the characteristic formula of a process. We first convert the process into a Labelled Transition System (a set of states and transitions between them), we minimize the states and transitions in that LTS, and from the minimized LTS we build the characteristic formula.

The LTS data structure we use is a union type with two possible types of values. A `Void` type which represents the empty process with no transitions and a `Node` which can contain a list of references to other nodes where each reference is tagged with an *action*. An action is another union type which can take either the form of a τ or a transition with information about the name of the action and the action type. The purpose of this LTS data structure is building a graph which represents the actual LTS from the process we are analysing.

Some of the model checking operations used an iterator structure to obtain the continuations of a process state after a given action. We have altered some of this iterator structure to suit our needs by allowing the iterator for all possible actions to also export the actions which are performed to reach a state. This way, by following all possible paths without repeating the process for previously visited states, we can extract the full LTS of a process using a lot of the machinery which was already present in the SLMC tool.

The LTS we build from the above procedure will have all the same transitions as the process which originated it and will have as many states as structurally different configurations are found by the extraction procedure. We take two steps to minimize the size of the LTS while maintaining it behaviorally equivalent. A process like `a.0 + a.0` will contain two transitions to the inactive process by means of an `a` action, which are redundant transitions. We perform a first minimizing step by identifying these easy to spot redundancies and collapse them into a single transition. The second step we perform to minimize LTS size is by the application of Algorithm 4.2, which obtains a partition refinement of the LTS, grouping behaviorally equivalent states in the same partition, which then translates to a state in the final LTS. This minimization of the LTS serves the purpose of making the characteristic formula as succinct as possible thus avoiding redundant verifications while checking for bisimularity in our approach.

After minimizing the LTS we proceed to building the characteristic formula. In fact, what

Algorithm 4.2 Partition Refinement

Require: Q is the set of states
Ensure: P is the partition refinement of Q
  initial partition $P = \{Q\}$
  $change \leftarrow true$
  while $change$ **do**
    $change \leftarrow false$
    if exits partitions $T_1, T_2 \in P$ and action α such that
    $T_{1,1} = \{q \in T_1 \mid \exists q' \in T_2, q \xrightarrow{\alpha} q'\}$ and $T_{1,2} = T_1 \setminus T_{1,1}$ are both nonempty **then**
      $P \leftarrow (P \setminus \{T_1\}) \cup \{T_{1,1}, T_{1,2}\}$
      $changes \leftarrow true$
    end if
  end while

we do is build a characteristic equation system using recursive formulae based on the way Steffen described his characteristic formula. We assign one fresh property name for each state in the LTS and define the property as the conjunction of the elements present in Steffen's characteristic formula. The main difference here is that instead of expanding the characteristic formula for the behavior after a transition we just fill the spot with the property regarding the state that is reached. Since we minimized the LTS using partition refinement, these states will be unique with regard to the behavior of the process. By adopting the equation system approach we eliminate a lot of redundancy in our characteristic formulae and also remove the need to add a special mean to handle recursive behavior in the LTS, it is handled by the model checking algorithm instead of by the characteristic formula generation method. A detail that further helps with both the memory size required for holding a characteristic formula and also helps in speeding its verification is the fact that we use our `<[]>` modality in the appropriate places when describing a state's behavior. Due to the formulae themselves being able to keep references to property environments and also due to the environment kept in a formula having priority in the lookup procedure, we can create a new environment for each characteristic formula and pass it over to as a part of the characteristic formula (which will consist of a call to the property generated for the state representing the start of the process being analyzed), which keeps clutter away from the model checker environment.

4.4 Evolution

The implementation presented on this chapter for building and efficiently checking characteristic formulae in SLMC was by no means our first approach to the issue and we went through some intermediate solutions. After a few iterations we got to the current state of the implementation, but we would like to provide an overview of the evolution of the solution throughout the

execution of the thesis. In this section we will describe the major milestones stepped during the elaboration of this thesis, their issues and how they were solved.

Firstly we implemented the algorithm proposed by Graf and Sifakis without any kind of partition refinement or transition prunning. This solution led to very big characteristic formulae which was basically untreatable. The time and space required to check all but the simplest of characteristic formulae was immense, which made it obvious we needed some way to better represent characteristic formulae.

The characteristic formulae at this time had a great deal of redundancy which was introduced by the doubling of formulae for continuation states. This way of creating the formula also introduced redundancy in the checks which were verified, where the characteristic formula for each state needed to be verified twice: once to satisfy the *may* component of the characteristic formula and another to satisfy the *must* component. To eliminate the spatial and procedural redundancy present in this implementation we proceeded with the implementation of the `<[]>` modality. The use of this modality in our characteristic formulae visibly decreased the number of processes which required too much space to be treatable, but even with the algorithm specially tailored for the modality's semantics some processes generated characteristic formulae which took too long to verify with our methods. The amount of time it took to verify these characteristic formulae was mainly due to the redundant paths to reach a certain state and that leading to the characteristic formula for that state being checked against it repeatedly.

The next milestone in our solution was the implementation of the cache in the model checking algorithm. The first version of the cache we implemented had success speeding up the verification of the characteristic formulae we were experimenting with at the time to manageable levels. The verification time of the examples we were testing no longer became the issue with this implementation, but the issue shifted towards the formula size again. We would get relatively small processes generating such a big characteristic formula that it took an enormous amount of time to generate it, even if verification time was practically instant (due to the cache). It was also around this time that we optimized the way we generated characteristic formulae by pruning superfluous transitions and by partition refining the LTS. These measures did not prove sufficient to reduce the formula size to a treatable level.

Since the main factor in the formula size was the amount of redundancy introduced by the different ways to get to a certain state, it was clear that having a unique representation for the state and only one definition for its behavior would greatly improve the spatial performance of our solution. The results with this approach were very satisfactory for characteristic formulae, however the cache system we had in place was severely underperforming for certain types of formulae. We remind that the cache system is in place for the whole of the model checking algorithm and not just for characteristic formula verification. This meant the cache should at the very least not hinder the tool's performance, which it did in some cases.

We then proceeded to analyzing why the cache was underperforming so much and took measures to alleviate the problem. In this stage we needed to rework both the way the cache was organized and what algorithms were used in indexing and comparing processes. These changes came to the current solution which is not yet optimal as there are still some cases in

which not using the cache is better than using it (and we do have a system parameter to switch it on or off), but the performance penalty is no longer prohibitive like it used to be.

5. The Jobshop Example

On this chapter we provide a more complex example for the usage of our approach to verifying bisimilarity between processes. We will use the jobshop example which Milner used to illustrate CCS in [21]. This example is fairly more complex than the simple semaphore examples provided thus far since not only it has a bigger set of states but also because it contains name restriction which means many of its states are not immediately distinguishable.

5.1 Description

The example we will be working with in this chapter aims to represent the behavior of a workshop. In the modeled workshop, which we will actually call Jobshop, we have two workers and two tools, a mallet and a hammer. Due to spatial restrictions it is not possible for a worker to be performing two jobs at the same time. The jobs presented to the workers can be of many kinds but they can be categorized into three difficulties: easy, normal and hard jobs. The easy jobs can be done by the workers with their bare hands, while the hard jobs will require the use of a hammer. The normal jobs can be managed by the use of either a mallet or a hammer. It should be obvious that when a tool is being used by one worker, it cannot be used by any other at the same time.

5.2 Implementation

With the objective of analyzing this system within our tool we need to first describe the system with appropriate language and abstractions. While the tool supports a variation of the π-calculus, the extensions made by this work do not support name passing which means we are limited to using CCS processes for modeling the system. We model the behavior of both tools and the workers. The tools can be defined as

```
defproc Mallet(getm,putm) =
  getm?().putm?().Mallet(getm,putm) ;

defproc Hammer(geth,puth) =
  geth?().puth?().Hammer(geth,puth) ;
```

This representation of the tools used in the jobshop works like a semaphore in the sense that the tools are resources and their users post a message when they wish to utilize a tool (`getm` and `geth` for the mallet and hammer, respectively) and another one (`putm` or `puth`) when they no longer need it. As with most protocols we expect processes to behave properly, meaning they will not use `putX` without having first taken the tool by performing the respective `getX`.

Our implementation of the jobbers in the jobshop takes these concerns in consideration, and is defined as

```
defproc Jobber(getm,putm,geth,puth) = select {
  inEasy?().outEasy!().Jobber(getm,putm,geth,puth) ;
  inMed?().StartMed(getm,putm,geth,puth) ;
  inHard?().geth!().puth!().outHard!().Jobber(getm,putm,geth,puth)
} and StartMed(getm,putm,geth,puth) = select {
  getm!().putm!().outMed!().Jobber(getm,putm,geth,puth) ;
  geth!().puth!().outMed!().Jobber(getm,putm,geth,puth)
};
```

In the formulation we make for jobbers they will be waiting for a message indicating the arrival of a new job and what is the job's difficulty (easy, medium or hard). After getting an easy job, the jobber will perform it without further communication until the job is done and he sends notification of that fact via a message through the `outEasy` channel. If the job the jobber takes is hard he will proceed to request the hammer, perform the job without communication, return the hammer by sending a message in the `puth` channel, and then announcing on the `outHard` channel that the job is complete. When confronted with a request for a normal difficulty job, the jobber will non-deterministically get either a mallet or a hammer, perform the work, return the same tool he took and announce he completed the job. After performing any job the jobber will resume taking new jobs to perform.

We have so far defined all the elements composing our jobshop, but we have yet to put them all together to form the actual system representing he jobshop. Here is how we accomplish that

```
defproc Jobshop =
  new getm, putm, geth, puth in (
    Hammer(geth,puth) | Mallet(getm,putm) |
    Jobber(getm,putm,geth,puth) |
    Jobber(getm,putm,geth,puth)
);
```

The jobshop system consists of a hammer, a mallet and two jobbers composed in parallel giving the jobbers access to the tools required for the normal and hard difficulty jobs. The system is thus self-contained and will be able to accept, perform and eventually complete all of the jobs asked of it. We restrict the names used to access the mallet and hammer in our jobshop disallowing their use from other processes and also ensuring the jobbers do not use any other tools except the ones in the jobshop. The `new` operator from the π-calculus adopted by the SLMC tool restricts the scope of the `getm`, `geth`, `putm` and `puth` names by creating private channels for the purpose of the jobshop. Like the Jobber process, this system will be waiting to receive jobs of varying difficulty. The system will be able to take up to two parallel jobs and progress in their execution based on the available internal resources. The only visible actions

made by the system are the obtaining of jobs and the signaling of their completion, and all other internal steps in acquiring and releasing tools will be τ actions which are not observable.

5.3 Verifications

In the previous section we modeled the Jobshop system by describing its components and arranging them properly. In this section we will analyze how changing one of the components affects the system in terms of how its behavior is perceived.

5.3.1 Positive Example

The first scenario for verification consists of replacing the jobber by one who uses a mallet instead of a hammer to perform hard jobs. The second scenario will be a change in the jobber's behavior in which they hold on to the hammer, instead of releasing it right after completing the job, until an easy job comes by since that requires the jobber's hands to be free.

We start by defining a property which represents the original system's behavior, which is done by the command

```
defprop sys = proc(Jobshop) ;
```

A system with the same behavior as the `Jobshop` process will satisfy the `sys` property as long as it does not use any channel which is not present in the original jobshop as these actions are not contemplated by the characteristic formula. As a sanity test we can check that the process `Jobshop` itself satisfies this property, by running the command

```
> check Jobshop |= sys ;
* Process Jobshop satisfies the formula sys *
```

The first change we make to the system is replacing the jobbers for ones using a mallet instead of a hammer for performing hard jobs. We can accomplish this with the following `Jobshop2` process

```
defproc Jobshop2 =
  new getm, putm, geth, puth in (
    Mallet(getm,putm) | Hammer(geth,puth) |
    Jobber(getm,putm,geth,puth) |
    Jobber(getm,putm,geth,puth)
  ) ;
```

The difference between this process and the `Jobshop` is that the names are passed to the `Jobber` process so that the name used to get the tool for the hard jobs is the name passed to the

Mallet process. This name switch does not affect the handling of normal jobs since the way the jobber uses the tools is completely symmetric, meaning the change only affects the order of the choice construct, not its meaning. The resulting sub-process is literally the same as it would be in the case that the choices had been specified in a different order.

We expect this system to be bisimilar to the Jobshop process. The effect of these changes is that the tools' roles are reversed in the sense that a jobber will have to wait for a mallet (previously hammer) to be available for a hard job, and the jobber doing the medium job can just take a hammer (previously mallet) if the mallet is being used by the other jobber. The dynamics of the system do not change and since the precise actions are undistinguishable from outside the process, this system provides the exact same behavior as the original jobshop did. Our tool can confirm this expectation by checking that the Jobshop2 process satisfies the sys property. One good way to guard against unintended introductions of names which are used is to perform the symmetrical verification, which is, that Jobshop satisfies the characteristic formula of Jobshop2.

```
> check Jobshop2 |= sys ;
* Process Jobshop2 satisfies the formula sys *
> check Jobshop |= proc(Jobshop2) ;
* Process Jobshop satisfies the formula proc *
```

5.3.2 Negative Example

In our second example we change the jobber's behavior to not release the hammer after getting it until an easy job is asked of them. This will not only prevent one jobber to have the hammer forever once he acquired it, but it also makes some sense in the context of the jobshop since the jobber who does not need a tool to perform an easy job, can just place it down and free up both his hands for the job. The process representing this new jobber and the jobshop we define with it are shown next

```
defproc Jobber2(geth,puth,getm,putm) =
  select {
    inEasy?().outEasy!().Jobber2(geth,puth,getm,putm) ;
    inHard?().geth!().outHard!().JobberH(geth,puth,getm,putm) ;
    inMedium?().DoMedium2(geth,puth,getm,putm)
  }
and DoMedium2(geth,puth,getm,putm) =
  select {
    geth!().outMedium!().JobberH(geth,puth,getm,putm) ;
    getm!().putm!().Jobber2(geth,puth,getm,putm)
  }
```

```
and JobberH(geth,puth,getm,putm) =
  select {
    inEasy?().puth!().outEasy!().Jobber2(geth,puth,getm,putm) ;
    inHard?().outHard!().JobberH(geth,puth,getm,putm) ;
    inMedium?().outMedium!().JobberH(geth,puth,getm,putm)
} ;

defproc Jobshop3 =
  new getm, putm, geth, puth in (
    Mallet(getm,putm) | Hammer(geth,puth) |
    Jobber2(getm,putm,geth,puth) |
    Jobber2(getm,putm,geth,puth)
  ) ;
```

This version of a jobber starts out like the previous definition but after acquiring a hammer, by using the `geth` message, it will not use the `puth` message indicating the hammer was released. Instead it performs the job, announces that he finished it and proceeds to a state where medium and hard jobs can be performed immediately without picking up any additional tools, while easy jobs need to release the hammer and only then perform the job.

We expect this altered jobshop to not be bisimilar with the original version since there are some aspects of the internal behavior which are notoriously different. Namely, after a hard job is completed it is possible, while not certain, that the next hard job will be done without a single internal step, which the possibility does not exist in the original jobshop at all. This change on system behavior is enough to warrant non-bisimilarity between the two systems. Another example of behavior distinguishing the two systems is that on `Jobshop3` it is possible to perform the action stating that a medium job is finished only one τ after the job was requested. A property describing this is `dist`, defined as

```
defprop dist = <inMedium><tau><outMedium>true;
```

which will be satisfied by `Jobshop3` and not by `Jobshop`, as shown in the following session

```
> check Jobshop3 |= dist ;
* Process Jobshop3 satisfies the formula dist *

> check Jobshop |= dist ;
* Process Jobshop does not satisfy the formula dist *
```

Since we can devise a behavioral property which is able to distinguish the two processes (one that is satisfied by one and not the other), we already have sufficient proof that the systems are not bisimilar. Using the `sys` property we defined earlier we can reach the same conclusion.

```
> check Jobshop3 |= sys ;
* Process Jobshop3 does not satisfy the formula sys *
```

6. Closing Remarks

The Spatial Logic Model Checker tool is one of many tools in the area of software verification. This tool allows the verification of software systems by means of model checking, where you build a representation of the system in a simplified language (a process calculus-like language in the specific case of the SLMC tool) and verify that the model representation of the system satisfies, or not, some properties defined in a logic suited for the properties we wish to prove about the system. Another way to reason about software systems is when we build a process modeling the desired behavior (a specification) of our system and we build one which depicts as best as possible its actual implementation. We can then use forms of equivalence to determine whether or not the implementation conforms with the specification, thus verifying its correctness.

The focus of the work presented in this thesis is to incorporate behavioral analysis into the SLMC tool, and doing it using the model checker's capabilities for reasoning about the behavior of the process. The main idea is to take the process and build a characteristic formula which accounts for all of the process' behavior. If a process is able to satisfy said formula, it will exhibit the same behavior and thus be equivalent to the process originating the formula.

This approach to checking bisimularity relations between processes has been studied before but since it leads to formulae with an exponential blowup with regard to the size of the Labelled Transition System for the process, the idea is not used in any of the most widely known tool which can verify bisimilarity. Most tools which support checking for behavioral equivalence between processes do so by using partition refinement algorithms which refine the LTS of both processes being compared based on the desired equivalence. While there is nothing wrong with the way these tools perform these equivalence comparisons we wanted to give characteristic formulae a chance since they give rise to an interesting way to define properties, which enriches our logic. Using this notion of characteristic formulae we can use processes directly to define a property, or part of a property, which indicates that from that point on the process being checked should behave exactly like the process describing the property.

We have taken on the challenge of implementing characteristic formulae for our model checker and make their verification competitive with other approaches for equivalence testing. In order to overcome the challenges posed by characteristic formulae we needed to modify the model checker to be able to define mutually recursive properties, to cache previous visited results while checking a property and also to introduce a modality which more closely describes the intents of a characteristic formula. These new functionalities worked together to make characteristic formulae a viable way to verify behavioral equivalence in our model checker, and also serve as means to provide additional expressiveness to the logic used by the SLMC tool.

The implementation we have obtained works pretty efficiently for characteristic formulae. We have, however, some issues with the cache implementation when it comes to verifying some kinds of properties. On some cases, namely properties which rename restricted names (`inside` and `hidden`), the caching algorithm incurs more overhead than desired. The current implementation minimizes this issue to a point where the tool is still usable, although not optimal.

6.1 Future Work

With confirmation that using characteristic formulae yields good results for the verification of bisimilarity relations between CCS processes we have opened way for more complex uses of this technique. In this section we will enumerate some ideas on how to follow up on the work done for this thesis and provide some preliminary discussion on the issues that might arise when pursuing those objectives.

The more urgent and obvious future work to do as a follow up of this is the implementation of a caching algorithm which performs well for all of the logic available to the SLMC tool. We have changed the way the cache is organized several times in the course of the implementation of the functionalities described in this thesis. We have also changed what notion of equality is used for the processes present in the cache. On our first version of the cache we used structural congruence and the hash function used in the implementation of process sets for the verification of fixed points. This provided poor results with the cache organization we had, since the hash function was not very discriminative and a many processes clashed leading to bigger collision lists on the hash table. We switched to the current implementation where we use a more discriminatory hash function which takes much more information into account, but also where the equality function is more superficial not taking possible permutations into account for example. This latest combination provides the best results we have encountered so far, but is far from perfect and we should strive to implement a cache which performs better on a wider range of situations.

Bernardo Toninho's Masters thesis revolved around the implementation of another extension to the SLMC logic and tool [28]. On this work the SLMC logic was enhanced with some epistemic and knowledge modalities which aim to allow reasoning about security properties using the SLMC tool. It could be interesting to integrate the two extensions into one coherent tool. The work performed by Toninho involved not only adding new modalities to the logic but also an in-depth change of the process' internal representation (since they were changed to be able to transmit terms instead of just names). The major issue with the integration of these two extensions would be integration of the term system into the cache in our work. The integration of these two works is a little dependant on which algorithms are used to cache and how well they integrate with Toninho's work.

While strong bisimulation is a good tool for testing out the applicability of the approach we selected, the most used form of equivalence relation for the purpose of checking a system's correctness is actually weak bisimulation (the bisimulation relation which somehow ignores silent τ actions when comparing two processes), mostly due to the fact that the specification can be much more abstract than when we are using strong bisimulation. For this reason, and to make our tool usable in more scenarios, it would be interesting to adapt the approach to encompass the possibility of generating characteristic formulae for weak bisimulation instead of strong bisimulation. The implementation of this extension would not be trivial since characteristic formulae for weak bisimulation is not so direct to obtain as for strong bisimulation. Some approaches to this issue use weak versions of the *may* and *must* modalities. A weak *may* modality can be

rather easily devised by adding the possibility to perform silent actions before the action and after it before reaching the state satisfying the continuation formula. A weak *must* modality, however, is not so trivially achieved. A property for a weak *must* modality should state that every silent transition which eventually leads to the action must then eventually also lead to a state satisfying the continuation formula, which is much harder to express. Another approach suggested to characterizing weak bisimulations is using strong bisimulation characteristic formula to define the property being checked but when the actual verification is made the process being verified (or at least its LTS) is modified so that silent transitions are eliminated. This solution could be achieved by changing the iterators being used by the model checking algorithm, but that could have strange interactions with the rest of the algorithm which need to be identified and analyzed.

The expressive power of CCS is not as powerful as the one from π-calculus and as such the specification of some systems might benefit more from the full strength of our process language instead of just the CCS subset of it (without the name passing of the π-calculus). For these cases it would be beneficial for the tool to be able to provide the same functionality as in the case of CCS processes and as such there is interest in expanding the approach to the π-calculus. The step towards this solution might not be as easy as one may initially think. LTS systems for π-calculus processes are more complicated than the LTS system for CCS since they need to account for the various possibilities of name input and output which may alter the course of the computation. This would also be an issue in the way we represented the characteristic formula since not all paths would be linear and all paths would need to be accounted for.

Bibliography

[1] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980.

[2] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[4] L. Caires. Behavioral and spatial observations in a logic for the pi-calculus. In Igor Waluzievich, editor, *Proceeding of the Conference on Foundations of Software Science and Computation Structures FoSSaCS 2005*, number 2987 in Lecture Notes in Computer Science. Springer-Verlag, 2005.

[5] Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.

[6] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[7] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*, pages 30–33. MIT Press, 1999.

[8] Jean claude Fernandez. Ald'ebaran: A tool for verification of communicating processes. Technical report, 1989.

[9] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[10] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 48–58, London, UK, 1992. Springer-Verlag.

[11] Jean-Claude Fernandez and Laurent Mounier. Verifying bisimulations "on the fly". In *FORTE '90: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 95–110, Amsterdam, The Netherlands, The Netherlands, 1991. North-Holland Publishing Co.

[12] Robert. W. Floyd. Assigning meanings to programs. *Proceedings of Symposia on Applied Mathematics*, 19:19–31, 1967.

[13] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of cadp 2001. Technical report, 2001.

[14] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Wolper,, Pierre.

[15] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[16] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[18] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[19] Paris C. Kanellakis and Scott A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 228–240, New York, NY, USA, 1983. ACM.

[20] Donald Michie. Memo functions and machine learning. *Nature*, 1968.

[21] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[23] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[24] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[25] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[26] J. Sifakis S. Graf. A modal characterization of observational congruence of finite terms of ccs. In *Automata, Languages and Programming*, pages 222–234. Springer-Verlag, 1984.

[27] Bernhard Steffen. Characteristic formulae. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 723–732, London, UK, 1989. Springer-Verlag.

[28] Bernardo Toninho. A spatial-epistemic logic and tool for reasoning about security protocols. Master's thesis, FCT/UNL, 2009.

[29] Björn Victor and Faron Moller. The mobility workbench - a tool for the pi-calculus. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 428–440, London, UK, 1994. Springer-Verlag.

[30] H. T. Vieira and L. Caires. The spatial logic model checker user's manual. Technical report, Departamento de Informática FCT/UNL, 03 2004. URL=http://www-ctp.di.fct.unl.pt/SLMC/v0.9/manual.pdf.

[31] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref – a symbolic bisimulation tool box. In Susanne Graf and Wenhui Zhang, editors, *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *Lecture Notes in Computer Science*, pages 477–492, Beijing, China, October 2006. Springer-Verlag.