Autran Macêdo

Solução Exata de Problemas de Escalonamento Determinísticos por Meio de Verificação Exata de Modelos

Tese apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Belo Horizonte, Minas Gerais, Brasil

Universidade Federal de Minas Gerais - UFMG

2002

Thesis Committee

This thesis was presented and approved at August, 30, 2002.

Prof. Sérgio Vale Aguiar Campos (Advisor)

Prof. Carlos Roberto Venâncio de Carvalho

Prof. Henrique Pacca Loureiro Luna

Prof. Geraldo Robson Mateus

Prof. Edjard de Souza Mota

Prof. Edmundo Albuquerque de Souza e Silva

Dedicatória

A
Deus, o farol
Heribaldo e Lindinalva, o barco
Ricardo, a vela
Ricardo, Renê, Mariana e Daniela, as estrelas
Luís Carlos Ríspoli, o mar sereno
Vza, a brisa

Agradecimentos

Uma seção de agradecimentos, em geral, simboliza o término de um trabalho. Confesso que, em alguns momentos, temi não chegar a esta seção. Contudo, vários fatores contribuíram para que eu lograsse êxito nesta empresa. Deus acima de tudo foi minha fonte de fé: Ele não dá a seus filhos um fardo que estes não possam carregar. Assim, acreditei e não me desviei do caminho.

Ao longo do caminho, deparei-me com várias pessoas. Algumas eu já conhecia outras não. Todas elas, direta ou indiretamente, me auxiliaram no doutorado. Quero expressar aqui a minha gratidão a todas elas nominando-as uma a uma, tanto quanto me é possível lembrar o nome de tantas pessoas (jurídicas e físicas).

O governo brasileiro através da CAPES apoiou-me finaneiramente ao longo do doutorado. A Universidade Federal de Uberlândia (UFU) e a Faculdade de Computação (FACOM) ofereceram-me todas as condições para que o meu doutoramento fosse realizado. A Universidade Federal de Minas Gerais (UFMG) e o Departamento de Ciência da Computação (DCC) tornaram disponível os recursos necessários à realização de meu trabalho.

Encontrei excelentes professores ao longo do doutorado. Alguns deles serão sempre referências em minha vida profissional e pessoal: Berthier Ribeiro, Carlos Venâncio, Claudionor Coelho, Edjard Mota, Edmundo Silva, Frederico Campos, Henrique Pacca, Newton Vieira, Robson Mateus, Sérgio Campos.

Andréa Iabrudi, Adriano César, Camillo Jorge, Carlos Frederico, Denilson Barbosa, Gilberto Miranda, Gurvan Huiban, Hervaldo Sampaio, Hugo Barros, Ilmério da Silva, Jones Albuquerque, José Pio, Karla Borges, Linnyer Beatrys, Manoel Palhares, Marco Cristo, Maria de Lourdes, Mark Allan, Paulo Rodrigues, Pável Calado, Ricardo Poley, Silvio Jamil partilharam comigo momentos inesquecíveis de cunho acadêmico e/ou pessoal.

Anilton Joaquim, Cláudio Camargo, João Augusto Pacheco, Luís Fernando Faina, Marcelo Rodrigues, Márcia Aparecida, Sandra de Amo, Silvânia Azevedo, Valéria Bar-

ros seguiram-me à distância. Por meio de mensagens eletrônicas, telefonemas e alguns contatos pessoais esporádicos recebi deles menções de apoio, otimismo e carinho.

Alexandre Dias, Antônia Rocha, Belkiz Costa, Emília Soares, Geraldo Oliveira, Gilberto Luiz Costa, Gustavo Oliveira, Helvécio Lopes, Maristela Soares, Luciana Costa, Renata Viana, Túlia Andrade, por meio de seus respectivos trabalhos, tornaram minha vida muito mais confortável e agradável no DCC.

Algumas pessoas tiveram uma relação muita próxima com o trabalho de tese. João Paulo Kitajima foi quem me incentivou a iniciar o doutorado, foi meu primeiro orientador e foi quem me propiciou minhas primeiras viagens ao exterior. Sérgio Campos, meu orientador, inspirou-me motivação e confiança. Suas intervenções sempre corretas ajudaram-me a encontrar o rumo da tese e manter-me focado no mesmo. Sérgio possui um estilo de escrita direto e fluido que determinou meu estilo de escrita atual. Hervaldo Sampaio, médico e contemporâneo de doutoramento no DCC, dispendeu parte de seu tempo para pesquisar artigos médicos sobre Refluxo Vésico-Ureteral, uma anomalia que minha filha Daniela teve no início de sua infância. Gurvan Huiban, parceiro de república e de doutorado, ajudou-me com experimentos com a ferramenta CPLEX. Hugo Barros ajudou-me com a programação de uma ferramenta para geração automática de modelos SMV. Adriana Mariano ajudou-me a manter-me focado no doutorado.

Finalmente, gostaria de dizer algumas palavras sobre pessoas muito especiais para mim. Eu tenho pais maravilhosos, que são minha fonte de inspiração. Ao longo de todo o tempo de doutorado, sempre estiveram presentes em minha mente e em meu coração. Eu tenho um irmão, que acima de tudo é meu melhor amigo. Eu tenho quatro filhos que são a razão de minha vida, a razão do meu respirar. Eu tenho um amigo, Luís Carlos Ríspoli, que desencarnou num acidente automobilístico. A todos eles peço desculpas pela minha ausência durante esses últimos anos ... E no final de tudo, eu a encontrei. Ela passou em minha vida como a brisa da manhã, suave e fugaz, mas foi suficiente para se tornar inesquecível.

Obrigado, meu Deus, por tudo: pela força, pelo auxílio, pela esperança, pela saúde, pelo sucesso, pelo insucesso, pela minha família, pela brisa ...

Resumo

Problemas de *Scheduling* estão relacionados com o sequenciamento de um conjunto de atividades a serem executadas por um conjunto de recursos ao longo de um determinado peíodo de tempo. Este tipo de problema possui diferentes configurações. Neste trabalho focamos uma classe de problema conhecida como problemas determinístico-estáticos. Nós apresentamos Verificação Simbólica de Modelos (VSM) como uma nova abordagem de solução para esta classe de problemas.

VSM é uma técnica de verificação formal que tem alcançado êxito na verificação de diferentes sistemas complexos. Nós temos utilizado VSM para dar solução exata a problemas determinístico-estáticos do tipo *job-shop* e *flow-shop*. Ao longo deste trabalho, nós apresentamos a modelagem e os resultados que obtivemos na solução destes problemas. Finalmente, nós apontamos também direções futuras para este trabalho.

Abstract

Scheduling relates to ordering a set of activities to be performed by a set of resources over a period of time. Scheduling ranges over a very large variety of problems, however we are concerned with deterministic static scheduling problems. In this thesis, we present Symbolic Model Checking (SMC) as a new approach to solve this class of problem.

SMC is a finite state formal verification technique that has been successfully used to verify many complex systems. We have been using SMC to give exact solution to deterministic static scheduling problems, such as job-shop and flow-shop problems. We present the modeling and the results we have obtained solving some instances of these problems, and finally, we point out future directions to this work.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Traditional Approaches 2

1.3 Symbolic Model Checking 3

1.4 The Proposed Approach and Results 3

1.5 Contributions 6

1.6 Organization of the Thesis 6

2 *Symbolic Model Checking - SMC* **7**

2.1 Model Checking - MC 8

2.1.1 Kripke Structure 9

2.1.2 CTL - Computation Tree Logic 12

2.1.3 A Microwave Example 16

2.1.4 CTL Model Checking 17

2.2 Symbolic Model Checking - SMC 18

2.2.1 Ordered Binary Decision Diagrams - OBDDs 19

2.2.2 The Model 19

2.2.3 Search Algorithms 21

2.2.4 SMV - An OBDD Symbolic Model Checking Tool 22

2.2.5 Verus - A Symbolic Model Checking Tool for Real Time Systems . . 25

3 Symbolic Model Checking over a Video on Demand System 28

3.1 The ALMADEM-VOD Server 29

3.2 Modeling the ALMADEM-VOD Server in Verus 34

3.3 Results 36

3.4 Conclusion 39

4 *Scheduling* 41

4.1 Basic Concepts 42

4.1.1 Workshops 44

4.2 Traditional Approaches to Job-Shop Problem 46

4.3 Conclusion 48

5 Symbolic Model Checking Applied to Deterministic Scheduling Problems 50

5.1 Modeling JSP into OBDD Domain 50

5.2 Modeling Job-Shop Problems into SMV 53

5.3 Computing Makespan 58

5.4 Conclusion 61

6 New Model Checking Algorithms 63

6.1 MINCOND Algorithm 64

6.1.1 MINCOND Application 66

6.2 MINCOND-P Algorithm 68
6.2.1 MINCOND-P application 70
6.3 MINCOUNT-P Algorithm 71
6.3.1 MINCOUNT-P application 73
6.4 Conclusion 74

7 Integration between Production and Operational Planning 75

7.1 Integrating Information between Tactical and Operational Levels 76
7.2 Multi-Period Scheduling Problem 77
7.3 Modeling Multi-Period Job-Shop into SMV 80
7.4 Conclusion 85

8 Conclusion and Future Works 86

8.1 Future Works 88

A Instances of JSP and FSP 90

B Mixed Integer and Linear Programming Model for JSP 94

C Automatic Generator for SMV Programs 97

List of Tables

5.1 Description of an hypothetical manufacturing industry problem. It presents the demanded time and execution ordering to produce automobile's front panels. 53

5.2 Consumption of OBDD nodes, time and memory by SMV when executing some flow-shop and job-shop problems. The column "time" is in seconds, except in the 6×6 job-shop cell, that is in minutes. 59

5.3 Time comparison (in seconds) between SMV and CPLEX in computing the makespan of some JSPs and FSPs. 60

List of Figures

2.1 A program example and its correspondent Kripke structure. 11

2.2 **Linear** and **branching-time** structure of time of temporal logics. 12

2.3 A Kripke structure and a relating computation tree. Kripke state labeled to "**a b**" was taken as root and the Kripke structure was unwinded from this state. The unwinding process generated an infinite tree over which CTL is applied to. 13

2.4 Basic CTL operators over a computation tree. The black states represent the states in which proposition g holds. The "s" designates the state taken as root. This figure presents the configuration of computation trees so that the respective CTL operator can yields to true. 15

2.5 A Kripke structure representing a microwave. 16

2.6 Binary decision tree (on top) and a correspondent OBDD (below) for the boolean formula $(a \wedge b) \vee (c \wedge d)$. The dashed arc is taken when $var(v) = 0$. The solid arc is taken when $var(v) = 1$. 20

2.7 Example of a transition and its symbolic representation. 21

2.8 The MIN algorithm determines the minimum path between state $s \in S$ and state $f \in F$, in a breadth-first search way. 22

2.9 A SMV model for the microwave example presented in Section 2.1.3. . . . 23

2.10 A Verus program that models a non-deterministic event and its respective alarm. 26

3.1 Overall architecture for a video service. 28

3.2 Software architecture of the video server. 30

3.3 Service cycle with a duration of T seconds. t_s accounts both for seek and rotational delay times. 31

3.4 Service cycle occupation in the ALMADEM-VOD server for periods varying from 1s to 9s. The number of clients in the system is 20. 33

3.5 An illustration of the ALMADEM-VOD server specification in Verus. . . . 35

3.6 Variation of the service time occupation by clients for the server and for Verus ($T = 4s$). 36

3.7 Service time for the ALMADEM-VOD server and for the revised Verus model ($T = 4s$). 37

3.8 Synchronization of disk and network threads through a pair of *signal* and *wait* primitives. 39

3.9 Service time for the new versions of the server and of the Verus model ($T = 4s$). 40

4.1 A taxonomy for scheduling problems. The shaded rectangle represents the class of deterministic static scheduling problems that is the focus of this thesis. 43

4.2 The structure of flow-shop. 44

4.3 The structure of an open-shop. 45

4.4 Disjunctive graph representing a job-shop 3×3 (3 jobs in 3 machines) . . . 47

5.1 An OBDD representing the ordering for the conclusion of task 7. The boolean variables c_x and c_y represent hypothetical machines x and y, respectively. In this illustration the machines are represented by a 3-bit counters. Machine x is the one that runs task 7, and machine y runs task 6. Task 7 can be run only if task 6, technological precedence of 7, is finished. Task 6 is finished when all boolean variables representing machine y is zero. Task 7 is finished when can reach the terminal node 1, and there is only one path to it. 52

5.2 An excerpt of the module Main of the SMV program relating to the problem presented in Table 5.1. 54

5.3 Excerpt of our SMV program relating to a task of a JSP. 56

5.4 Excerpt of our SMV program relating to a machine that chooses one task among two of them to run. 57

6.1 The MINCOND algorithm and an illustration of its behavior. 64

6.2 Gantt chart for the 3×3 job-shop relating an hypothetical manufacturing industry of automobile front panels described in Table 5.1. This illustration presents a sequence of tasks that results in a minimum makespan. 66

6.3 Gantt chart for the 3×3 job-shop relating an hypothetical manufacturing industry of automobile front panels. This illustration presents a sequence of tasks in which Panel A finishes as soon as possible but minimizes the late of the other tasks. 67

6.4 The MINCOND-P algorithm and an illustration of its behavior. This algorithm searches for paths between states I and F, in which all states along the path satisfy condition C. If MINCOND-P finds such paths, it returns the size of the shortest one. Otherwise MINCOND-P returns *infinity*. The illustration presents a case in which F is reached in 4 steps. 69

6.5 Gantt chart for the 3×3 job-shop problem example, considering that machines 1 and 3 can not run in parallel anymore. 71

6.6 MINCOUNT-P algorithm returns a path between state I and F, such that in this path m states satisfies C. m is the least number of states satisfying C over all paths leading from I to F. If there is no path between I and F, MINCOUNT-P returns the special value NOPATH. If $m = 0$, then MINCOUNT-P returns ∞ (infinity). The illustration shows the behavior of the algorithm. It walks backward reaching states over paths between I and F. In this illustration, we can see that I is reached in 5 steps. 72

7.1 A 3×3 JSP represented as disjunctive graph (on top) and an optimal sequence for the corresponding makespan represented by a Gantt chart (in the bottom). 78

7.2 Model for exchange of information between tactical and operational level. . 79

7.3 The representation of a multi-period JSP 3×3. 80

7.4 A simplified excerpt of the main module of our SMV model for a multi-period JSP. To control the multi-periodicity, we have added variables such as NP and t_y.p where y={1,3,4,6}. Variable t_1.p, for example, indicates in which period t_1 has been done. 81

7.5 Excerpt of our SMV model for a task of a multi-period JSP. 83

7.6 Excerpt of our SMV model for a machine that runs two tasks in an multi-period JSP. 84

7.7 Makespan for a JSP 3×3 of 3 periods. 85

B.1 MILP model for JSP 3×2. 95

B.2 The data for the MILP model for JSP 3×2 96

Chapter 1

Introduction

1.1 Motivation

Scheduling relates to ordering a set of activities to be performed by a set of resources over a period of time. Scheduling problems arise when the necessary resources for performing the activities are scarce. Due to this scarcity it becomes essential to determine the best order of execution for all activities. Determining the best order implies in scanning a space that have a finite or countably infinite number of possible solutions. This class of problem is known to be NP-Hard. Therefore, depending on the size of the instance problem, it is necessary the use of appropriate tools to get a feasible solution.

Scheduling problems range over a great variety of domains. We can find scheduling problems in areas, such as manufacturing, publishing, transport, health, computing, etc. For this reason, scheduling problems have different classes of problems. We are concerned with a class in which all parameters about resources and activities are stated in advance. This class of problems is known as deterministic static scheduling problems. For now on, we refer to activities as *jobs* and resources as *machines*.

The main objective of this work is to explore the application of formal methods to the solution of scheduling problems. Specifically we are applying a formal method based in a temporal logic model checking technique. This technique is known as Symbolic Model Checking (SMC). SMC models a problem as finite state transition graph. Efficient algorithms traverses the state space searching for model properties. The properties are specified using temporal operators or quantitative algorithms.

The application of SMC in scheduling problems presents a new perspective to solving

this class of problems. SMC is a general problem solver in opposite to certain traditional tools. Being so, SMC provides great flexibility to solve a wide range of scheduling. Besides, the efficiency of its algorithms and its internal representation allows the modeling of complex instance problems.

1.2 Traditional Approaches

Researchers have studied many different scheduling problems. There are problems involving one machine, several machines, related and unrelated machines. Besides, there are also different job characteristics and optimality criteria to consider. In common with all of them is that the great majority of this problems are NP-Hard. This implies that it seems unlikely that there is a polynomial time algorithm relating the size of the input problem that gives an exact solution to them. Therefore there are different approaches to cope with them.

Some approaches consist of *relaxing* the restrictions of the problem. The relaxation makes the problem simpler and consequently easier to solve. For example, the problem of non preemptive scheduling of independent jobs on two identical processors, and minimum makespan as optimization criterion is NP-hard; but if we relax this problem by allowing preemption it becomes an $O(n)$ complexity time problem [12], where n is the number of jobs. In general, the relaxation is used to help in the design of approximation algorithms.

Among approximation algorithms, there are also different approaches to cope with scheduling problems. There are *constructive methods* in which one important representative is the *shifting bottleneck heuristic* [2, 8]. There are also *iterative methods* in which we can find the *local search* algorithms that comprises *tabu search*, *simulated annealing* [4, 9], and *genetic algorithms* [54]. All these algorithms are very efficient and in some cases are used to determine bounds to certain exact approaches.

Exact approaches to scheduling problems use *enumerative methods* such as branch-and-bound (B&B) algorithms [5, 15, 16, 24, 25, 65]. B&B algorithms perform an (implicit) enumeration of all feasible solutions (schedules) and have been very successful in solving important scheduling problems. However, to get efficiency, some implementations are designed taking into considerations so many intrinsics characteristics of the instance problem that is not guaranteed that the same efficiency will occur in a different instance of the same problem. This means that some implementations work fine just with that specific instance. Any change in characteristic of the problem may incur some kind of algorithm

modification and re-implementation.

1.3 Symbolic Model Checking

Symbolic Model Checking (SMC) [66] is an automatic formal technique proposed for verifying finite states concurrent systems. SMC models a system being verified as a state-transition graph. Properties about the model are specified in a temporal logic. The model checking process consists of exploring all states of the model to determine if the specified properties are satisfied. This verification always terminates and the truth value of the properties are presented. When any property is falsified, SMC is able to give a *counter-example*: a path in the model that demonstrates the property falsification. Counter-example is a trace of states that presents a possible computation in which the model does not satisfy the property. It is a very important tool for debugging systems. It generally provides very good insights about the erroneous behavior of the system being modeled.

SMC has been successful in verifying several large and complex systems such as the Futurebus+ protocol [30] and the PCI bus performance [18]. States and transition between states of the model are represented symbolically by boolean formulas and implemented by OBDDs [17]. The use of OBDD allows SMC to verify systems with as many as 10^{30} states (in some specific cases it was already reached 10^{120} states [31]).

1.4 The Proposed Approach and Results

Symbolic Model Checking (SMC) is a temporal formal method technique appropriate to finite states systems. Since solving scheduling problems consists of identifying a feasible solution over all possible solutions, SMC is very appropriate to this class of problem. The underlying model of SMC and its basic model checking algorithms can solve many class of scheduling problems, such job-shop problem (JSP). In this case, SMC is able to compute the *minimum makespan*, that corresponds to a sequence of jobs over machines such that this sequence imposes the least minimum time to accomplish all jobs. In SMC, computing the minimum makespan of a JSP, for example, reduces to the reachability problem between two states of the model.

SMC is a general problem solver. Like CPLEX[56], SMC is able to deal with some non predictable questions. In a manufacturing plant, for example, the production manager could face a situation where he/she needs to recalculate its production line schedule to

adequate to some new circumstances. For example:

1. one needs to determine a new minimum makespan since a specific job must finish before all others. It is not the case of simply computing the new makespan not considering this specific job. Again, SMC answers this question by evoking an appropriate quantitative algorithm combined with an appropriate temporal logic property.

2. one needs to determine a new makespan, since, for some reason, he/she needs now to serialize two specific machines. We will see that we can deal with this restriction only evoking a quantitative algorithm with an appropriate CTL property;

Our study about SMC has begun in the context of a Video on Demand (VOD) System. At first, we have applied SMC to determine the nominal capacity of a multimedia server [11, 21]. We have analyzed the ALMADEM-VOD server a component of a VOD system. We have determined the performance bounds to the server, and these bounds have pointed out to a discrepancy with the actual server. The discrepancy was analyzed and an error was found in the code of the server. After correcting this error, ALMADEM-VOD improved its performance (number of clients being served at the same time) in 40%.

After VOD work, we have started the study about scheduling which involved the application of SMC in solving some class of scheduling problems known as flow-shop and job-shop problems. We have modeled job-shop problems into SMC and computed the minimum makespan of some non-trivial instances [63]. We have determined, for example, the minimum makespan of a job-shop 6×6 (six jobs in six machines), whose solution spaces is around $(6!)^6$ possible sequences.

SMC can be seen as a finite state problem solver tool box. When the current symbolic algorithms are not well appropriate to deal with a specific problem, we can implement new algorithms and incorporate them into the tool box. It is important to mention that no modification is necessary to the original verification algorithms already installed. Indeed, in this work, we present three new algorithms that we have implemented to cope with scheduling problems: MINCOND, MINCOND-P, and MINCOUNT-P.

MINCOND returns the size of the shortest path (i.e. number of edges) between two specified states s_I and s_F, such that a specified condition C holds somewhere along this path. The number of times C holds in the path does not matter. This algorithm is very useful to answer for example "what is the makespan since we want a job j terminates before a job k?" That is *job j terminates before a job k* corresponds to the condition we want to be satisfied. MINCOND is able to find a minimum path between s_I and s_F such

that along this minimum path the condition is satisfied at least once. That is the case of example 1 above.

MINCOND-P is slightly different from MINCOND. MINCOND-P returns the size of the shortest path (i.e. number of edges) between s_I and s_F, such that condition C holds in ALL states of the path that precedes s_F. MINCOND-P starts from s_I searching for paths that lead to s_F. However, only states that satisfy C are considered. This algorithm is very useful to deal with problems like "what is the new makespan since machine 1 and machine 3 can not run in parallel anymore?" This is the case described in example 2 above.

MINCOUNT-P is an extension to the algorithm MINCOUNT [22]. Consider $\pi_1, \pi_2, \ldots, \pi_p$ being all paths between states s_I and s_F. MINCOUNT searches for states that satisfies a condition C along the paths π_i, $1 < i < p$ and returns $m = min(m_1, m_2, \ldots, m_p)$ such that m_i is the number of states satisfying C along the path π_i. MINCOUNT-P extends MINCOUNT by presenting the path relating to m. This algorithm is very useful when we are facing problems like the following. In many manufacturing industries, the jobs can be processed by anyone of the machines of the production line. If we want to determine a sequence for the jobs over the machines, such that the makespan is minimum and some specific machine is used as less as possible, then the MINCOUNT-P does this task.

Several examples have been applied to MINCOND and MINCOND-P. The results have confirmed the viability of the approach. SMC has correctly computed the minimum makespan of the problems, and SMC indeed has served as tool box.

Finally, we have also applied SMC to solving the problem of integration between production and scheduling planning. Classical models for production (tactical) and scheduling (operational) planning is concerned with different types of information. The former works with more general information such as quantity of products to be produced over periods of time (weeks or months). The latter works with more detailed information, such as jobs, machines, running time of jobs over machines. Tactical managers need to know whether their production planning is feasible at the operational level. In other words, if there exist a scheduling of production (operational planning) such that the production planning is feasible. Due to the lack of operational informations at tactical level, that is not always the case. We have presented a SMV model to solve this problem. We have also presented the result we have obtained solving an hypothetical integration problem.

1.5 Contributions

In this work we have proposed the application of SMC (a formal method technique) to give exact solution to scheduling problems. The model checking approach assures the correct answer to different questions that can occur in a plant, for example. These questions can appear in unexpected fashion and in general can be answered by SMC only by using appropriate algorithms and/or specifying a correspondent CTL formula.

The algorithms we have proposed enhance the capability of SMC as finite state general problem solver. It implements some features that are not obtained directly by original CTL formulations.

We also have presented new algorithms that enhance the verification power of SMC. Besides we have presented SMC models for job-shop, flow-shop and integration decision planning problems.

The SMC approach to scheduling problems also opens up new possibilities to two independent research groups: Combinatorial Optimization and Formal Methods. To the former, it offers a new view of how to solve this kind of problems. Since SMC makes available temporal logic feature, it can be used to determine if is possible to get some job finished before a given deadline, for example. To the latter group, it offers a great amount of research about how to deal with big state space.

1.6 Organization of the Thesis

The next chapter presents an introduction to Formal Methods and describes Model Checking technique. This description includes Kripke structures, temporal logic, and OBDD. It also includes SMC tools such as SMV and Verus. Chapter 3 presents the result we have obtained with the formal verification of a VOD server. Chapter 4 describes scheduling problems and declares the class of scheduling problems we are working in this thesis. Chapter 5 presents how we can model scheduling problems into SMC. We also present example of a model and the results we have obtained solving some instance problems. Chapter 6 presents the algorithms we have designed to enhance the verification power of SMC that are used to solve scheduling problems. At last, we have Chapter 7 that describes how SMC can deal with the problem of integrating different levels of decision. Finally, we present the conclusion about this work and the future directions of this work.

Chapter 2

Symbolic Model Checking - SMC

Formal Methods (FM) are mathematical based techniques and tools for specifying and verifying systems. Specification techniques are used to formalize the requisites and properties of a system and its product usually can be converted in system documentation. Some examples of specification techniques/tools are Z [77], VDM [58], CSP [53]. Verification techniques and tools go one step beyond. They are able to help the system designer to find errors in the system. The system is modeled in a suitable language and properties about the system can be verified. Some examples are SMV [66], CV [39]. Other examples of specification and verification techniques/tools as well as real case applications of FM can be found in [81, 32].

Many verification techniques and tools in general are very known by Theorem Provers and Model Checkers. In both of them the verification process comprises three phases [66, 55]:

Modeling - description of characteristics and behaviors of the system in a given language;

Specification - description of the system's properties that we want verify; and

Verification - execution of the verification process in order to determine if the properties hold for the model.

In the Theorem Prover (TP) approach, the system is modeled as a set of formulas Φ (of an appropriate mathematical logic). The specification is also described as a formula ϕ and the verification is the process of finding a proof for ϕ such that $\Phi \vdash \phi$. The proof usually is a manual or interactive process.

In the Model Checking (MC) approach, the system is modeled for an appropriate logic. The model ($\mathcal{M}$) is finite and the specification is described as formula ϕ. The MC process consists of verifying whether $\mathcal{M}$ satisfies ϕ by scanning all reachable states of $\mathcal{M}$ from a state s of this model. This process can be represented mathematically by $\mathcal{M}, s \models \phi$.

The MC approach is more restrictive than TP approach in the following aspects: (1) it deals with finite systems; (2) it proves the satisfiability of ϕ with respect to $\mathcal{M}$, not to all models $\mathcal{M}$, such that $\mathcal{M} \models \phi$. These aspects make TP a more general technique than MC approach. However, they also confer important features to MC that usually are not present in TP. In general, the MC verification process is fast and fully automatic. Besides, MC is able to present a **counterexample**, when ϕ is falsified. Counterexample is a computation sequence in the model that proves that $\mathcal{M} \not\models \phi$.

This chapter presents Symbolic Model Checking (SMC), a representative of MC approach, that is the tool used in this thesis to exactly solve scheduling problems. Initially, we make a review about MC formalism concerning **Kripke structure**, the underlying model of MC, and **Computation Tree Logic**, a temporal logic that is used to reason about the modeling systems. We also present how a simple microwave is modeled by a Kripke structure and how CTL can reason about this model, concerning a microwave property. After that, we make a review about SMC focusing its **symbolic model** and **OBDD**, the data structure used to implement this model. We also present a basic SMC search algorithm used to find the minimum path between two states of the model. This algorithm is used in this thesis to solve some scheduling problems and has inspired some new algorithms presented in Chapter 6. Finally, we present two SMC tools that we have used along this thesis: SMV and Verus.

2.1 Model Checking - MC

MC is a technique originally proposed to deal with finite states systems [1]. The system being verified is represented by an appropriate model and the properties we want to verify are specified in a correspondent reasoning system. The MC process consists in scanning (all) states of the model to check if the model conforms the properties. Although MC approach can only deal with finite states systems, there are important systems that fall into this class, such as digital circuit design and communication protocols.

[1] Nowadays, there are some extensions that allow Model Checking to be used with infinite systems, but these techniques are out of scope of this work.

MC technique has some variants. *Temporal logic* model checking [29, 74] represent the system as finite state transition graph and a temporal logic [64] is used to specify properties about the system. Efficient algorithms search the state space to check if the model satisfies the properties. In *automata* approach, the system and the properties are represented as automata. Then, the system is compared to the properties to determine if they hold to the system. This comparison is accomplished by techniques such as language inclusion, refinement orderings, and observational equivalence [32]. Another approach is model checking by using *integer linear programming* (ILP) [33, 28]. In this approach, the system and the properties about the system are modeled as a linear inequality system. The equations relating to the the system properties are expressed in a such way that model the properties that the system should not have. The inequality system then is applied to an ILP method. If an integral solution is found then the system has the necessary conditions for the violation of the system properties.

The MC approach we will strengthen here is the temporal logic model checking. Finite states transition systems are modeled as Kripke structure and a temporal logic is used to specify properties.

2.1.1 Kripke Structure

Kripke structure [31] is a well known model for finite states transition systems. Formally, let AP be a set of atomic proposition. A Kripke structure K over AP is a 4-tuple (S, S_0, ρ, L), where S is a finite set of states, S_0 is the set of initial states, $\rho \subseteq S \times S$ is a transition relation (that is total - that is, all states have successors), and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of propositions true in that state.

A state in a Kripke structure represents a snapshot of the system being modeled. The system is represented by a set of variables and each state represents the valuation of these variables in a specific moment. Considering D a finite set and $V = \{v_1, v_2, \ldots, v_n\}$ the variables of the system, the valuation for $s \in S$ is the result of the function $s : V \rightarrow D$, that associates a value in D for each variable in V.

The Kripke structure use formulas of propositional logic to represent states. Let $D = \{FALSE, TRUE\}$ and $V = \{v_1, v_2, v_3\}$, the valuation ($v_1 \leftarrow TRUE,\ v_2 \leftarrow FALSE,\ v_3 \leftarrow TRUE$) can be expressed by the following formula of the propositional logic ($v_1 \wedge \neg v_2 \wedge v_3$). As a formula can be true in many states, it can be used to represent the set of states in which the valuation is true.

The propositional logic is also used to represent the transition between states. To this extent another set of variables V' is created, such that for each $v \in V$ there is a corresponding variable $v' \in V'$. Variables in V are used to represent the current state and V' to represent the next state. A valuation to V and V' corresponds to setting an ordered pair of states, and can be expressed by formulas as before. For instance, consider that the current states are represented by the formula $(v_1 \ \wedge \ \neg v_2 \ \wedge \ v_3)$, if in the next state $v_1 \leftarrow FALSE$, then the transition between the current to the next state can be expressed by the formula $(v_1 \ \wedge \ \neg v_2 \ \wedge \ v_3 \ \wedge \ \neg v'_1 \ \wedge \ \neg v'_2 \ \wedge \ v'_3)$. The set of transitions of a system is referred to as *transition relation*, and if ρ represents a transition relation, then $\rho(V, V')$ denotes a formula that expresses it.

Finally, let us consider the set of atomic propositions AP. Each atomic proposition has the form $v = d$, where $v \in V$ and $d \in D$. An atomic proposition $v = d$ is true in a state s, if $s(v) = d$. When v ranges over boolean domain {TRUE, FALSE}, we write v to mean $s(v) = TRUE$ and $\neg v$ to mean $s(v) = FALSE$.

We now show how Kripke structure $K = (S, S_0, \rho, L)$ models a system from propositional formula $\mathcal{S}_\mathcal{I}$ and from the system $\mathcal{R}$.

- S is the set of states consisting of all possible valuations for V;
- S_0, initial states, is the the set of $s \in S$ such that satisfies the formula $\mathcal{S}_\mathcal{I}$;
- $\rho \subseteq S \times S$ is the transition relation. $\rho(s, s')$ holds if $\mathcal{R}$ yields to true when each $v \in V$ and each $v' \in V'$ is assigned to $s(v)$ and $s(v')$, respectively;
- $L : S \rightarrow 2^{AP}$ is the function that labels the states. $L(s) \subseteq AP$ is the set of $p \in AP$, such that p is true.

A computation in the system is represented by a path in K. A valid path from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ \ldots$, such that $s_0 = s$ and $\rho(s_i, s_{i+1})$ holds $\forall i \geq 0$. As an example, consider the program of Figure 2.1-a. According to this program $V = \{x, y\}$, $D = \{0, 1\}$. The set of initial states is represented by $\mathcal{S}_\mathcal{I}(x, y) \equiv (x = 0) \wedge (y = 1)$ and the transition relation by $\rho(x, y, x', y') \equiv (x' = y) \wedge (y' = y)$. The corresponding Kripke structure $K = (S, S_0, \rho, L)$ to these formulas is the following:

- $S = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$
- $S_0 = \{(0, 1)\}$
- $\rho = \{((0, 0), (0, 0)), \ ((0, 1), (1, 1)), \ ((1, 0), (0, 0)), \ ((1, 1), (1, 1))\}$.

- $L((0,0)) = \{x = 0, y = 0\}$, $L((0,1)) = \{x = 0, y = 1\}$, $L((1,0)) = \{x = 1, y = 0\}$, $L((1,1)) = \{x = 1, y = 1\}$

The Figure 2.1-b illustrates K. The states are numbered only to ease the explanation about this figure. Observe that starting from S_0 states 1 and 2 are not **reachable**.The only valid computation in this system is that related to the path 0,3,3,3,3,3,...

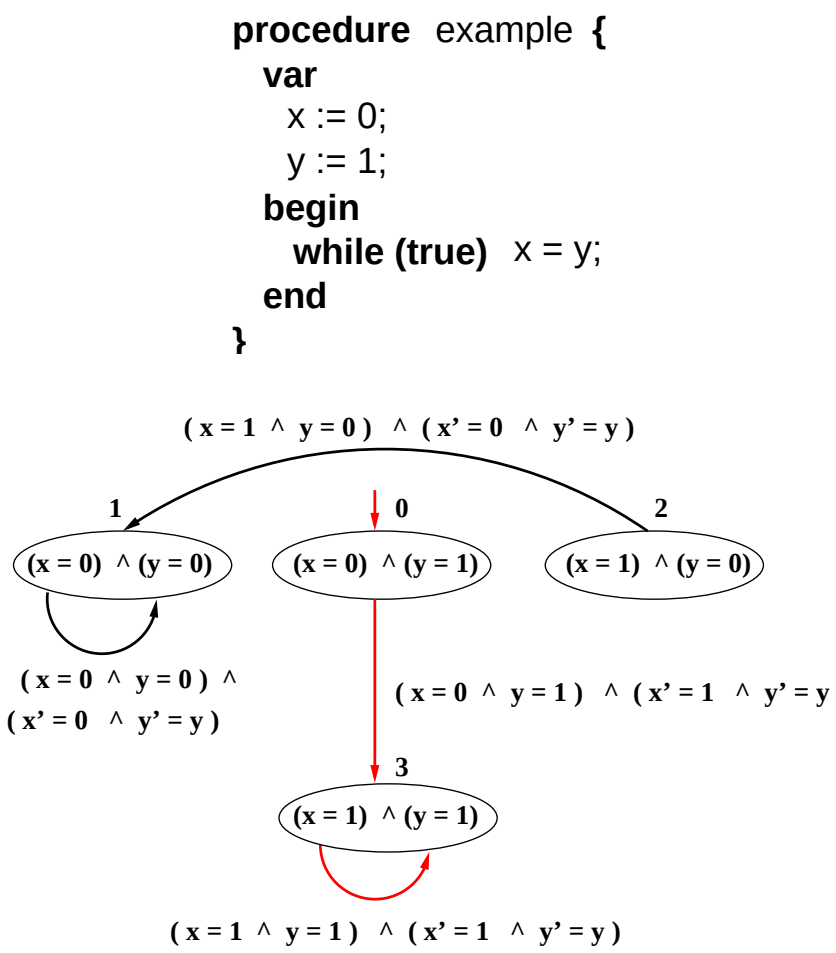


Figure 2.1: A program example and its correspondent Kripke structure.

Definition 2.1 *A state s_n is reachable from a state s_0, if there is a path $\pi = s_0 s_1 s_2 \ldots s_n$ such that $\rho(s_{i-1}, s_i)$ holds for $0 \leq i \leq n$.*

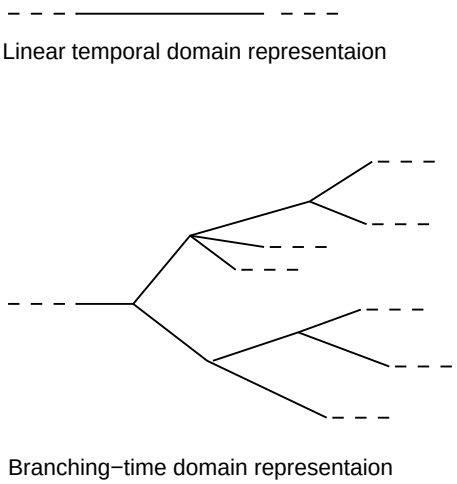


Figure 2.2: **Linear** and **branching-time** structure of time of temporal logics.

.

2.1.2 CTL - Computation Tree Logic

Temporal logic is a formalism very useful to describe sequences of transitions between states [66]. With temporal logic we are able to reason about the system in terms of occurrences of events. For example, given a model K the temporal logic offers reasoning power to determine if a certain event will *eventually* occur or if it *always* occurs.

There exists several types of temporal logic [72]. These logics vary according temporal structure (*linear* or *branching-time*) and time characteristic (*continuous* or *discrete*). **Linear temporal logics** reason about the time as a chain of time instances. **Branching-time logics** reason about the time as having many possible futures from a given instance of time. (See figure 2.2[2].) **Time is continuous** if between two instances of time is always possible to find another one. **Time is discrete** if between two instances of time is not always possible to find another one.

The logic of our study is a branching-time and discrete one known as Computation Tree Logic (CTL) [29]. Its name is due to the reasoning over *trees of computation*. The tree is (metaphorically) obtained unwinding the Kripke structure from a determined state (taken as root). Figure 2.3 presents an example.

CTL provides operators to be applied over the paths formed by the computation tree. When these operators are specified in a formula they must appear in a pair and in this order: *path quantifier* followed by *temporal operator*. A path quantifier defines the scope of

[2]This figure was taken from [72].

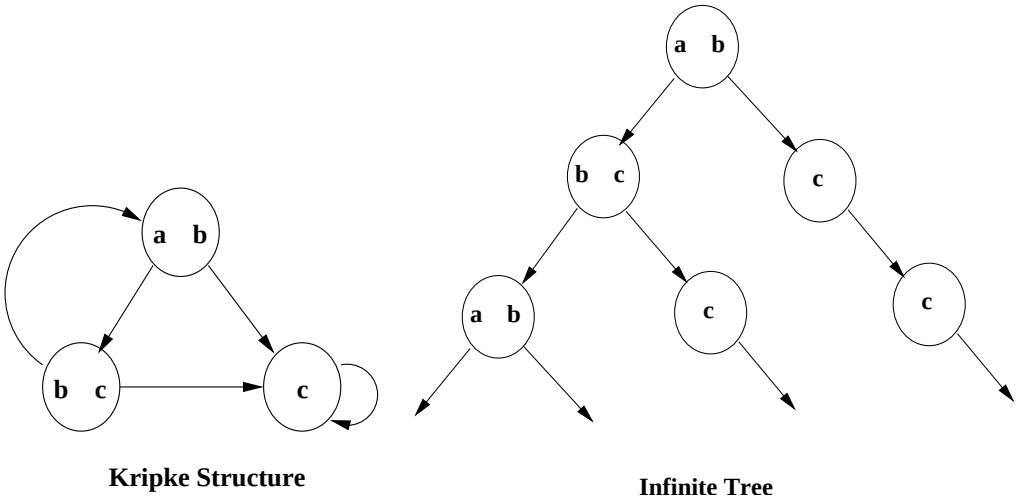


Figure 2.3: A Kripke structure and a relating computation tree. Kripke state labeled to "**a b**" was taken as root and the Kripke structure was unwinded from this state. The unwinding process generated an infinite tree over which CTL is applied to.

the paths over which a formula f must hold. There are two path quantifiers: **A**, meaning **all** paths; and **E**, meaning **some** path. A temporal operator defines the appropriate temporal behavior that is supposed to happen along a path relating a formula f. The temporal operators are the following:

- **F** ("in the future" or "eventually") - starting from the root, f holds in some state of the path;
- **G** ("globally" or "always") - starting from the root, f holds in all states of the path;
- **R** ("release") - there is a state s in the path where formulas f and g hold and all preceding states from s does not satisfies f;
- **U** ("until") - there is a state s in the path where a formula g is satisfied and all predecessor states of s satisfies f.
- **X** ("next time") - starting from the root, f holds in the second state of the path.

A well formed CTL formula is defined as follows:

1. $TRUE$, $FALSE$ are CTL formulas;
2. If $p \in AP$, then p is a CTL formula, such that AP is the set of atomic propositions;
3. If f and g are CTL formulas, then $\neg f$, $f \vee g$, $f \wedge g$, AFf, EFf, AGf, EGf, A[fRg], E[fRg], A[fUg], E[fUg], AXf, EXf, are CTL formulas.

Considering the Kripke model $M = (S_0, S, \rho, L)$, we denote that M satisfies a CTL formula f from a state $s \in S$ as

$$M, s \models f$$

Let f and g be CTL formulas, the satisfaction relation $\models$ is defined inductively as follows [55]:

$M, s \models TRUE$ and $M, s \models FALSE$ for all $s \in S$
$M, s \models p \quad \Leftrightarrow p \in L(s)$
$M, s \models \neg f \quad \Leftrightarrow M, s \not\models f$
$M, s \models f \vee g \quad \Leftrightarrow M, s \models f$ or $M, s \models g$
$M, s \models f \wedge g \quad \Leftrightarrow M, s \models f$ and $M, s \models g$
$M, s \models AFf \quad \Leftrightarrow$ for all paths from $s, s_k \in S$ is reachable and $s_k \models f$
$M, s \models EFf \quad \Leftrightarrow$ for some path from $s, s_k \in S$ is reachable and $s_k \models f$
$M, s \models AGf \quad \Leftrightarrow$ for all paths $\pi = s_0 s_1 s_2 \ldots, s_i \models f$, for all $i \geq 0$, and $s_0 = s$
$M, s \models EGf \quad \Leftrightarrow$ for some path $\pi = s_0 s_1 s_2 \ldots, s_i \models f$, for all $i \geq 0$, and $s_0 = s$
$M, s \models AXf \quad \Leftrightarrow$ for all s_x such that $\rho(s, s_k)$ is defined, $s_k \models f$
$M, s \models A[fUg] \Leftrightarrow$ for all paths $\pi = s_0 s_1 s_2 \ldots s_k \ldots, s_i \models f, 0 \leq i < k$ and $s_k \models g$
$M, s \models E[fUg] \Leftrightarrow$ for some path $\pi = s_0 s_1 s_2 \ldots s_k \ldots, s_i \models f, 0 \leq i < k$ and $s_k \models g$
$M, s \models A[fRg] \Leftrightarrow$ for all paths $\pi = s_0 s_1 s_2 \ldots s_k \ldots, s_i \not\models f, 0 \leq i < k$ and $s_k \models g$
$M, s \models E[fRg] \Leftrightarrow$ for some path $\pi = s_0 s_1 s_2 \ldots s_k \ldots, s_i \not\models f, 0 \leq i < k$ and $s_k \models g$

Despite all combinations we can get with path quantifiers and temporal operators presented above, we can express any CTL formula using $\vee$, $\neg$, **EX**, **EU**, **EG** [31]:

- AF $f = \neg$ EG $\neg f$
- AG $f = \neg$ EF $\neg f$
- AX $f = \neg$ EX $\neg f$
- A[f U g] $\equiv \neg$ E[$\neg g$ U $(\neg f \wedge \neg g)$] $\wedge \neg$ EG $\neg g$
- A[f R g] $\equiv \neg$ E[$\neg f$ U $\neg g$]
- EF $f =$ E[$\top$ U f]
- E[f R g] $\equiv \neg$ A[$\neg f$ U $\neg g$]

Figure 2.4 presents the computation of the most frequently used CTL operators. Some typical examples of CTL formulas relating to concurrent reactive systems are presented below.

EF$(started \wedge \neg ready)$ - it is possible to get to a state where *started* holds but *ready* does not hold.

AG$(req \rightarrow$ **AF**$ack)$ - it is always the case that if the signal *req* is high, then eventually *ack* will also be high.

A$[greenLight$ **U** $armMoves\ ]$ - it is always the case that the robot's arm moves after the green light is on;

E$[greenLight$ **R** $armMoves\ ]$ - it does exist a situation in which the robot's arm moves before green light is on;

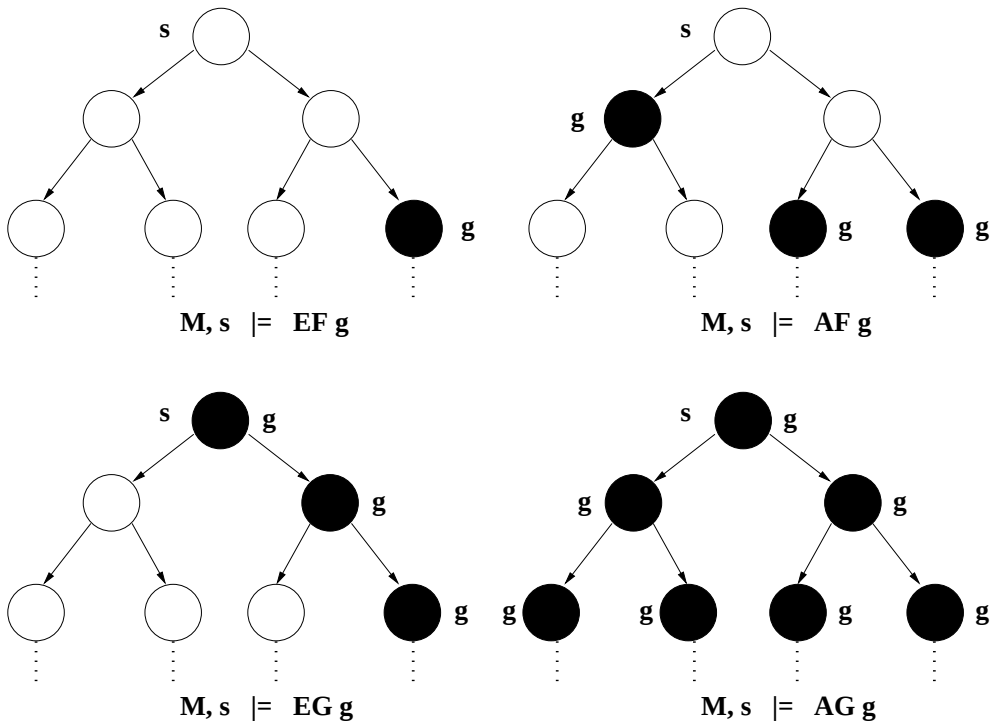


Figure 2.4: Basic CTL operators over a computation tree. The black states represent the states in which proposition g holds. The "s" designates the state taken as root. This figure presents the configuration of computation trees so that the respective CTL operator can yields to true.

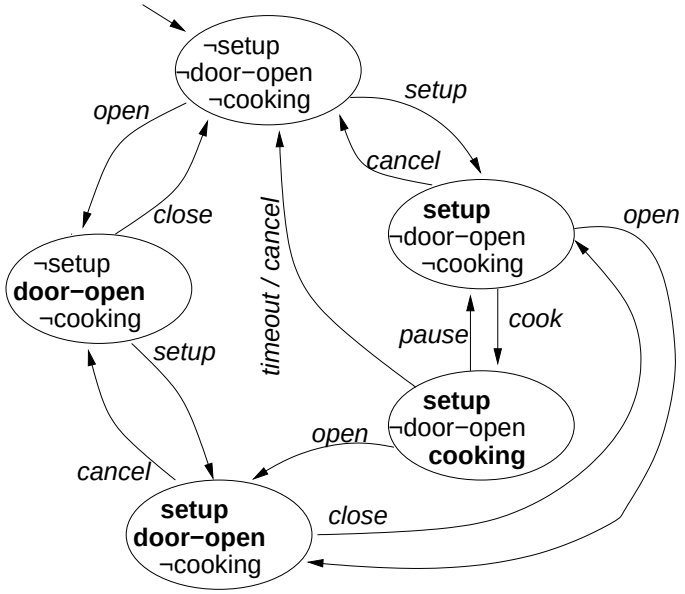


Figure 2.5: A Kripke structure representing a microwave.

2.1.3 A Microwave Example

We present an example of verification using CTL logic. The example is about a simplified microwave, inspired in a similar example given by Clarke, Grumberg, and Peled [31]. The possible operations in our microwave are the following: (a) we can open or close the door of the microwave; (b) we can set the time of cooking; (c) we can cook; (d) we can pause or cancel the cooking; (e) we can restart the cooking; and (f) the microwave turns itself off after the time of cooking is over - timeout. This microwave can be modeled by three boolean variables:

setup represents the parameters of the microwave set by user, such as time of cooking;

door-open indicates whether the door of the microwave is opened or not;

cooking indicates if the microwave is cooking the meal.

Figure 2.5 presents a Kripke structure related to our microwave example. The transition between states are labeled to the events that cause the transition to be taken. These labels are merely illustrative. The initial state is indicated by the incoming edge without source. By definition, states of Kripke structure are label to the variables that are true in that state. To easy the understanding of the model, we labeled their states using all variables according to their respective value in each state. Variables in bold face have true value. Variables preceded by negation symbol (¬) have false value.

A very important (life) safety property concerns with cooking with door opened. We can verify this property with CTL logic by expressing

$$\mathrm{AG}(\mathit{cooking} \rightarrow \neg \mathit{doorOpened})$$

We can simplify this formula in terms of equivalent ones:

$$\begin{aligned}
&\mathrm{AG}(\mathsf{cooking} \rightarrow \neg\mathsf{door\text{-}open}) \equiv \\
&\neg\mathrm{EF}\neg(\mathsf{cooking} \rightarrow \neg\mathsf{door\text{-}open}) \equiv \\
&\neg\mathrm{EF}\neg(\neg\mathsf{cooking} \vee \neg\mathsf{door\text{-}open}) \equiv \\
&\neg\mathrm{EF}(\mathsf{cooking} \wedge \mathsf{door\text{-}open})
\end{aligned}$$

Therefore, we need to find out if there exists a state in the model such that the proposition $\mathsf{cooking} \wedge \mathsf{door\text{-}open}$ holds in any state. Since there is no state that satisfies this proposition, the model satisfies the property $\neg\mathrm{EF}(\mathsf{cooking} \wedge \mathsf{door\text{-}open})$.

2.1.4 CTL Model Checking

CTL model checking consists of searching for states of Kripke model with label f, where f is a CTL formula. The set of states labeled to f are the ones that satisfies the formula. Formally, let f be a CTL formula and $\mathit{label}(s)$ be the set of subformulas of f that are true in $s \in S$. The CTL model checking problem is related to determining the set $S = \{s \mid M, s \models f \rightarrow f \in \mathit{label}(s)\}$.

The model check process has two phases: translation and labelling. The translation phase consists of rewriting a CTL formula in terms of $\neg$, $\vee$, EX, EG, and EU. The labelling is a process of i steps, where i is the number of sub-formulas of f. In each ith step, the $i-1$ nested CTL operator (sub-formula) labels a state if the sub-formula is true in that state.

The labelling process observe the following rules:

- label s to p, if $p \in AP$ and $p \in L(s)$;
- label s to $\neg f1$, if s is not labeled with $f1$
- label s to $f1 \vee f2$ if s is labeled with $f1$ or with $f2$;
- label s to $\mathrm{EX}f$, if t is labeled with f and $R(s,t)$;

- label s to E[$f1$ U $f2$]

  1. if any s is labeled with $f2$, then label it with E[$f1$ U $f2$];
  2. repeat backward from s: label t to E[$f1$ U $f2$] if t is labeled with $f1$ and exists a state u labeled with E[$f1$ U $f2$], such that $R(t, u)$;

- label s to EG[f]

  1. label all states to EG[f];
  2. delete EG[f] from any state s in which if s is not labeled with f;
  3. delete EG[f] from any state s if does not exist a state t labeled with EG[f], such that $R(s, t)$.

Using a more efficient EG labelling algorithm that takes into consideration the decomposition of graph into "nontrivial strongly connected components" [31, 55], the complexity of the labelling algorithm is $O(i.(V + E))$, where i is the number of connectives $\vee$ and $\wedge$ in the CTL formula, V is the number of states, and E is the number of transitions.

2.2 Symbolic Model Checking - SMC

Although the complexity of the labelling algorithm is linear in the size of the model, the model suffers from *state explosion problem*. This problem is related to the exponential growth of the number of states of the model in the number of variables and in the number of components of the system that execute in parallel. In the early implementations of model checking, where the transition relation was represented by adjacency list, this problem implied severe limitation in the systems that could be verified.

However, using a symbolic representation of states implemented by *ordered binary decision diagram* (OBDD) [17], McMillan has proposed a new approach to model check systems known as Symbolic Model Checking (SMC). SMC [66] is a formal verification technique, that has been successful in verifying several large and complex systems such as the Futurebus+ protocol [30] and the PCI bus performance [18]. States and transition between states of the model are represented symbolically by boolean formulas and implemented by OBDDs. The use of OBDD allows SMC to verify systems with as many as 10^{30} states (in some specific cases it was already reached 10^{120} states [31]).

2.2.1 Ordered Binary Decision Diagrams - OBDDs

OBDDs [17] are the main data structure of SMC. OBDDs are an efficient way to represent boolean formulas. Often, they provide a much more concise representation than traditional representations, such as conjunctive normal forms and disjunctive normal forms. OBDDs are a canonical representation for boolean formulas. This means that two boolean formulas are logically equivalent if and only if its OBDDs are isomorphic. This property simplifies the execution of frequent operations, like checking the equivalence of two formulas or deciding if a formula is satisfiable or not.

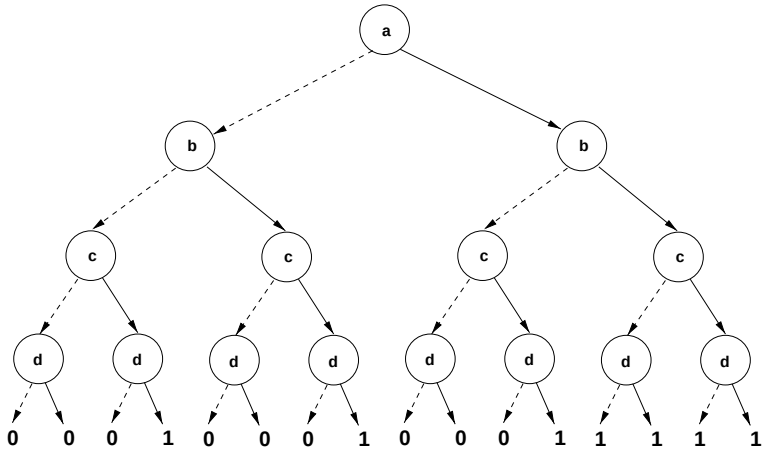
An OBDD is a directed acyclic graph with two kinds of vertex: non-terminal and terminal. Each non-terminal vertex v is labeled by $var(v)$, a distinct variable of the corresponding boolean formula. Each v has at least one incident arc (except the root vertex). Each v also has two outgoing arcs directed toward two children: $left(v)$, corresponding to the case where $var(v) = 0$, and $right(v)$, corresponding to the case where $var(v) = 1$.

An OBDD has two terminal vertices labeled by 0 and 1, representing the truth value of the formula, respectively, *false* and *true*. For every truth assignment to the boolean variables of the formula, there is a corresponding path in the OBDD from root to a terminal vertex. If the path ends in the terminal vertex labeled by 0, then the assignment does not satisfy the formula, and conversely, if the terminal vertex labeled by 1 is reached, then the formula is satisfied by the assignment. Figure 2.6 illustrates an OBDD for the boolean formula $(a \wedge b) \vee (c \wedge d)$ compared to a Binary Decision Tree for this same formula.

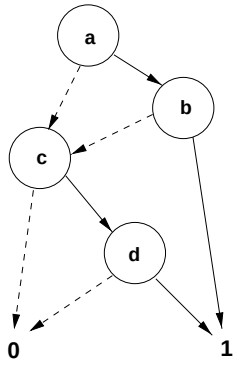
OBDD, however, has drawbacks. In our approach, the most significant is related to the the variable ordering. Given a boolean formula, the size of the corresponding OBDD is highly dependent of the variable ordering. The OBDD can grow from linear to exponential to the number of variables of the formula. In addition, the problem of choosing an variable order that minimize the OBDD size is co NP-complete [17]. Despite the existence of heuristics to automatic ordering the variables, sometimes is necessary to order them manually.

2.2.2 The Model

A system (problem) is represented as a state transition graph implemented by OBDDs. In this graph, system variables are modeled as atomic propositions represented by boolean variables. An assignment of values to all boolean variables defines a state in the graph (we assume that different states have different labels as described in [66]). For example,



Binary Tree Decision



OBDD

Figure 2.6: Binary decision tree (on top) and a correspondent OBDD (below) for the boolean formula $(a \wedge b) \vee (c \wedge d)$. The dashed arc is taken when $var(v) = 0$. The solid arc is taken when $var(v) = 1$.

if the model has three boolean propositions a, b, and c, then $(a = 1\ b = 1\ c = 1)$, $(a = 0\ b = 0\ c = 1)$, and $(a = 1\ b = 0\ c = 0)$ are possible states. The *symbolic representations* of these states are $(a\ b\ c)$, $(\overline{a}\ \overline{b}\ c)$, and $(a\ \overline{b}\ \overline{c})$, respectively, where a means that this variable is true in the state and $\overline{a}$ means that this variable is false.

Boolean formulas over variables of the model can be true or false in a given state. The value of a boolean formula in a state is obtained by assigning to the formula the values of the variables in that state. Formulas represent all states in which the formulas are true. For example, the formula $a \vee c$ is true in all the three example states discussed above. The graph representation used by our algorithms is a direct consequence of this observation. We use a boolean formula to denote the set of states in which that formula is satisfied. For example, the formula *true* represents the set of all states, the formula *false* represents the empty set with no states, and the formula $a \vee c$ represents the set of states in which a or

c are true. Because symbols are used to represent states, algorithms that use this method are called symbolic algorithms.

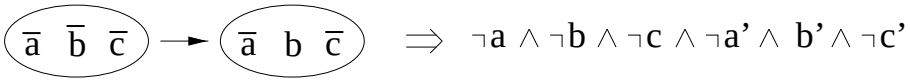


Figure 2.7: Example of a transition and its symbolic representation.

Transitions can also be represented by boolean formulas. A transition $s \rightarrow t$ is represented by using two distinct sets of variables, S for the current state s and T for the next state t. Each variable in S has exactly one correspondent variable in T. For instance, if there are variables $a, b, c \in S$, then there are variables labeled as $a', b', c' \in T$. Let f_s be the formula associated with the state s and f_t be the formula associated with the state t. Then, the transition $s \rightarrow t$ is represented by the formula $f_s \wedge f_t$. For example, a transition from the state $(\overline{a}\,\overline{b}\,\overline{c})$ to the state $(\overline{a}\,b\,\overline{c})$ is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$, as illustrated in Figure 2.7. The meaning of this formula is the following: there exists a transition from state s to state t if and only if the substitution of the variable values for s, in the current state variables, and of those of t, in the next state variables, yields *true* to the formula. The transition relation of the whole system is constructed from the disjunction of all transitions in the graph.

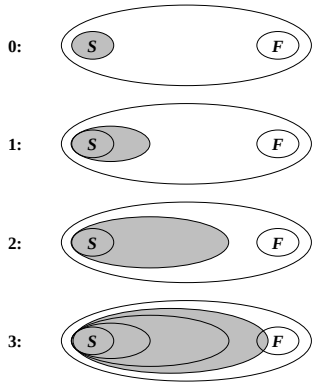
In the same way as boolean formulas can represent sets of states, they can also represent sets of transitions. Symbolic Model Checking (SMC) takes advantage of this fact by representing sets of transitions using BDDs. The BDD representation can group sets of transitions into a single formula, which often significantly reduces the size of the final representation.

2.2.3 Search Algorithms

In [17], Bryant presents efficient algorithms to execute basic operations over boolean formulas. The time complexity of these algorithms is proportional to the size of the OBDD being manipulated. To our approach, our primary interest is the MIN algorithm [22], that traverses the SMC state transition graph, in a breadth-first search way, looking for the minimum path (number of arcs) between two specific states.

The MIN algorithm takes two sets of states S (start) and F (final), and returns the shortest path (number of arcs) between $s \in S$ and $f \in F$. If there is no such path, the algorithm returns ∞ (infinity). At each step, the algorithm computes the set of states

that are reachable from S by paths of increasing length using a forwards search algorithm. Intuitively, the loop in the algorithm computes the set of states that are reachable from F. If at any point, we encounter a state satisfying F, we return the number of steps taken to reach that state. Figure 2.8 illustrates this behavior, where the shaded area corresponds to the states reachable from S visited at each step. In the algorithm presented in this figure, function $T(S)$ returns the set of states that are successors of some state $s \in S$; R and R' represent sets of states.



proc *MIN* (*S, F*)
$i = 0$;
R =S;
$R' = T(R) \cup R$;
while $(R' \neq R \wedge R' \cap F = \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T(R) \cup R$;
if $(R' \cap F \neq \emptyset)$
    then return i;
    else return ∞;

Figure 2.8: The MIN algorithm determines the minimum path between state $s \in S$ and state $f \in F$, in a breadth-first search way.

2.2.4 SMV - An OBDD Symbolic Model Checking Tool

SMV [66] is a symbolic model checker that implements all theorical features described above (Kripke model, CTL, and OBDD). SMV models a problem as a finite state machine. It is able to model synchronous and asynchronous problems. Non-determininsm is provided and it can be used to model uncertaint events or unavailable modeling information. The properties about the model can be specified by CTL or LTL expressions. The input language provides code encapsulation (module) and passage of parameters among modules. The semantics of the assignment is similar to the assignment of data flow languages. An example of SMV program is presented in Figure 2.9. It is an an excerpt of a model relating to the microwave example presented in Section 2.1.3.

An SMV model is implemented by **modules**. The modules models processes that run in parallel and all SMV programs must be constructed with at least the module main. The microwave model presented in this SMV program has two modules: the module main,

```
1   MODULE main
2   VAR
3     door : {opened, closed};
4     cook : boolean;
5     setup : 0..5;
6     m : mw_cell (setup, cook);
7   ASSIGN
8     init (door)   := closed;
9     next (door)   := {opened, closed};
10    init (setup)  := 0;
11    next (setup) := case
12                        m.i = 0 : {0,1,2,3,4,5};
13                    esac;
14    cook := ((door = closed) & (setup > 0));

15  SPEC !cook → EF m.cooking
16  SPEC cook → AF m.cooking

17  MODULE mw_cell (time, go)
18  VAR
19    i : 0..5;
20  ASSIGN
21    init (i) := time;
22    next (i) := case
23                  go & (i > 0) : i - 1;
24                  1 : i;
25                  esac;
26  DEFINE
27    cooking := i > 0;
```

Figure 2.9: A SMV model for the microwave example presented in Section 2.1.3.

ranging over lines 1 to 14, models a microwave user, setting the clock time for cooking a meal, and openning or closing the door of the microwave; the module mw_cell, ranging over lines 17 to 27, models the microwave cooking a meal. This SMV program also has two specifications in lines 15 and 16. These specifications are CTL properties that the SMV must verify.

The *VAR* section of a SMV program determines the state space of the Kripke structure, since the state space is composed by all possible combinations of values of all variables of the model. Variables in SMV can be one of the following types: *boolean*; *scalar*; *numeric*, that is a range of integer numbers; or *user_defined*. Array of variables are allowed. The lines 3 through 6 and line 19 present the variables declared for the microwave example.

The variable door (line 3) represents the door of the microwave. Its type is scalar what means that it can assume one of the values stated between the brackets, *opened* or *closed*, in this case. The line 4 presents the declaration of variable cook. This variable is boolean and models the microwave state of cooking a meal. If cook is assigned to true, then the microwave is cooking. If cook is assigned to false, then the microwave is idle. The variable setup in line 5 is numeric and its range varies from 0 to 5. This variable represents the possible set up time of the microwave for cooking a meal. Line 6 presents an example of user_defined type. m represents the microwave action of cooking a meal, that is, it controls the passage of cooking time. Its behavior is defined by the module mw_cell declared in lines 17 through 27.

The *ASSIGN* section of a SMV program defines the transition relation of the model. This section defines the initial and the next valuations of the program variables. Each valuation determines a state in the spcae of the model. In this program example, the *ASSIGN* section ranges over lines 7 through 14, in module main, and over lines 20 through 25, in module mw_cell. The *init* and *next* declaration along these lines define, respectively, the initial value and the next value of the variables in the state space.

Line 8 states that the variable door is initially assigned to *closed*. This means that the microwave door is always closed in the initial state of this model. Just considering this variable, if we have omitted this line, we would have two initial states, one state in which the value of variable door is *opened*, and another state in which its value is *closed*. The value of door in a next state can be *opened* or *closed*, as defined in line 9. This kind of assignement is non-deterministic.

Lines 10 and 11 relate to variable setup. Assigning a value to this variable models a person setting the microwave timer for cooking a meal. Initially this variable is assigned

to 0. Its value in a next state depends on the condition m.i = 0, in the *case* construction. If m.i is greater than 0, it means that the microwave is cooking a meal, hence a new setup time is not allowed. Otherwise, if m.i = 0 yelds to true, the variable setup is assigned non-deterministically to any integer value between 0 to 5.

The variable cook indicates the condition for the microwave cooking a meal. cook is assigned to true or false depending the result of the boolean expression stated in line 14. Finally, the variable i of module mw_cell represents the clock time of the microwave. Initially, it is assigned to the value assigned to setup, line 21. At each state, i is decremented by 1, if door is assigned to *closed* and setup is greater than 0, line 23. Note the relation between line 23 and line 14.

The *SPEC* section of a SMV program is where we declare the properties (specifications) we want to be verified. In this model, we are interested to know if it is possible the microwave cooks a meal having its door opened or its clocktime set to 0 (see line 15). We are also interested in determining if the microwave really cooks a meal since all conditions for cooking are set (line 16).

SMV can be found fully documented at Carnegie Mellon University [79] or at Istituto per la Ricerca Scientifica e Tecnologica [26], where also can be freely downloaded.

2.2.5 Verus - A Symbolic Model Checking Tool for Real Time Systems

Verus is a formal verification tool originally designed for time critical systems. It is based on CTL-OBDD-SMC technology and provides features specific time features, such as: deadline, delay, and priority. Verus also provides quantitative timing information about the model, such as: the minimum and maximum time interval between two events; the number of times an event has occurred in a given interval. These kind of informations are very important to detect system errors and/or optimize the system parameters.

The time in Verus is discrete, although some other tools adopt continuous time approach [3, 52]. The continuous time demands a large state space to model a system, in general inhibiting the modeling of complex systems. Discrete time was project decision: it allows Verus to cope with very complex systems, with a state space of the order of 10^{30} states [18, 20]. It has been used successfully in different types of system, such as airplane controllers [20], robotics [19], hardware design [18], flexible manufacturing [50], and multimedia systems [21].

```
1   boolean occurred;

2   alarm () {
3       boolean ring;
4       while (true) {
5           if (occurred) ring = true;
6           else ring = false;
7           wait (1);
8       }
9   }

10  event () {
11      while (true) {
12          occurred = select{false, true};
13          wait (1);
14      }
15  }

16  spec
17      AG (occurred -> AF alarm.ring);
```

Figure 2.10: A Verus program that models a non-deterministic event and its respective alarm.

The Verus modeling language is similar to C language. This language paradigm is well known by the majority of designers and generally minimizes training. To present the Verus language, we will model a non deterministic event and an alarm (that rings when the event occurs). A Verus program relating to this scenario is presented in Figure 2.10.

This program presents two functions: alarm and event. Each Verus function models an independent process. The function alarm is declared over lines 2 through 9. It models an alarm that rings every time its associate event occurs. The function event models a non-deterministic occurrence of a certain event, and it is declared over lines 10 through 15. As we can see the function main is not mandatory.

The communication between funtions occurs by global variables. A variable is global when it is declared outside a function, like variable occurred at line 1. This variable is *boolean* and models the occurrence of the event. The other variable type recognized by

Verus is *integer*.

The function alarm models an alarm that rings as soon as an event occurs. The boolean variable ring at line 3 models the ringing of the alarm. If this variable is true then the alarm is ringing. The function alarm loops forever assigning ring according to the value of occurred. If occurred is true, this means that the event has occurred, then ring must be set to true. The alarm will keep on ringing until the event be ceased, that is, the variable occurred be set to false. The variable occurred is assigned by the function event. This function loops forever assigning occurred non-deterministically to true or false (line 12).

The passage of time in the model is implemented by the *wait* statement, lines 7 and 13. This statement is responsible for creating the states of the model space. The functions of a Verus program can only perceive the new values of the program variables after *wait* statements.

Lines 16 and 17 present the specification of a property that Verus must verify. The property of line 17 informally speaking specifies that "everywhere in the model (AG) that event occurs (that is, the variable occurred is true), always in the future (AF) the alarm will ring (that is, the variable alarm.ring is true)".

Verus is fully documented and is free for download in the site of Prof. Sérgio Campos, at Federal University of Minas Gerais (http://www.dcc.ufmg.br/~scampos).

Chapter 3

Symbolic Model Checking over a Video on Demand System

The development of new technologies for high bandwidth networks, wireless communication, data compression, and high performance CPUs has made it technically possible to deploy sophisticated infrastructures for multimedia applications [44, 80]. This type of infrastructure opens up opportunities for exploring multimedia applications such as quality audio and video on demand (from home), virtual reality environments, digital libraries, and cooperative design. Figure 3.1 illustrates an infrastructure for such applications.

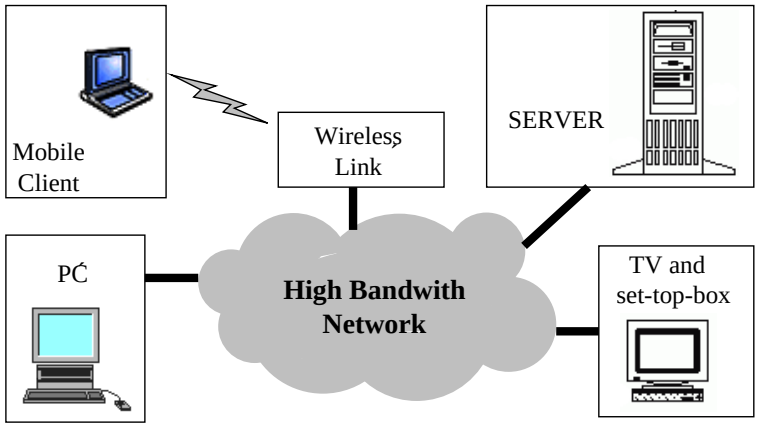


Figure 3.1: Overall architecture for a video service.

The user has access to the applications mentioned above through a laptop (connected, for instance, to a wireless link), a PC-based workstation, or a TV set connected to a set-top box. Due to the high interactivity of such applications, the success of the service is heavily dependent on the delays incurred in the high bandwidth network, on the delays incurred

at the server and client machines, and on the performance at the multimedia server. All these delays will determine the number of clients a VOD server can cope with at the same time.

A VOD server is quite distinct from conventional servers, such as database and Web servers, which do not have to take into account strict time constraints. In a VOD server, failure to meet the time application constraints will certainly lead to user dissatisfaction and consequently to risks of commercial failure. Further, to be cost effective a VOD server must present good performance (which is usually measured as the number of users which can be served simultaneously).

By the time we have been studying Verus the ALMADEN project was in course. ALMADEM was a project financed by the Brazilian Ministry of Science and Technology, whose main purpose was the research about applications and algorithms for high performance multimedia networks. One of the products of this project was the ALMADEM-VOD server [69], whose prototype was operational, but without any formal evaluation about its performance bounds.

This chapter describes our work in the ALMADEM project, in determining the performance bounds of the ALMADEM-VOD server. We describe the ALMADEM-VOD server and its respective Verus model. We also describe the verification of this model and the minimum (in the worst case) and the maximum number of clients (in the best case) determined by Verus.

3.1 The ALMADEM-VOD Server

The fundamental premise in the development of the ALMADEM-VOD server is that it should use only off-the-shelf low cost components, as also done in [44, 70, 80]. As a result, the server was implemented on a PC-based platform running the Linux operating system. To fulfill the real time requirements of the video application, the operating system was adapted in specific points such as the disk access and process scheduling routines.

Considering the video service architecture presented in Figure 3.1, a user accesses a Web interface through a remote client machine to select a film. Once the film is selected, a requisition for a stream for that film is sent to the server, through a TCP connection. The server then runs an admission control routine which schedules the request, if there are enough resources available. Typically, the main bottleneck is disk bandwidth. Once the request is scheduled, blocks of data are sent periodically to the client in push mode

as UDP messages. This software organization corresponds to the ALMADEM-VOD video server, and is illustrated in Figure 3.2.

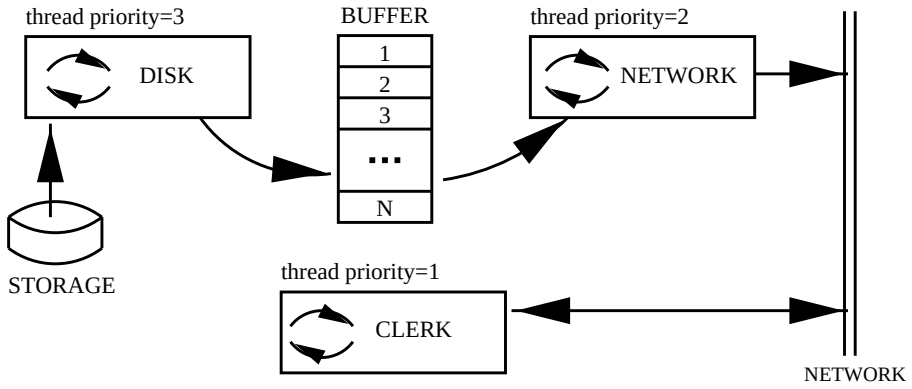


Figure 3.2: Software architecture of the video server.

Two data structures and three separate processes are distinguished. The data structures are called *storage* and *buffer*. The processes (implemented as POSIX *threads*) are called *disk*, *network*, and *clerk*. To ensure proper timing in the scheduling of these threads, we rely on one of the real time scheduling policies available with the Linux operating system. We use the SCHED_FIFO policy which implements a first-in-first-out scheduling scheme with static priorities. In this policy, the priority number of a thread is direct proportional to the system priority. Despite its simplicity, this scheme works quite well if the machine is dedicated to the video server task (i.e., we run the video server in run level 1).

The *storage* structure is composed of secondary or tertiary devices and is used to hold the collection of films available to the users. The current implementation of the ALMADEM-VOD server considers only secondary devices in the form of conventional SCSI-2 [78] disks of 4G bytes each. The disks store the films encoded and compressed in MPEG-1 [67] format. Each film is divided in *blocks* which are retrieved for delivery to the client machine. In its simplest implementation, which is adopted in this study, the ALMADEM-VOD server considers a *contiguous* layout of films on disk. In this layout, all blocks of a same film are stored contiguously on disk. More sophisticated layout schemes, involving striping techniques [10, 27], region-based allocation [46], and randomized placement [76], have been discussed extensively in the literature but are not the focus of this work.

The *buffer* structure is basically main memory space used to synchronize the *disk* and *network* threads. It is implemented as a circular buffer which is filled by the *disk* thread and emptied by the *network* thread.

The *network* thread is responsible for taking the blocks of film from the buffer and shipping them across the network. It is scheduled whenever the *disk* thread is blocked at the disk driver waiting for a disk access to complete.

The thread named *clerk* listens at a TCP port for the requests from the client machines. Such requests might come from new clients or from a current client which requests, for instance, a *pause* in the exhibition. Once it detects a client request, this thread passes the information to an admission control routine for proper scheduling. If the server is saturated (i.e., it is currently serving a maximum number of clients), a request for a new stream is not scheduled and a denial message is sent to the respective client.

The *disk* thread is responsible for reading the data from the secondary storage and storing them at the buffer area. To avoid delays introduced by the operating system (which we cannot control), disk accesses are performed through direct access functions which communicate directly with the SCSI controller device. For each active client, a separate block (which is composed of several MPEG frames) of (average) size B bytes is read, passed to the buffer area, and from there shipped (by the *network* thread) to the corresponding client machine. While that client consumes the frames in that block, other clients can be attended to. This *cyclic scheduling* process is repeated with a fixed time period equal to T, as illustrated in Figure 3.3. To implement this fixed time period, the *disk* thread monitors the Real Time Clock (RTC) device in the Linux kernel.

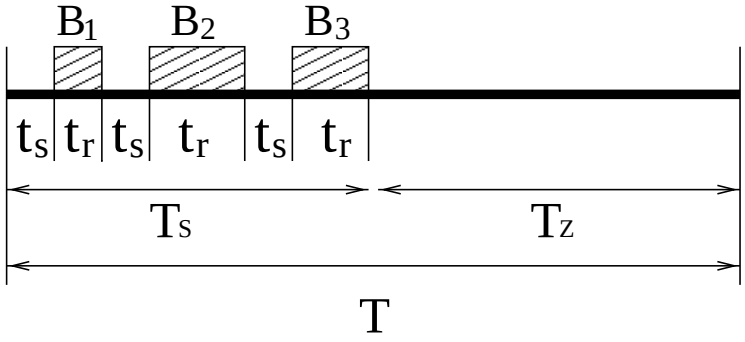


Figure 3.3: Service cycle with a duration of T seconds. t_s accounts both for seek and rotational delay times.

The time *period* T defines the *service cycle*. At each service cycle, all scheduled clients are served. To serve a client, the *server* incurs two fundamental delays: a seek time t_s and a transfer time t_r. The time t_s is the time to position the disk head at the proper cylinder, plus the time to position the disk head at the proper block of data in this cylinder (i.e., t_s includes rotational latency time). The time t_r is the time to read the block of data from disk. The total time T_s spent serving all clients in the system is called the *service*

time. The *sleeping time* T_z is the portion of the service cycle in which no clients are served. Such sleeping time is necessary because, to avoid buffer overflow at the client machine, the server does not attend a same client twice in a service cycle. The ratio T_s/T defines the *occupation* of the service cycle. Larger the occupation of the service cycle, higher is the load in the system.

In the ALMADEM-VOD server (as seen in Figure 3.3), the transfer time can vary from one film to another because the block sizes, though constant for a same film, differ from one film to another. The reason is that the coding scheme might vary from one film to another (for instance, a film might be encoded for a smaller window size) and that the compression rate is not constant across various films. The important detail is that, in the ALMADEM-VOD server, any block of any film is composed of roughly a same number of frames, which defines the duration of the service cycle. To exemplify, consider that each client consumes frames at the typical rate of 30 fps (frames per second). Then, if each block sent to a client includes 30 frames, the value of T is 1 second to avoid interruption in the continuous display of the film at the client machine. If each block includes 120 frames, then the value of T is 4 seconds.

To simplify the implementation, the ALMADEM-VOD server uses a Constant Data Length (CDL) block instead of a Constant Time Length (CTL) block [80]. The length of the blocks in which a given film is divided is determined by the maximum consumption rate at the client. This ensures smooth display at the client machine. However, buffer overflow might occur at the client because the rate of arrival exceeds the average consumption rate. To avoid this problem, the client sends a *pause* message to the server whenever it detects that its buffer is filling up.

Let Rc_i be the rate (in bytes per second, or Bps) with which the ith client consumes a block of data. Further, let B_i be the size (in bytes) of the blocks of data for the ith client, as indicated in Figure 3.3. Then, the period T is given by

$$T = \frac{B_i}{Rc_i} \tag{3.1}$$

Additionally, let Rd be the transfer rate of the disks in our secondary storage and let N be the maximum number clients which can be served in a cycle. We can then write

$$\sum_{i=1}^{N} B_i = (T - N\ t_s)Rd \tag{3.2}$$

By substituting equation (3.1) into equation (3.2), we obtain

$$N = \frac{T}{t_s}\left(1 - \sum_{i=1}^{N} \frac{Rc_i}{Rd}\right) \tag{3.3}$$

Equation 3.3 shows that the maximum number of clients in the system is a direct function of the ratio $\sum_{i=1}^{N} Rc_i/Rd$ and thus, that the sum of all rates Rc_i must be smaller than the disk transfer rate R_d. Furthermore, the average block size (which determines the duration of the cycle service) must be large enough to provide an amortization of the time wasted with seek operations. In fact, a small average block size reduces the value of T making the fraction T/t_s smaller. This implies that the fraction of time available in a cycle for actually reading data from disk is smaller. This leads to a reduction in the maximum number of clients which can be attended simultaneously.

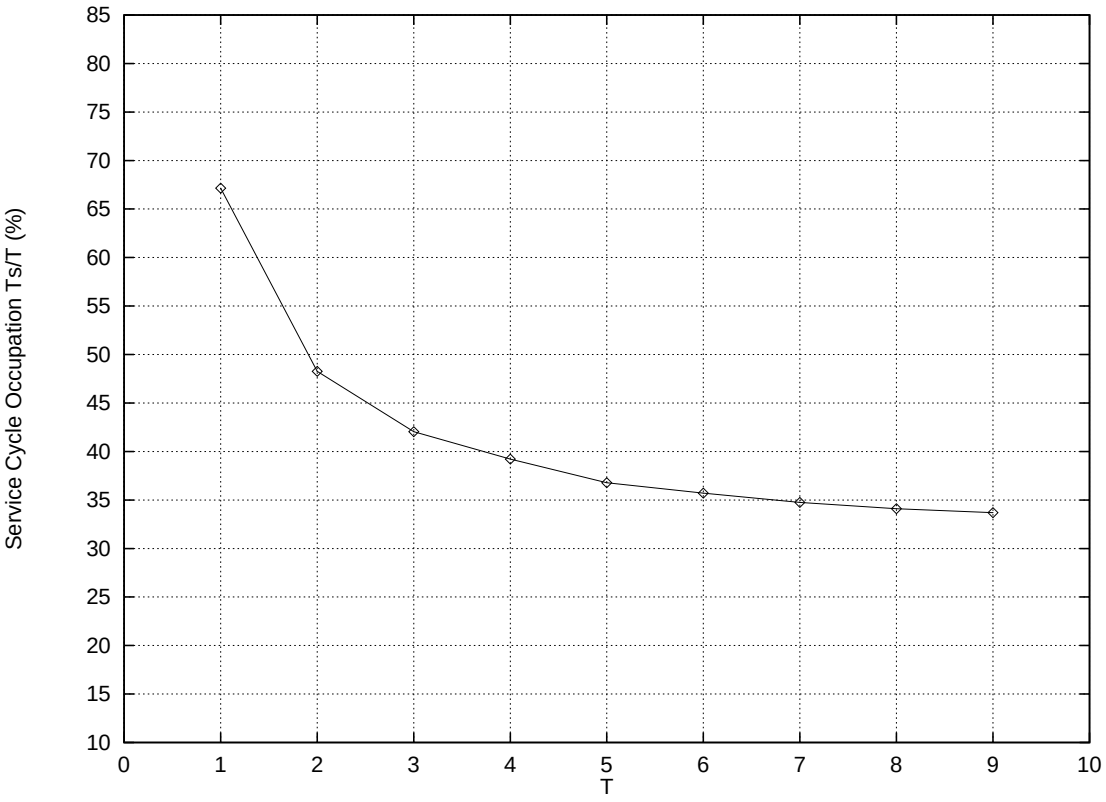


Figure 3.4: Service cycle occupation in the ALMADEM-VOD server for periods varying from 1s to 9s. The number of clients in the system is 20.

Amortizing seek time (through an increase in the service cycle) is critically important because it improves server performance (in terms of the maximum number of clients which can be served). However, an excessive increase in the service cycle is counter-productive because it implies in an excessive *latency* — the time a new user waits to be served. This is because new users are only served in the cycle which initiates following their arrival. Additionally, large service cycles require larger memory buffers at the server and at the client machines.

Figure 3.4 illustrates the occupation of the service cycle in the ALMADEM-VOD server

for values of the period T varying from 1 second to 9 seconds. The number of clients in the system is 20. When $T = 1$ the occupation of the service cycle is high because of seek time (t_s). The system spends more time doing seek than reading block of films. When T is increased, seek time is amortized by the reading time. As shown, the amortization of the seek time is less significant after $T = 7s$ and does not improve much after $T = 4s$. For periods larger than 4 seconds, the server incurs higher latency without considerable additional gains in server performance. Thus, for the ALMADEM-VOD server, we adopt $T = 4s$ as a good compromise between client latency and server performance.

At each client machine, it is necessary to run the ALMADEM-VOD *client* module which is composed of three sub-modules: *network*, *buffer*, and *decoder*. The *network* component is implemented as a thread which receives blocks of film from the server and saves them in the *buffer* data structure. From the buffer, each block is passed to the *decoder* component through a Linux pipe. This architecture isolates the decoder (which is normally a commercial piece of software) from the client code (which we implemented) and provides for great flexibility. For instance, we were able to substitute the MPEG decoder for an audio player in a couple of hours.

3.2 Modeling the ALMADEM-VOD Server in Verus

Verus has a specification language that is similar to the programming language C. The Verus language provides special primitives that allow the user (i.e., designers and engineers) to model timing aspects of a system such as deadlines, priorities and time delays. Thus, modeling a multimedia system in Verus resembles writing a C program. A Verus specification is converted into a state transition graph and efficient search algorithm determines if the model satisfies the properties.

The modeling of the ALMADEM-VOD server has occurred while the designers of this system were developing it. The system parameters we have considered in the model were obtained from the version that was running at that time. The system parameters are: (1) a period of 4 seconds; (2) a consumption rate Rc_i at each client machine which varies from 1Mbps to 1.2Mbps (mega bits per second); (3) a film block size B which varies from 500 KBytes to 600 KBytes; (4) a contiguous layout of the blocks of each film in disk; (5) a disk transfer rate of 7.8 MBps (mega bytes per second); (6) a disk seek time (including rotational delay time) which varies from 10 ms to 20 ms (milliseconds). The disk transfer rate were determined by reading disk blocks from disk. Although the nominal disk transfer rate of the disk is 10 MBps, reading 500 KBytes to 600 KBytes disk blocks of films has led

```
1    while (true) {
2        ...
3        seekTime = select {2,3,...,19,20}; /* valid seek time */
4        while (seekTime > 0) {
5            wait (1); /* passage of seek time */
6            seekTime = seekTime - 1; }
7        readingTime = select {12,13,...,19,20}; /* valid disk time */
8        while (readingTime > 0) {
9            wait (5); /* passage disk transfer time */
10           readingTime = readingTime - 1; }
11       ⋮
12   }
13   ⋮
14   /* minimum and maximum service time in a cycle of 4 seconds*/
15   MIN (beginningOfService, endOfService);
16   MAX (beginningOfService, endOfService);
```

Figure 3.5: An illustration of the ALMADEM-VOD server specification in Verus.

to disk transfer rate presented in item 5.

As mentioned before, performance in a VOD system is measured by the **number of users** it can serve simultaneously. The designers of the ALMADEM-VOD system had already measured the system performance, but they did not have any information about how close to optimum this performance was. Since the ALMADEM-VOD server is the bottleneck of the system, we have modeled it to determine the **minimum and maximum number of users**, respectively the worst and best case, the server can support in a cycle of service (4 seconds).

A simplified excerpt of the specification of the ALMADEM-VOD server is presented in Figure 3.5. The variable SEEKTIME corresponds to the seek and rotational delay times (t_s), and the variable READINGTIME relates to the transfer time (t_r). The model loops forever serving blocks of films to the clients (lines 1-12). As the requested block of film can be in any place of the disk, the block position is randomly selected (line 3). This position is related to the seek time the read-write disk head needs to reach the requested position from the current one. The values 2 through 20 represents units of milliseconds. The movement of the read-write disk head to the target is represented by lines 4 to 6. The transfer time is also randomly selected (line 7) because the size of blocks also varies. The values 12 through

20 corresponds to transfer time ranging from 60 to 100 milliseconds. The transferring is represented by lines 8-10, and each *wait* (line 9) corresponds to 5 milliseconds. The MIN and MAX statements (lines 15 and 16) relate to the quantitative algorithms mentioned in Section 2.2.3. These algorithms are able to determine, respectively, the minimum and maximum distance between two events. In the case of this model the events are the beginning and the end of a cycle of service. Using these algorithms we were able to determine the bounds of the of users served simultaneously.

3.3 Results

We have compared the quantitative results provided by Verus with the empirical results obtained from the server. The critical system parameters, discussed in Section 3.2, are the same both for the Verus model and for the version of the server used in our experiments.

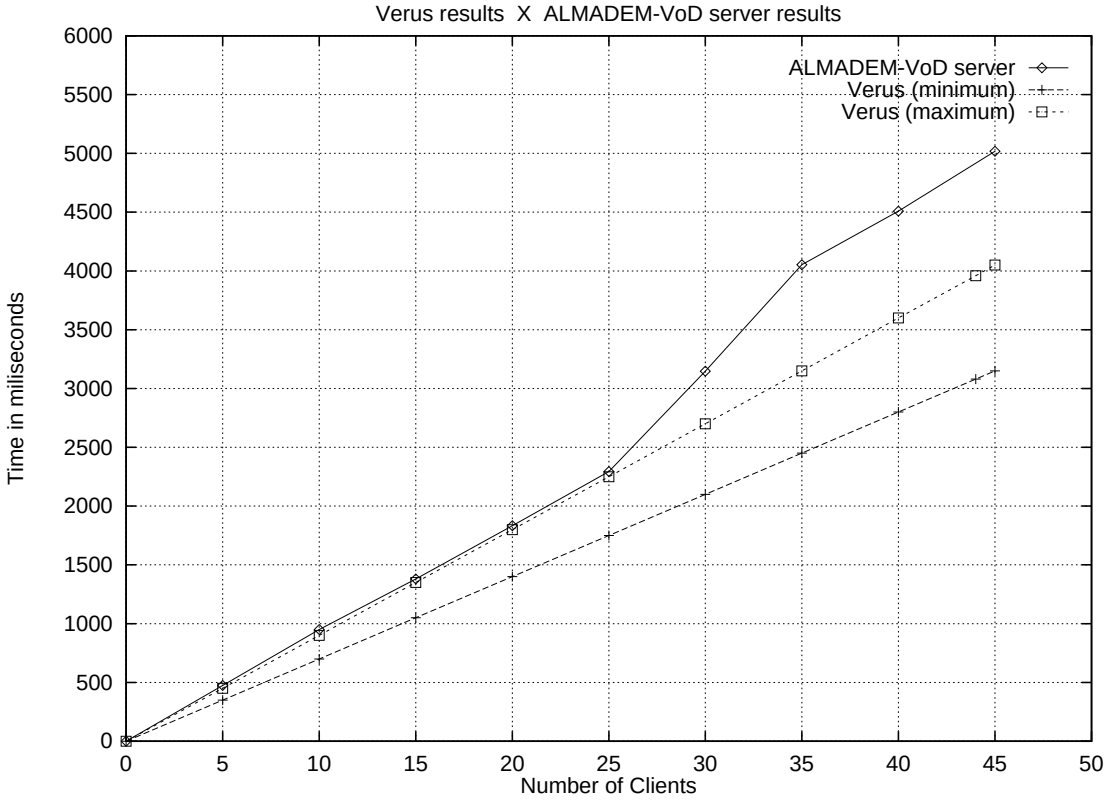


Figure 3.6: Variation of the service time occupation by clients for the server and for Verus ($T = 4s$).

In Figure 3.6, we illustrate how the service time evolves as the number of clients in the system increases. The continuous curve is relative to measurements obtained from the

ALMADEM-VOD server, while the dashed curves illustrate the minimum and maximum service times according to Verus. We observe two major facts: (1) the ALMADEM-VOD server always operates at or above the maximum service time predicted by Verus; (2) the ALMADEM-VOD server presents an unexpected non-linearity in its service time when saturation has not been reached yet (i.e., with 25 clients in the system).

These two observations motivated a through inspection of the implementation of our server and of the Verus model we built, in an attempt to make the results generated by the ALMADEM-VOD server and by Verus consistent.

We have analyzed the dynamic behavior of the Verus model and it was working properly. We have then measured once more the various system parameters we were using and found a problem. The Verus model we built considers a constant disk transfer rate. However, modern disk devices use a multi-zone organization in which there are more sectors per track in the outer tracks. As a result, the outer tracks yield a higher transfer rate because their tangential speed is higher. To correct the problem, we have changed the Verus model to consider that the transfer rate assumes basically 4 distinct values depending on the track the block of data is in. The new predictions for the service time are shown in Figure 3.7.

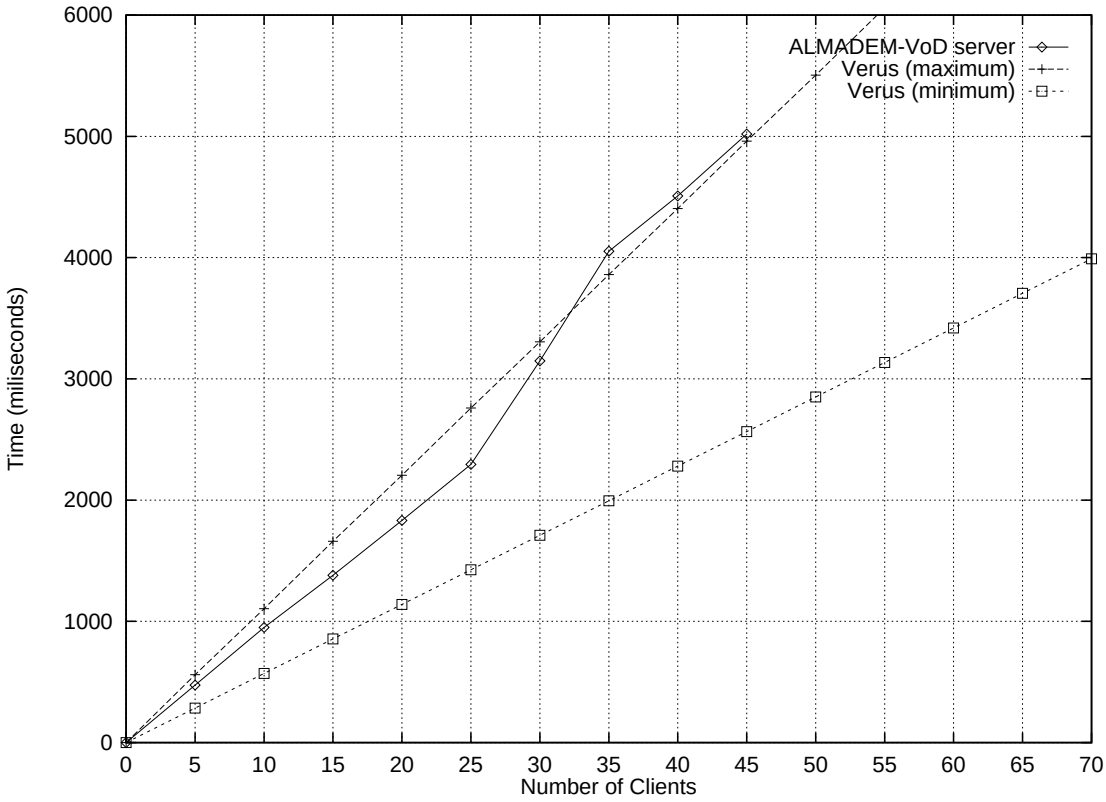


Figure 3.7: Service time for the ALMADEM-VOD server and for the revised Verus model ($T = 4s$).

We observe that now the maximum service times predicted by the revised Verus model are higher. However, under heavy load, the service times obtained from the ALMADEM-VOD server continue to exceed the maximum service times predicted by Verus. Further, the non-linearity in the service time of the server is still without explanation. Clearly, the implementation of the server is running into non anticipated overheads. In the search for an explanation, we have investigated the code for the VOD server and then noticed a peculiarity which had not been accounted for in the Verus model we built. This peculiarity is as follows.

In our laboratory, the video server sends blocks of film to its clients through a Myrinet switch which runs at a raw bandwidth of 1 Gbps (giga bits per second). At this bandwidth, conventional implementations of the link layer are unable to handle all the data which arrives at the network physical device. The result, which we observed systematically, is that packets of data are lost at the client machine if a block of film large enough is passed at once to the Linux link layer at the server side. Unfortunately, the block sizes which the server uses (between 500 KBytes and 600 KBytes) are large enough to cause the problem.

To deal with this problem, we have changed the implementation of the *network* thread to include the notion of mini-cycles. In each mini-cycle, only a portion of each block of film (called a *mini-block*) is sent to each client. While the link layer of a client X handles the reception of a mini-block, mini-blocks can be sent to the other clients in the system. By doing so, we avoid overloading the link layer at client machines. As a result, packet losses are no longer observed.

Mini-cycles are a technical solution to a technological mismatch i.e., current network devices are too fast for conventional operating systems. In this regard, mini-cycles are not really a part of the design of the ALMADEM-VOD server and were not considered in the Verus model we have built. Since we have tested our implementation of mini-cycles extensively, it seemed to us that mini-cycles would not interfere with the server operation. However, they do. If we simply run the mini-cycles, in situations of light load (for instance, when there is only one client in the system) too many mini-blocks will be sent to each client machine at once, overloading the corresponding network device. To deal with this type of problem, we have decided to put the *network* thread to sleep within a mini-cycle, such that each mini-cycle would have a minimal duration. As a result, the dynamic behavior of the system is now far more complex, because we have to manage several mini-cycles (with service and sleeping times) within each service cycle. At this point, a programming mistake was done.

To simplify the dynamics of the server operation, an attempt was made to guarantee

```
disk {                          network {
  while (1) {                     while (1) {
   ...                             ...
   pthread_cond_signal(&cs);       pthread_cond_wait(&cs,&mtx);
   ...                             ...
   Read blocks from disk;          mini-cycles();
   ...                             ...
  }                               }
}                               }
```

Figure 3.8: Synchronization of disk and network threads through a pair of *signal* and *wait* primitives.

that the *network* thread would not starve neither be overflow with too much data. To accomplish this effect, the programmer synchronized the *network* and *disk* threads through a pair of *signal* and a *wait* primitives, as illustrated in Figure 3.8. The idea is that the *network* thread would wait until the *disk* thread indicates that the blocks of film are available in the buffer. This was decided at implementation time as an extra measure to ensure consistent behavior. However, the side-effect is that true concurrency is prevented which results in considerable overhead when the system operates in situations of medium to high load. This overhead led to the results observed in Figure 3.7. To fix the problem, we removed the *signal* and *wait* primitives from the code. As a result, the evolution of the service time is now as illustrated in Figure 3.9.

3.4 Conclusion

The advantages of the use of formal verification for analyzing systems are well known. Some of them are the capability of finding subtle errors, and the improvement of the understanding of the system being analyzed. Both advantages have been materialized in the context of the ALMADEM-VOD.

The formal verification of the ALMADEM-VOD server has led to a better understanding of the server operation and has revealed inefficiencies in the server. The understanding about the server has been improved because to model the server in Verus we had to study the system requisities and its architectural organization together the team project. This study has facilitated the search and the finding of the causes of the inefficiencies pointed out by Verus.

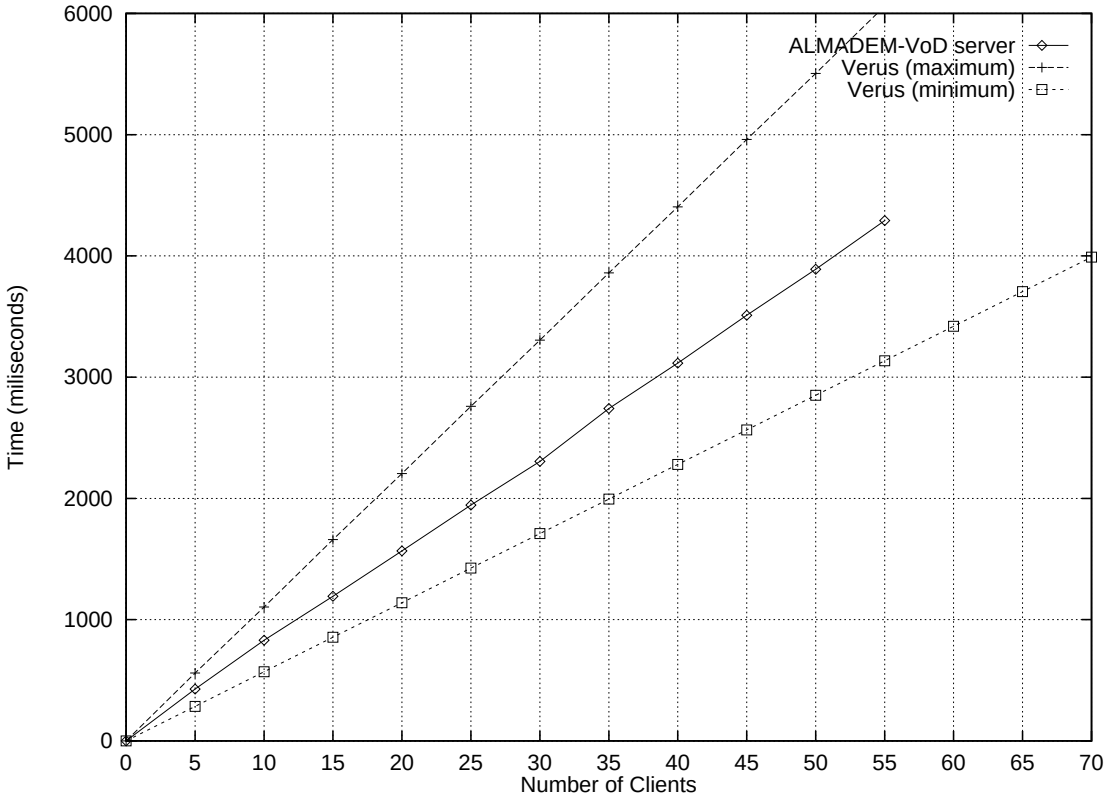


Figure 3.9: Service time for the new versions of the server and of the Verus model ($T = 4s$).

As observed, the service time for the ALMADEM-VOD that had an erratic behavior now increases linearly. Further, it is within the minimum and maximum service times indicated by Verus. Since the period is 4000 milliseconds (i.e., $T = 4s$), we expect that the server will be able to attend a maximum of 50 clients on average. According to Verus, this maximum will be close to 35 clients in the worst possible situation and close to 70 in the best case scenario. This has represented a performance improvement around 40%.

The use of Verus has been shown a good choice. The quantitative algorithms has been very effective in determining the performance bounds of the server. These bounds relate to the minimum and the maximum number of clients, respectively, in the worst and best case. Worst case in this context is related with a series of client request that impose frequent hard disk movement, such that each movement has great amplitude. In the best case, the movement amplitude is very small. Besides, the Verus modeling language resembling C has allowed an effective participation of the project team during the verification process. This participation was determinant for the better understanding of the server mentioned above.

Chapter 4

Scheduling

Scheduling [6] is a decision-making function and a body of a theory. As a function, it determines an allocation of resources to tasks over a period of time. This allocation is computed regarding to an optimization criterion. As a body of a theory, it comprehends a collection of models, principles, techniques and conclusions about the scheduling function.

Scheduling problems arise when the necessary resources for performing a set of tasks are scarce. In general, the resources are machines, but can be any kind of good, such as money, water. Due to the scarcity of resources, it is necessary to determine the best order for executing the tasks, so that the overall result accomplish a certain optimization criterion. In general, choosing the "best order" implies to investigate all possibles orders. Since, the number of possible orders is a combination of all possible values of the problem parameters, some instance problem can be intractable. Therefore, special approaches are necessary to deal with them. That is the role of Scheduling area.

There are many examples of scheduling problems in our daily life, such as economics, logistics, transport, sports, computing. A common one occurs in airports and bus stations. Consider an airport that has dozens of gates (resources) and hundreds of airplanes (tasks) that arrive and depart daily. The problem in this scenario is to determine the best schedule (airplanes-gates) such that the departure delay of airplanes is minimum. It is not a simple problem since there are different sizes of gates and airplanes; different amount of time necessary for preparing all kinds of airplane for landing and taking-off; and there is non-determinism in weather condition and in a great sort of events (airplane malfunction) [73].

This chapter describes our study about Scheduling. Our interest for this area appeared after reading an article by Adams, Balas and Zawack [2]. This article presented an algorithmic approach for a problem known as *job-shop problem*. This class of problem is known

to be NP-hard [45] and our interest since then has been related to studying how Symbolic Model Checking (SMC)[1] can solve this specific kind of problem. Being that, this chapter presents the basic concepts about Scheduling, the class of problem known as *workshops*, traditional approaches to solve workshops, and how the SMC approach differs from these traditional ones.

4.1 Basic Concepts

The complexity of a scheduling problem is determined by its environment, which is usually stated in terms of three fields $\alpha|\beta|\gamma$ [61].

The α field is associated with the machine environment. A problem can involve one or more machines. These machines can be parallel identical machines, identified by P; uniform parallel machines, identified by Q; or parallel unrelated machines, identified by R. Q represents non identical machines but related to each other by some way, for example, uniform speed. R represent machines that are not related to each other in any way. When the jobs are compounded by multiple operations and an ordering is imposed on these operations, we have a scenario known as *workshop*. If the ordering is the same for each job, we have a flow-shop, designated by F. If the ordering is not the same for all jobs, than we have a job-shop, indicated by J. If no ordering is imposed on the operations, we have an open-shop, identified by O.

The β field is associated with the job characteristics, such as: preemption, identified by **pmtn**; release time, identified by **r**; and order of precedence in the jobs, indicated by **prec**. Finally, the γ field is concerned with the optimality criterion. It defines the objective of the problem, i.e. the criterion to be optimized. Some of these criteria are: c_j, completion time of job j; $\sum c_j$, total completion time of the schedule; C_{max}, completion time of all jobs (also known as *makespan*); L_j lateness of job j; L_{max} lateness of all jobs. Examples of scheduling problems stated in terms of $\alpha|\beta|\gamma$ is presented below:

$1|prec|C_{max}$ a scheduling problem in one machine; the jobs have precedence among them; looking for the makespan (the minimum completion time of all jobs);

$R2|r_j, pmtn, prec|L_{max}$ the problem occurs in a 2 unrelated machines; the jobs have release time, can be preempted, and there is a precedence order among them; the optimization criterion is to minimize the total lateness.

[1] See Chapter 2.

$J10||C_{max}$ - the problem is a job-shop in 10 machines and the optimization criterion is the makespan.

In coarse grain, we can organize Scheduling problems as stochastic and deterministic problems. Stochastic scheduling problems have its parameters defined as probability functions [73]. For example, the workers assembly one bike type Y in 23 minutes in 30% of times, and in 70% of times they assembly in 32 minutes. In deterministic scheduling problems all parameters are strictly stated. There are two variants in this class of problem: Dynamic and Static. Dynamic scheduling problems are characterized by the fact that events can modify an already established running schedule. For example, a high priority contract is firmed and all schedule production must be modified. In static scheduling problems all parameters are known in advance and they do not change. This taxonomy is illustrated in Figure 4.1 [38].

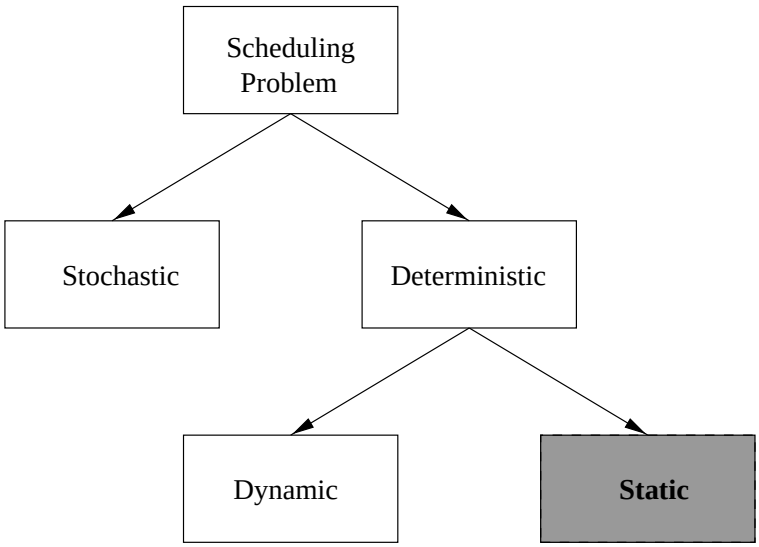


Figure 4.1: A taxonomy for scheduling problems. The shaded rectangle represents the class of deterministic static scheduling problems that is the focus of this thesis.

Among deterministic static scheduling problems there are some problems that are known as *workshops*. The workshops characterize by having multiple operations in each job and an imposed ordering over these operations. Depending of this ordering, the workshop can be a flow-shop, a open-shop or a job-shop. Therefore, in the next sections, we describe these problems and also present some traditional approaches that give solution to them.

4.1.1 Workshops

A workshop environment consists of a set M of machines and a set T of tasks. The set T is partitioned and each partition of T defines a job. The number of partitions is determined by the number n of jobs. The workshop varies according to the kind of ordering imposed to tasks. Depending on the kind of ordering, the workshop is a flow-shop, a open-shop or a job-shop.

A *flow-shop* has the following characteristics: $|M| = |job\ i|$ such that $job\ i$ is a partition of T, for all $i, 1 \leq i \leq n$; and if the machines are numbered 1, 2, ..., m, the tasks of $job\ i$, $i, 1 \leq i \leq n$, are numbered $(i,1),(i,2),\ldots,(i,m)$. Besides, the sequence of task execution is the same for all jobs. In other words, the tasks of all jobs have to be executed by the same sequence of machines. Examples of flow-shop are the pipeline in a computer as DLX computer [51] and the assembly line of an automobile. The Figure 4.2 [71] illustrates a flow-shop. In this Figure, the job 1 is executed by machines 1, 2, ..., m and, in this case, task execution order is dictated by this machine sequence: task 11 (task 1 of job 1) is executed by machine 1, task 12 (task 2 of job 1) is executed by machine 2, ..., task $1m$ (task m of job 1) is executed by machine m. This same sequence of task execution is applied to all other jobs.

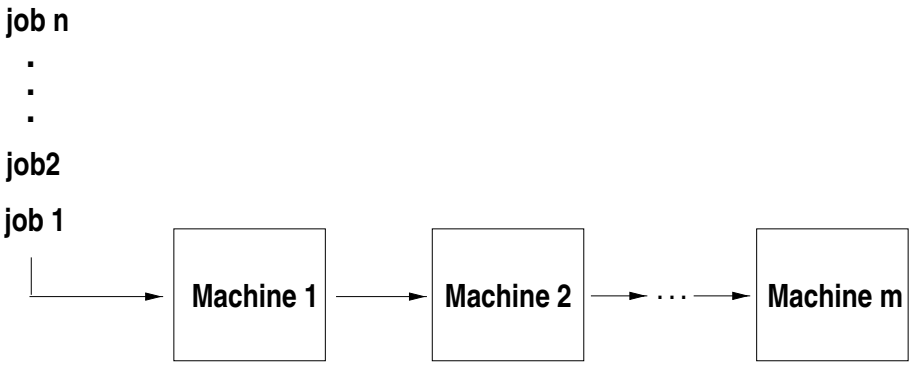


Figure 4.2: The structure of flow-shop.

An *open-shop* is similar to flow-shop, except that the order of task execution of $job\ i$ can be different from $job\ j$ [14]. In other words, the sequence of machines for each job can be different. Figure 4.3 illustrates an open-shop. By this Figure, we can see that task 11 (task 1 of job 1) is executed by machine 1, while task 21 (task 1 of job 2) is executed by machine 2. Machine 2 also executes task nm (task m of job n), while tasks $1m$ and $2m$ are executed by machine m.

Considering the $\alpha|\beta|\gamma$ fields and the makespan criterion, the flow-shop and the open-

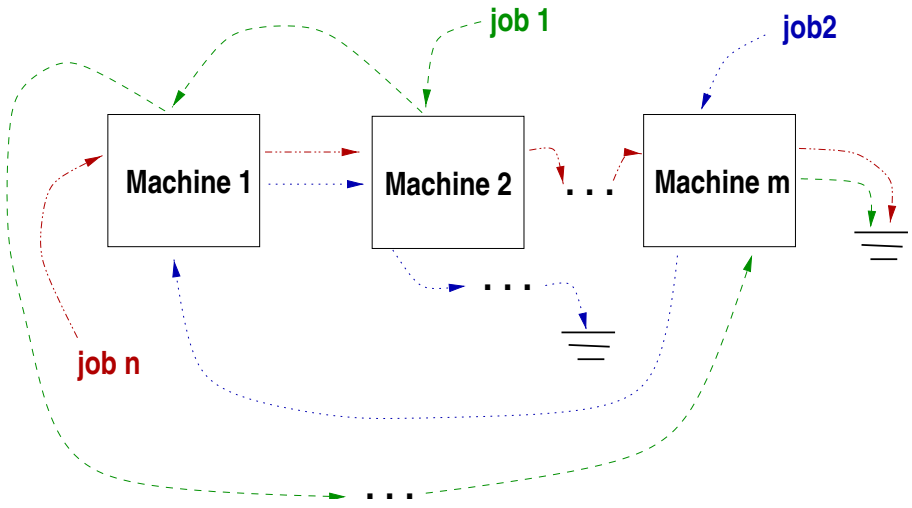


Figure 4.3: The structure of an open-shop.

shop are stated, respectively, as $F||C_{max}$ and $O||C_{max}$. The F and O in the α field indicates the existence of more than one machine and the kind of workshop. Both class of problems have NP-hard complexity [14], but there are instances that are tractable. For example, $F2||C_{max}$ is solved by Johnson's algorithm in $O(nlogn)$ time [57] and $O2||C_{max}$ is solved by Gonzalez-Sahni algorithm in $O(n)$ time [47]. Complexity of other scheduling problem instances can be found in [73].

A job-shop scheduling problem (JSP) can be stated as follows. Consider a set M of machines and a set T of tasks. The set T is partitioned. Each partition of T defines a job and the number of partitions is determined by the number of jobs. A task is executed by only one machine and is defined by the time a machine takes to execute it. Each machine executes one task at time (preemption is not allowed) and the sequence of machines for each job is predetermined. The problem is to determine the best sequence of tasks to be executed by the machines so that an optimization criterion is achieved. The optimization criterion we are interested for is the minimum *makespan* (i.e., the total time for completion of all jobs). JSP problems are NP-hard and one of the most intractable problems ever considered [41].

A JSP problem can be represented by a *disjunctive* graph $G = \{N, A, D\}$ [2, 7, 48, 7, 75], such that:

N is the set of nodes of the graph. $N = T \cup \{0, *\}$, where nodes 0 and $*$ are the dummy tasks of *start* and *finish*;

A is the set of arcs (pairs of tasks) that imposes an execution order (technological prece-

dence) among tasks of a job. For example, the pair (i, j) states that task i is the technological precedence of task j. In other words, task j can only be executed if task i has been already finished;

D is the set of disjunctive arcs. $D = \bigcup\{D_k : k \in M\}$, where D_k is the set of pairs of all tasks that are to be executed by machine $k \in M$. If $i, j \in T$ are to be executed in k, then the pairs $(i, j), (j, i) \in D_k$. $(i, j), (j, i)$ are arcs with opposite directions representing the possibility of machine k choosing task i to run before task j or vice-versa, respectively. $(i, j), (j, i)$ are named *disjunctive* because in a possible schedule just one of these arcs is chosen.

The makespan problem is to determine the start time t for each task, such that the resulting graph is acyclic and the completion of all tasks occurs in the least minimum time. This problem can be formulated as follow

$$\begin{cases} min\ t_* & & & & \\ t_j - t_i & \geq & d_i, & (i,j) \in A & (1) \\ t_i & \geq & 0, & i \in N & (2) \\ t_j - t_i \geq d_i \vee t_i - t_j & \geq & d_j, & (i,j) \in D_k, k \in M & (3) \end{cases}$$

where d_i is the fixed processing time of task i (all arcs (i, j) of G are labeled by d_i), and t_i represents the earliest possible start time of task i, that has to be determined during optimization.

The restriction (1) imposes an order of execution among tasks of each job. The restriction (2) guarantees the completion of all tasks, and (3) assures the non preemption of machine $k \in M$. Figure 4.4 illustrates a disjunctive graph representing job-shop 3×3 (3 jobs over 3 machines).

Because the JSP has been our case study along our work, in the next section we present some traditional approaches to JSP.

4.2 Traditional Approaches to Job-Shop Problem

Job-shop problem (JSP) have been studied along these 40 years. Its history is related to the challenging benchmark problem of 10×10 (10 jobs and 10 machines), proposed by Fisher and Thompson in 1963 [13]. Along this time, some few instances of JSP have been determined as being polynomial time complexity (e.g. $J2||C_{max}$), but in essence this class of problem is NP-hard complex [45, 73].

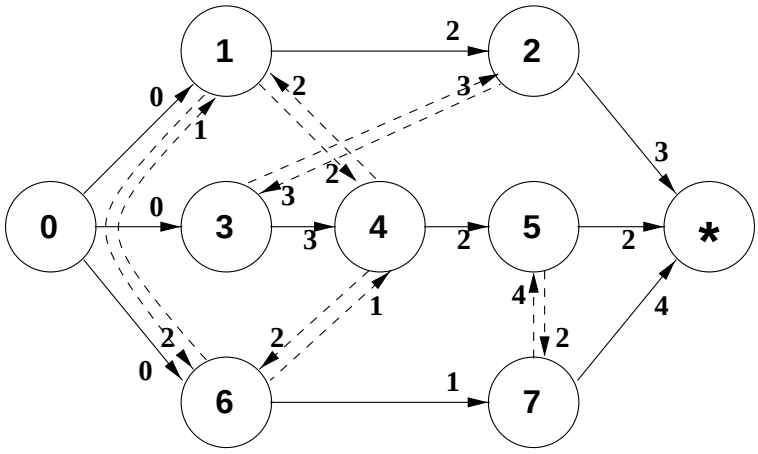


Figure 4.4: Disjunctive graph representing a job-shop 3×3 (3 jobs in 3 machines)

Researchers dealing with JSP have been applying different approaches. Some attack the problem by *relaxing* its restrictions. The relaxation makes the problem simpler and consequently easier to solve. For example, the problem $P2||C_{max}$ (the problem of non preemptive scheduling of independent tasks on two identical processors, and minimum makespan as optimization criterion) is NP-hard; but if we relax this problem by allowing preemption it becomes an $O(n)$ complexity time problem [12], where n is the number of jobs. Although when we relax problem properties we are not solving the original problem, the result can be used as an intermediate result to achieve the final result [23] or as an approximation result.

Among approximation algorithms, the *shifting bottleneck heuristic* [2, 8] is one of the most powerful algorithm [13]. The idea of this heuristic is to solve, for each machine, an one-machine problem determining an optimal schedule. The assumption is that the optimal schedule of each one-machine coincides with an optimal job-shop schedule. Other heuristics are based on local search that comprises *tabu search*, *simulated annealing* [4, 9], and *genetic algorithms* [54].

Tabu search and simulated annealing methods are similar. These methods are based on neighborhood structures and rules that guide to new structures. Applied to scheduling problems these structures relates to schedules. The difference between Tabu search and simulated annealing lies in the acceptance/rejection criterion for the new schedules. This criterion in simulated annealing is a probabilistic process, while in tabu search is a deterministic process [62].

Genetic algorithms represents the principle of natural evolution. Applied in scheduling, the individuals are sequences or schedules. The individuals are evaluated in terms of fitness

against a value returned by an objective function. The best fit individuals are combined to form a new *generation* of individuals [35, 40, 68].

The use of tabu search, simulated annealing, and genetic algorithms has advantages and disadvantages. They can be used without complete knowledge about the structural properties of the problem. They can be coded easily and usually give fairly good solutions. However, the amount of time to compute a solution tends to be relatively long in comparisons with more rigorous problem-specific approaches [73].

Another approach to scheduling problems is by exact enumerative methods as branch and bound (B&B) algorithm. The principle of this algorithm is the enumeration of all feasible solutions (schedules) by using some inference rules, which describe simple cuts and branchings. Some efficient implementations of this approach for solving JSP are [5, 15, 16, 24, 25, 65].

4.3 Conclusion

As we have already mentioned, our interest in JSP has begun after reading an article by Adams, Balas and Zawack [2]. JSP is an NP-hard problem [45] and instance problems of order 10×10 was a challenge for almost 25 years. Any way, it is very difficult to solve instances of order of few dozens of jobs.

We have presented approximative and exact approaches that have been solving JSP. The approximative approach have been successful in solving larger instances than mentioned above, but this approach does not compute an exact solution. The exact approach generally use the B&B technique with different kind of branching and cutting rules. The SMC approach is also an exact method. However there are some differences between B&B and SMC approaches.

The B&B approach, in general, scans the solution space of a JSP in a depth-first search way. It is very effective and has been successful in exactly solving important JSP instances, like 10×10 by Carlier and Pinzon implementation [23]. Very efficient B&B implementations are designed to solve some specific instances of JSP. However, if some issues of the problem change in some times we need to modify the B&B implementation.

On the other hand, the SMC approach models JSP as a graph and a breadth-first search algorithm scan all reachable states to determine the best schedule. SMC is able to deal with a state space of the order 10^{30}. Although this representative power allows SMC to

cope with non trivial JSP instances, we can not solve a 10×10 instance yet. However, since SMC is a general problem solver, it has great flexibility in modeling different kinds of scheduling problems.

Chapter 5

Symbolic Model Checking Applied to Deterministic Scheduling Problems

In the context of this work, modeling scheduling problems into Symbolic Model Checking (SMC)[1] domain means modeling these problems into Ordered Binary Decision Diagram (OBDD)[2] domain. Therefore, initially we present a mathematical modeling of job-shop scheduling problem (JSP) into OBDD. Next, we present how to model JSP into SMC by an OBDD-based SMC tool known as SMV. After that, we present the results we have obtained computing the makespan of some JSP instances using SMV. We also compare these results to those obtained by CPLEX.

5.1 Modeling JSP into OBDD Domain

Recalling job-shop scheduling problem (JSP)[3], M is the set of machines and T is the set of tasks. T is partitioned and each partition defines a job. The number of partitions is determined by the number of jobs, that is given as a parameter of the instance problem. This problem can be represented by a *disjunctive* graph $G = \{N, A, D\}$, such that: $N = T \cup \{0, *\}$ is the set of nodes of the graph; A is the set of arcs; and D is the set of disjunctive arcs. d_i is the fixed processing time of task $i \in T$, and t_i represents the earliest possible start time of task i, that has to be determined during optimization.

Now let us model JSP into OBDD domain. Consider a set of boolean variables $\mathcal{N}$, such

[1] See Section 2.2
[2] See Section 2.2.1
[3] See Section 4.1.1

that $\mathcal{N} = |N|$, and a finite set of non-negative integers $\mathcal{U}$. We can model the nodes of G (tasks) by a one-to-one function $f : N \rightarrow \mathcal{N}$ mapping each node to a boolean variable. The fixed processing time of each task can be modeled by a partial function $g : N \rightarrow \mathcal{U}$ mapping each task to a time u.

M is represented by a set of boolean variables $\mathcal{M}$, such that $|\mathcal{M}| = p|M|$, where p is the number of bits necessary to represent $g(n), \forall n \in N$. Let $p = 3$, each $k \in M$ will be represented by a 3 boolean variables, for instance, $c_k\_bit2, c_k\_bit1, c_k\_bit0$. These 3 boolean variables work as 3-bit counter that counts the execution time of a task. When a task $n \in N$ is chosen to run this counter is initially set to $g(n)$. As an example, recall task 1 of Figure 4.4 and let x be the identification of the machine that runs this task, the symbolic representation for the fixed processing time of this task is

$$\neg c_x\_bit2 \ \wedge \ c_x\_bit1 \ \wedge \ \neg c_x\_bit0.$$

This is the the boolean encoding of 2, since $g(1) = 2$.

Given $h : N \rightarrow M$, that returns the machine associated to the task $n \in N$, the execution of task n by machine m can be represented by $j : N \times M \times \mathcal{U} \rightarrow \mathcal{U}$, such that

$$j(n, m, g(n)) = \begin{cases} j(n,\ g(n) - 1) & if\ g(n) > 0 \ \wedge \ h(n) = m \\ 0 & otherwise \end{cases}$$

When $j(n, m, g(n)) = 0$, it means that n has been finished by machine m, then the boolean variable $f(n)$ must be set to *true* to indicate this situation at time u. That is the purpose of function $k : \mathcal{N} \times \mathcal{U} \rightarrow \{0, 1\}$, such that

$$k(f(n), u) = \begin{cases} 1 & if\ h(n, m, g(n) = 0 \ \wedge \ v(m, u - 1) = n \\ 0 & otherwise \end{cases}$$

where $v : M \times \mathcal{U} \rightarrow N$ returns the task that machine m is running at time u.

JSP impose, by definition, a precedence order of execution between tasks of a same job. This order of precedence is also known as *technological precedence*. The technological precedence among tasks is implemented by OBDD configuration. The configuration returns the truth value whether a task has finished or not. Recall again Figure 4.4. The task 7, for example, is finished only if $j(6, h(6), g(6)) = 0$ and $j(7, h(7), g(7)) = 0$, that is task 6 (technological precedence of task 7) and task 7 are both finished. Figure 5.1 illustrates this scenario.

This modeling was implemented by an SMC tool known as SMV, and this implementation is described in the next section.

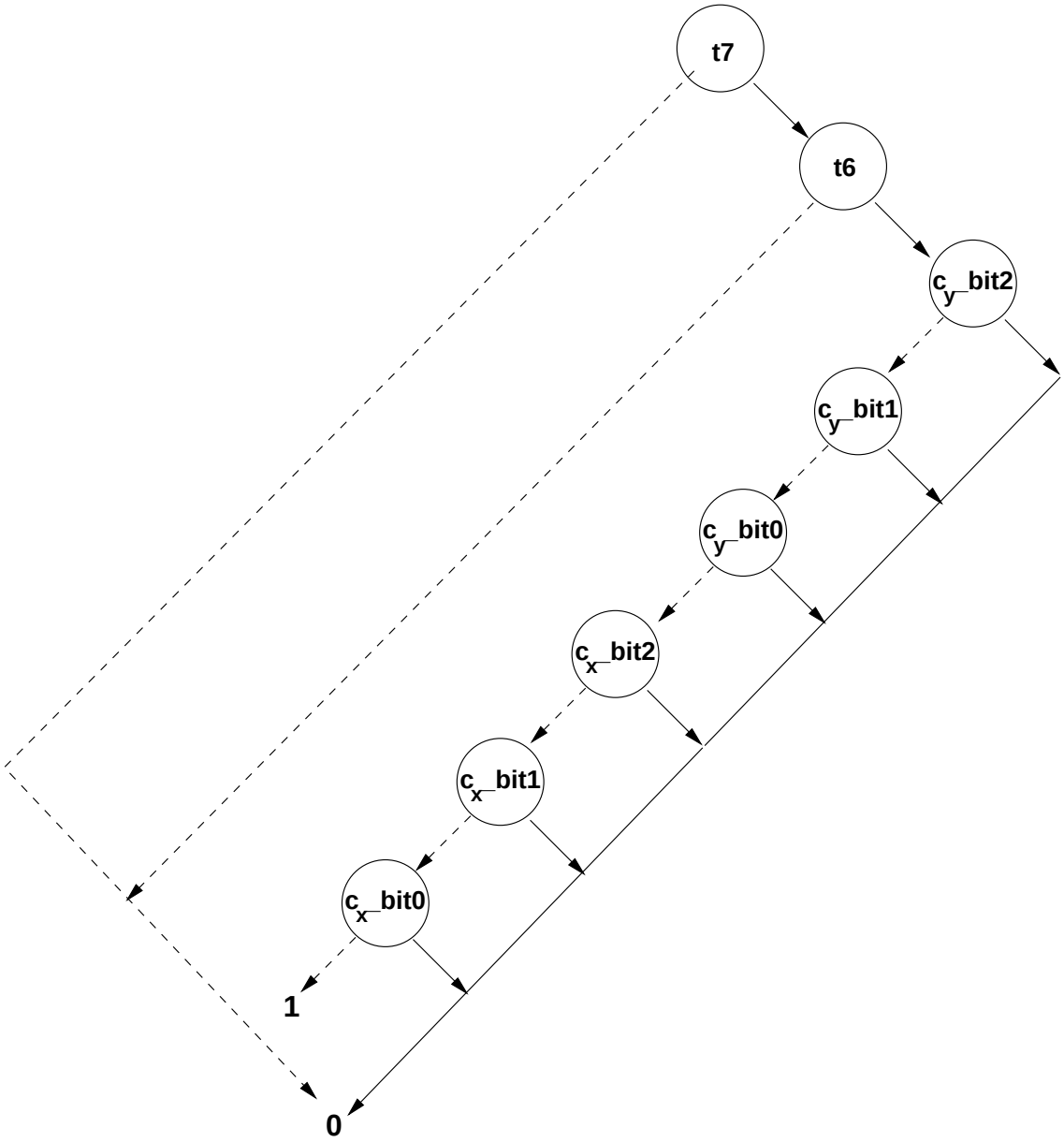


Figure 5.1: An OBDD representing the ordering for the conclusion of task 7. The boolean variables c_x and c_y represent hypothetical machines x and y, respectively. In this illustration the machines are represented by a 3-bit counters. Machine x is the one that runs task 7, and machine y runs task 6. Task 7 can be run only if task 6, technological precedence of 7, is finished. Task 6 is finished when all boolean variables representing machine y is zero. Task 7 is finished when can reach the terminal node 1, and there is only one path to it.

5.2 Modeling Job-Shop Problems into SMV

We have start modeling JSP into SMC through the tool Verus. However, by the time we were doing this work, Verus needed some improvements that could not be done immediately. For this reason, we started exploring SMV.

Consider an hypothetical manufacturing industry of automobile front panel, as follows. This industry produces three types of panel and each one needs to be handled by a specific set of machines to be produced. In this example, we have specified three machines. The machine ordering for each panel is known in advance. Table 5.1 presents the description of this problem in terms of time and technological precedence. Panel B, for example, demands 2 units of time on Machine 1, 3 units on Machine 2 and 2 units of time on Machine 3. However, Panel B must be processed by those machines in the following order: 2, 1 and 3. This example consists of a classical job-shop problem and its disjunctive graph is the one illustrated in Figure 4.4.

	Machine 1	Machine 2	Machine 3
Panel A	2	3	—
Panel B	2	3	2
Panel C	1	—	4

Demanded time by each machine.

Panel A	Machine 1	Machine 2	
Panel B	Machine 2	Machine 1	Machine 3
Panel C	Machine 1	Machine 3	

Machine execution ordering for each panel.

Table 5.1: Description of an hypothetical manufacturing industry problem. It presents the demanded time and execution ordering to produce automobile's front panels.

The sections below present an SMV program that models this problem. This program was designed with 3 modules and each of them is explained.

Module Main

Figure 5.2 presents an excerpt of the module Main of the SMV program.

```
1  MODULE main
2  VAR
3    boolean: begJS, endJS;

4    t_1: task (1, 2, m1.j);
5    t_2: task (1, 3, m2.j);

6    t_3: task (2, 3, m2.j);
7    t_4: task (2, 2, m1.j);
8    t_5: task (2, 2, m3.j);

9    t_6: task (3, 1, m1.j);
10   t_7: task (3, 4, m3.j);

11   m1: sched3 (t_1, TRUE, t_4, t_3.done, t_6, TRUE);
12   m2: sched2 (t_2, t_1.done, t_3, TRUE);
13   m3: sched2 (t_5, t_4.done, t_7, t_6.done);

14 ASSIGN
15   endJS := t_2.done & t_5.done & t_7.done;

16 COMPUTE MIN (begJS, endJS)
```

Figure 5.2: An excerpt of the module Main of the SMV program relating to the problem presented in Table 5.1.

Line 3 presents the declaration of begJS and endJS. These variables model the dummy tasks 0 and * of the disjunctive graph.

Lines 4 through 10 present the declaration of the tasks of the instance problem. The variables t_1 and t_2 relate to the panel A on machines 1 and 2, respectively. t_3, t_4 and t_5 relate to the panel B on machines 1, 2 and 3, respectively. Finally, t_6 and t_7 correspond to the panel C on machines 1 and 3, respectively.

The task variables above are of type task. task is a module that models the behavior of a task being processed by its respective machine. In SMV, when we declare a variable as being of a type module, this variable becomes an instance of this module. Each one of the tasks is characterized by some parameters: identification of its job, its processing time, and the identification of the job chosen to run. For example, when we declared variable t_3, we said that task t_3 compounds the job 2, its processing time is 3 units of time, and

the machine that runs this task is m2. The module task is explained in Section 5.2.

Lines 11 through 13 present the declaration of m1, m2, m3 that model the machines of the problem. m1 is of type sched3, and m2, m3 are both of type sched2. sched2 and sched3 are modules that model the JSP machines selecting a task to run. The difference between these two modules is the number of tasks that each "sched" module must deal with. sched3 deal with up to 3 tasks, and sched2 deal with 2 tasks. For example, m3 must run tasks t_5 and t_7, however these tasks can only run if their respective technological precedence has been finished, that is indicated by the variables t4.done and t6.done. When a task has no technological precedence the word TRUE indicates this situation. See that, in the declaration of m1 (line 11), t_1 and t_6 have no technological precedence. For a better understanding about "sched" modules, we explain module Sched2 in more detail later.

The **ASSIGN** declaration, in line 14, corresponds to the dynamic of the main module. When all tasks have been finished, endJS is set to TRUE, indicating the end of the job-shop. Since the model implements the technological precedence of the job-shop, it is only necessary to verify the last task of each job, as stated at line 15.

Line 16 relates to the computation of the makespan for the model. The **compute** statement calls the **MIN** procedure, that compute the minimum path between "a state" whose variable begJS is true to another "state" whose variable endJS is true.

Module Task

This module is responsible for modeling a task of a JSP, and has three parameters: me corresponding to the id of the job that contains the task; time is the fixed processing time of the task; and sj corresponds to the id of the job selected to run by the machine. A task only runs if it has been chosen by its respective machine. This choice is indicated by the parameter sj. Figure 5.3 illustrates an excerpt of module task.

Line 3 declares a variable i, that corresponds to a clock of the task. Initially, i is set to processing time of the task (line 5). If the content of me equals to sj than the task starts running (line 7), otherwise its initial value is unchanged (line 8). A task runs until its completion, that is clock equals to zero. When the clock is set to zero, the variable **done**[4] is assigned to true (line 11).

[4] **done** is not indeed a variable. It is a SMV feature to declare frequent expressions.

```
1  MODULE task (me, time, sj)
2  VAR
3    i : 0..4;
4  ASSIGN
5    init(i) := time;
6    next(i) := case
7                 (me = sj) & (i > 0): i - 1;
8                 1 : i;
9                 esac;
10 DEFINE
11   done := (i = 0);
```

Figure 5.3: Excerpt of our SMV program relating to a task of a JSP.

Module Sched2

In JSP, each job is divided into tasks. Each of them must be executed by a specific machine. Therefore, a task means a certain job running on a certain machine. Along this section, we use the terms "task" and "job" (in the context of a machine) indistinctly.

When a machine is declared, see lines 11 through 13 of Figure 5.2, the tasks that must run on the machine are expressed. A "sched" module models the machines of the JSP selecting one of these tasks to run. The selection is done according the technological precedence between the tasks.

Figure 5.4 presents the module sched2 that models a machine that must select one task among two tasks to be processed. Module sched3 is similar to sched2, except that the former is designed to deal with 3 tasks.

Line 3 presents the variable that indicates the identity of the job chosen to be run. In Line 4, the variable j_aux models the identity of the job in the previous state. The function of j_aux is to save the identity of the job being run until its completion.

The choice for a job to be run on a machine occurs according the technological precedence. A task is candidate to be executed only if all the following conditions are satisfied: (a) the task is not yet executed; (b) the task is not being executed; and (c) the technological precedence of the task is over. If more than one task accomplishes all the conditions, then one of them is randomly selected. Lines 8 through 16 model the behavior described above. Items "a" and "b" are indicated by the formal parameters t1 and t2. Recall that

```
1  MODULE sched2 (t1, pt1, t2, pt2)
2  VAR
3    j : 0..3;
4    j_aux: 0..3;

5  ASSIGN
6    init (j_aux) := 0;
7    next (j_aux) := j;

8    j := case
9            !t1.done & !t1.running & pt1 &
10           !t2.done & !t2.running & pt2 : t1_id, t2_id;

11           (t1.done | !pt1) &
12           !t2.done & !t2.running & pt2 : t2_id;

13           (t2.done | !pt2) &
14           !t1.done & !t1.running & pt1 : t1_id;

15           1 : j_aux;
16   esac;

17 DEFINE
18   t1_id := t1.id;
19   t2_id := t2.id;
```

Figure 5.4: Excerpt of our SMV program relating to a machine that chooses one task among two of them to run.

the correspondent actual parameters are tasks, therefore the variables done and running come from module task. Item "c" is modeled by the formal parameter pt1 and pt2, whose actual parameters are the predecessor of the tasks t1 and t2, respectively. The exclamation sign (!) relates to the negation sign.

The lines 18 and 19 are only necessary because of variable scope restriction of SMV.

5.3 Computing Makespan

An SMC model represents all possible states that a system can be in a certain moment. Let this system be a JSP, then all possible sequences are represented. Computing the makespan of this problem relates to find a feasible sequence among all states of this model. We have been using the MIN algorithm, described in section 2.2.3, to find a feasible one.

The MIN algorithm computes the minimum path between two states. In the case of our SMV model, let s_i be a state in which begJS is true, and s_f be a state in which endJS is true. Determining the makespan corresponds to finding a minimum path between states s_i and s_f. The MIN algorithm computes this path in a breadth-search way, therefore it takes into consideration only the states that can direct toward a solution. Since endJS is true only if all jobs have been finished, then we can guaranteed that

- if does exist a path between s_i and s_f, MIN return the minimum path;
- the minimum path corresponds to a sequence that corresponds to the minimum makespan.

Table 5.2 presents the results obtained by SMV in determining the minimum makespan of some JSP and FSP (flow-shop) instance problems. These problems have been run on a Pentium III 600 MHz PC-Linux with 256 MB of RAM memory. The JSP instance 3×2 was taken from [7], the 5×4 was taken from [2], and the 6×6 was based on [43]. The other JSP instances were derived from these original problems. The FSP instances correspond to the serialization of the corresponding JSP.

The column **Mks** presents the minimum makespan computed by SMV. The value of all computed makespan values have been checked by another tool [59] and all of them are correctly computed. The column **Time** refers to the processing time (in seconds) taken by SMV to compute the makespan. The columns **OBDD nodes** and **Megabytes** correspond

	Job-Shop				Flow-Shop			
	Mks	Time	OBDD nodes	Megabytes	Mks	Time	OBDD nodes	Megabytes
3×2	26	0.03	9,173	1.31	31	0.05	10,085	1.31
3×3	10	0.09	14,182	1.31	12	0.03	10,080	1.31
3×4	22	0.08	10,192	1.31	23	0.09	10,845	1.31
4×3	18	0.23	37,383	1.70	22	0.20	18,248	1.38
4×4	17	0.49	59,336	2.03	17	0.26	27,262	1.51
5×4	13	0.17	20,027	1.38	13	0.15	16,805	1.38
5×5	18	7.03	198,238	4.39	18	1.76	107,605	2.95
6×6	09	923.23	4,164,265	67.8	11	0.68	56,766	2.1

Table 5.2: Consumption of OBDD nodes, time and memory by SMV when executing some flow-shop and job-shop problems. The column "time" is in seconds, except in the 6×6 job-shop cell, that is in minutes.

to the number of nodes generated and the amount of RAM memory consumed, during the computation, respectively.

We can see that SMV demands more time to solve JSP than FSP as we increase the complexity of the instances of these problems. In certain instances, such as 5×5 and 6×6, the difference between JSP and FSP in terms of OBDD nodes and consequently memory space are very significant. This occurs because the size of OBDD for JSP is greater than for FSP, and the efficiency of OBDD-SMC approach is related to the size of OBDD.

The size of OBDD is related to the number of variables of the problem being modeled and the ordering of these variables along the OBDD. Depending the ordering of the variables, the size of OBDD can vary from linear to exponential to the number of variables. Since finding a good order for variables over a OBDD is a NP-complete problem [17], good ordering depends of heuristics.

One very useful heuristic is: "variables of the model that are closely related to each other must be as near as possible over OBDD". However, it is very hard to implement this heuristic in JSP instances, as we explain as follows. Considering to our model (see Section 5.2), the variables representing tasks of the same job should be close to each other along the OBDD. However, at the same time, these variables should also be close to the variables representing tasks that correspond to their respective technological precedence. Since the technological precedence in a JSP generally involves tasks of different jobs, the heuristic mentioned above generally is not feasible.

We have also compared the time performance of SMV in computing makespan against

CPLEX [56], another generic problem solver. CPLEX is a commercial tool for solving a wide range of optimization problems, whose modeling language is a mathematical language. The CPLEX model template relating the JSP and FSP instances are presented in Appendix B.

The Table 5.3 presents the time comparison mentioned above. The columns **SMV** and **CPLEX** present the time in seconds taken by each of these tools in computing the makespan of the presented JSPs and FSPs. Recalling that CPLEX is a commercial tool, we can say that SMV has presented processing times quite similar to CPLEX, except in the 6×6 JSP instance.

Job-Shop	SMV	CPLEX
3×2	0.03	0.03
3×3	0.09	0.02
3×4	0.08	0.04
4×3	0.23	0.01
4×4	0.49	0.12
5×4	0.17	0.08
5×5	7.03	0.62
6×6	923.23	4.44

Flow-Shop	SMV	CPLEX
3×2	0.05	0.04
3×3	0.03	0.07
3×4	0.09	0.03
4×3	0.20	0.12
4×4	0.26	0.19
5×4	0.15	0.06
5×5	1.76	0.50
6×6	0.68	105.12

Table 5.3: Time comparison (in seconds) between SMV and CPLEX in computing the makespan of some JSPs and FSPs.

CPLEX was significantly faster than SMV when processing the 6×6 JSP instance. However, in the 6×6 FSP instance the opposite occurred: SMV was very much faster than CPLEX. This result was a combination of two factors:

- the size of the OBDD;
- the demanded time of the tasks of the 6×6 instance.

The first item relates to the fact that the size of OBDD for FSPs generally is much minor than the JSP counterparts. Consequently, OBDD-SMC tends to be faster over FSPs than JSPs. The last item relates to the fact that the B&B technique could not cut-off paths of the solution tree when solving FSP.

5.4 Conclusion

Traditionally, exact solutions for deterministic scheduling problems have been obtained by using branch-and-bound (B&B) technique. The main approach of B&B technique is to find a feasible solution over a solution tree by avoiding to explore some paths that certainly can not lead to a feasible one. Indeed avoiding certain paths implies in gaining of time/space performance.

On the other hand, formal verification techniques (FMT) must explore all paths (states) of the model being verified to assert properties about the model. The exploration of all states is a necessity because we can only assure a certain property if we explore all states of the model.

We have been using an FMT based on OBDD-SMC to give exact solutions for deterministic scheduling problems. OBDD-SMC is very efficient in modeling and verifying complex systems with a large number (10^{30}) of states. The efficiency of the SMC algorithms and of the OBDD representation have led us to model complex JSP/FSP problems.

We have modeled some instances of JSP/FSP and computed their respective makespan using SMV (an OBDD-SMC tool). The SMV models were automatically generated by a tool especially created for this thesis (see Appendix C). The configuration of the modeled problems are presented on Appendix A, and the results of the makespan computation were presented in Table 5.2.

Being OBDD-SMC a generic problem solver, we have compared the processing time of SMV against CPLEX, another generic problem solver. The CPLEX template for JSP/FSP problems can be seen in Appendix B, and the results of this comparison were presented in Table 5.3. Considering that CPLEX is a commercial software product and has been receiving a lot of optimization since it was launched, the time performance of OBDD-SMC is very much comparable to CPLEX, mainly if we consider FSPs. These results points out that FSP is the class of scheduling problem that is more adherent to the OBDD-SMC approach.

Besides, considering the 6×6 FSP instance, SMV was very much faster than CPLEX. This result occurred because the B&B technique of CPLEX had great difficulty to prove the optimality of the solution. This difficulty was provided by the configuration of this instance, whose demanded time of all tasks is the same value. That is, all tasks demand the same amount of time. In this scenario, B&B technique has great difficulty to cut-off some paths of the solution tree. On the other hand, SMV was very fast in solving this

problem.

The time response for computing the makespan of 6×6 FSP instance enhances our expectation that indeed FSP is the niche of application of OBDD-SMC approach. We need now to model more complex FSP instances to evaluate the OBDD-SMC approach under more complex instances. In addition, further research is needed to characterize the FSP instances more accurately in terms of the time of the tasks and the related effect over B&B technique and OBDD-SMC itself.

Chapter 6

New Model Checking Algorithms

During the formal verification process, we spend much of the time doing quantitative analysis about the system being verified. Generally we are involved in analyzing paths of computation, looking for minimum or maximum paths in this model. Usually these paths carry very important information about the systems being verified.

In these chapter we present new minimum path algorithms applied to scheduling scenarios. Although these algorithm were designed with scheduling problems in mind, they can be used in any other scenario in which we have to reason about the distance between two states. The algorithms presented in the following subsections use the data described below:

- I - CTL formula representing the set of initial states;
- F - CTL formula representing the set of final states;
- C - CTL formula representing a condition that must be satisfied along the path between I through F;
- i - path size;
- R - current reached set of states;
- R' - set of states reached from R;
- $T(R)$ - function that computes the set of states that are reachable from R in one forward transition. It corresponds to the transition relation ρ of the Kripke model.

6.1 MINCOND Algorithm

MINCOND returns the size of the shortest path (i.e. number of edges) between I and F, such that condition C holds somewhere along this path. The number of times C holds in the path does not matter. If there is such path MINCOND returns its size, otherwise it returns *infinity*. The illustration presented in Figure 6.1 show a case in which F is reached in 4 steps.

```
1  proc MINCOND (I, F, C) {
2      i = 0;
3      R = ∅; R' = I;
4      P = ∅; P' = I ∩ C;
5      loop
6          if ((P' ∩ F ≠ ∅) ∨ (R' == R))
7              exit loop;
8          R = R';
9          P = P';
10         i = i + 1;
11         R' = T(R) ∪ R;
12         P' = T((R ∩ C) ∪ P);
13     end loop
14     if (P' ∩ F ≠ ∅) return i;
15     else return ∞;
16 }
```

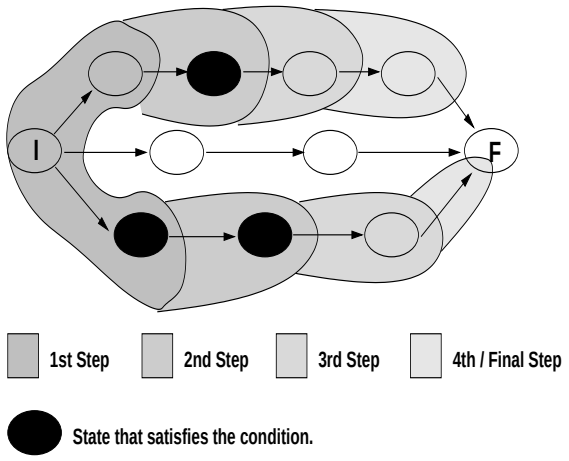


Figure 6.1: The MINCOND algorithm and an illustration of its behavior.

The states in which C holds are P and P', such that P' consists of states that are reachable from P in one forward transition.

MINCOND starts from I (lines 3-4). P and P' are set of states that form paths. At least in one of these states C holds. At each round of the loop (lines 5-13), MINCOND computes the set of states that can be reached in one forward transition from R (line 11). MINCOND also computes the set of states that satisfies condition C (line 12). The loop is finished if there is a path in P' that reaches F (the final states) or if there is not exist such a path (line 6). In this case, MINCOND returns ∞, otherwise it returns the size of the path (line 14).

We will prove the correctness of the algorithm by showing that the MINCOND terminates and a certain loop invariants holds. We will use in the proof the following notation:

- $S(i)$ is the set of all states that can be reached from I in i steps or less;
- $Cond(i) = S(i) \cap C$, i.e. the set of all states in $S(i)$ that satisfies C; and
- $Path\_C(i)$ corresponds to the set of states that form a path in which C holds, in i steps or less, such that

$$Path\_C(i) = \begin{cases} I \cap C & \text{if } i = 0 \\ T((Cond(i-1) \vee Path\_C(i-1)) & \text{if } i > 0 \end{cases}$$

The loop invariant is the following

$$Inv(i) : (R' = S(i) \ \wedge \ P' = Path\_C(i))$$

Lemma 6.1 *The invariant holds in the loop provided loop test holds.*

Proof: When $i = 0$, $Inv(0)$ holds before the loop, by the initialization of MINCOND (lines 2-4). When $i > 0$, $Inv(i)$ holds as follows. At each round of loop,

- i is incremented by one,
- R' is set to $T(R)$ plus all states in R (line 11). $R = S(i-1)$, hence $R' = T(S(i-1)) \cup S(i-1)$. By definition of S and function T, we have that $R' = S(i)$;
- P' is set to all states reached from R that satisfies C plus the successors of P (line12). $R \cap C = Cond(i-1)$ and $P = Path\_C(i-1)$, consequently, by definition, $P' = Path\_C(i)$.

Lemma 6.2 *MINCOND terminates.*

Proof: MINCOND terminates when it exits the loop. There are two conditions for MINCOND exiting the loop: (a) it does exist a path between I and F so that C holds in some state of this path; or (b) does not exist such path. In case "a", there is a state $s \in P'$ such that $s \in F$, then MINCOND exits the loop by the first half loop test and returns i. In case "b", despite of (possible) existing paths between I and F, none of them have a state that satisfies C. Hence, MINCOND exits the loop by the second half loop test, $R' == R$. By construction, $R' \supseteq R$, since the number of states is finite, there is a finite sequence of distinct values for R'. If condition "a" never holds, condition "b" will eventually hold. Consequently, MINCOND always exits the loop.

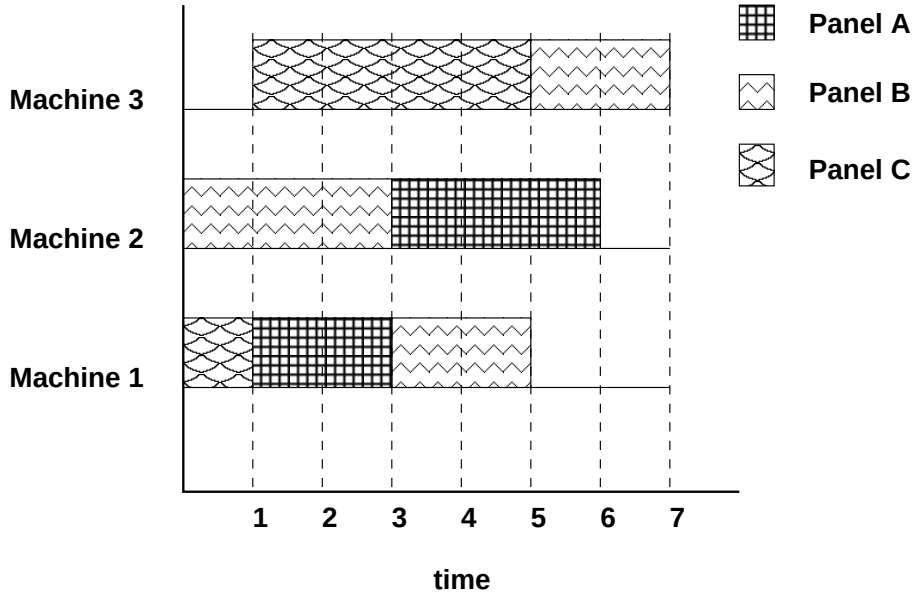


Figure 6.2: Gantt chart for the 3×3 job-shop relating an hypothetical manufacturing industry of automobile front panels described in Table 5.1. This illustration presents a sequence of tasks that results in a minimum makespan.

Lemma 6.3 *MINCOND returns the size of the shortest path between I and F so that C holds in some state of this path.*

Proof: Let s be the size of the shortest path of the shortest path between I and F so that t C holds in some state of this path. Let also m be the size of the shortest path in which C holds returned by MINCOND. If $m > s$ then MINCOND has had a deviation in the path through F. However, by algorithm construction, at each step we get all the successor states reached from the current one. Consequently, there is no way to get this deviation. If $m < s$ then MINCOND has exited the loop by the first half test and m is the size of the shortest path. This is a contradiction since we have asserted that the size of the shortest path is s. Consequently, $m = s$.

6.1.1 MINCOND Application

Let us consider the manufacturing industry of automobile front panels presented in Table 5.1. Figure 6.2 has presented a sequencing for the tasks so that the makespan is minimum. Based on this sequencing the scheduling manager plans a working day in the plant.

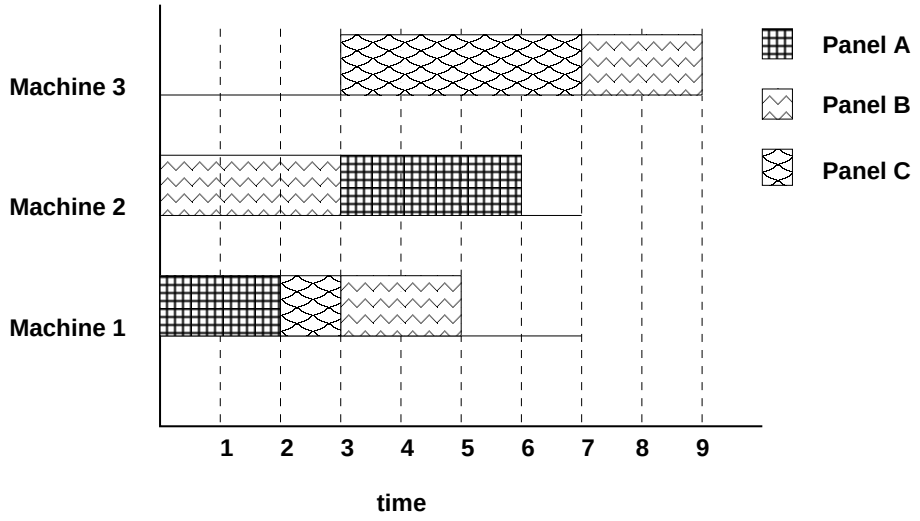


Figure 6.3: Gantt chart for the 3×3 job-shop relating an hypothetical manufacturing industry of automobile front panels. This illustration presents a sequence of tasks in which Panel A finishes as soon as possible but minimizes the late of the other tasks.

However, unexpected events can occur. For example, the production manager can receives an order of the board directory that he must deliver the Panel A as soon as possible, however minimizing the late of the other tasks. This order brakes its original sequencing since according it Panel A only finishes after Panel C has been finished. Hence, he needs to make some changes in this sequencing. It is not a simple change when many tasks are involved. To cope with this problem, the production manager only needs to specify a new property and runs SMV again. The property is

```
COMPUTE MINCOND [begJS, endJS, (t_2.done & !t_7.done)]
```

In other words, SMV computes the size of the shortest path between the beginning of all tasks (begJS) and the end of all tasks (endJS), so that along this path there is at least one state that satisfy the condition t_2.done & !t_7.done. The new makespan for this situation is 9.

The condition t_2.done & !t_7.done was taken by analyzing the original sequencing. We have taken the last task of Panel A and assured that it is finished before the first task of Panel C. An illustration of this new sequencing of task is in the Figure 6.3.

6.2 MINCOND-P Algorithm

MINCOND-P returns the size of the shortest path (i.e. number of edges) between I and F, such that condition C holds in ALL states proceeding F. MINCOND-P starts from I. At each step, it computes the set of states that can be reached in one forward transition from the current set of states. Only states that satisfy C are considered. This process continues until it reaches F or it detects that there is no path to F. In the former case, then MINCOND returns the size of the path, otherwise it returns infinity.

Figure 6.4 presents the algorithm and an illustration of its behavior. Lines 2 and 3 consist of initialization of variables. Since all states preceding F must satisfy C, then we remove all states that do not satisfy C from I (line 2). At each round of the loop (lines 4 through 11), MINCOND-P computes the set of states that are reached in one forward transition from R (line 9). We do not consider any state that do not satisfy C, however we take care to not discard any possible state in F that can lead to a solution. MINCOND-P exits the loop (lines 5-6) in one of the following situations:
(a) $R' = \emptyset$ – there is no reachable state that satisfies C;
(b) $R' == R$ – there is no path from I to F in which all states satisfy C; or
(c) $R' \cap F \neq \emptyset$ – F has been reached.
In case "a" and "b", MINCOND-P returns ∞ (infinity), in case "c" MINCOND-P returns i, as the size of the shortest path.

We will prove the correctness of the algorithm by showing that the MINCOND-P terminates and a certain loop invariant holds. We will use in the proof the following notation:

- $S(i)$ is the set of all states that can be reached from I in i steps or less;
- $Cond(i) = S(i) \cap C$, i.e. the set of all states in $S(i)$ that satisfies C.

The loop invariant is

$$Inv(i) : R' = Cond(i)$$

Lemma 6.4 *The invariant holds in the loop provided the loop test holds.*

Proof: When $i = 0$, $Inv(0)$ holds since R' is set to all states in I that satisfies C (line 2). When $i > 0$, $Inv(i)$ holds as follows. $R = Cond(i-1)$, by line 7. R' is set to all successor states of R that satisfies C (line 9), plus all all states in R (line 10). Consequently, $R' = Cond(i-1+1) = Cond(i)$.

```
1  proc MINCOND-P (I, F, C) {
2      R' = I ∩ C;
3      i = 0; R = ∅;
4      loop
5          if ((R' == ∅) ∨ (R' == R)∨
               (R' ∩ F ≠ ∅))
6              exit loop;
7          R = R';
8          i = i + 1;
9          R' = T(R) ∩ (C ∪ F);
10         R' = R' ∪ R;
11     end loop
12     if (R' ∩ F ≠ ∅) return i;
13     else return ∞;
14 }
```

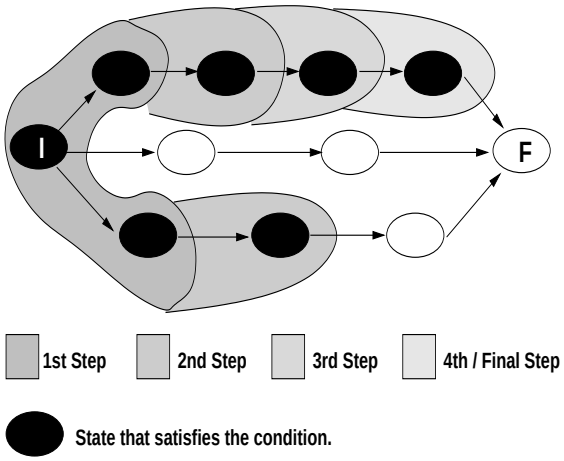


Figure 6.4: The MINCOND-P algorithm and an illustration of its behavior. This algorithm searches for paths between states I and F, in which all states along the path satisfy condition C. If MINCOND-P finds such paths, it returns the size of the shortest one. Otherwise MINCOND-P returns *infinity*. The illustration presents a case in which F is reached in 4 steps.

Lemma 6.5 *MINCOND-P terminates.*

Proof: MINCOND-P terminates because eventually the loop test will be satisfied, as following. For $i \geq 0$,
(a) if $Cond(i) = \emptyset$ then MINCOND-P terminates by holding $R' == \emptyset$ test;
(b)if a state in $Cond(i)$ satisfies F then MINCOND-P terminates by holding $R' \cap F \neq \emptyset$, and returns i;
(c) otherwise, we have the following. By construction, $R' \supseteq R$ (lines 9-10), since the number of states is finite, there is a finite sequence of distinct values for R', then the loop test $R' == R$ will be satisfied.

Lemma 6.6 *MINCOND-P returns the size of the shortest path between I and F, such that conditon C holds in all states preceding F along this path.*

Proof: It is guaranteed by the loop invariant that MINCOND-P only considers states that satisfy C. Consequently, if MINCOND-P reaches F, all states preceeding F satisfy C. Let s be the size of the shortest path of this kind, and p the size of the shortest path returned by MINCOND-P. If $p > s$ then MINCOND-P has had a deviation in the path through F. However, by algorithm construction, at each step we get all the successor states reached from the current one. Consequently, there is no way to get this deviation. If $p < s$ then MINCOND-P has exited the loop test ($R' \cap F \neq \emptyset$) and m is the size of the shortest path. However, this is a contradiction since we have asserted that the size of the shortest path is s. Consequently, $p = s$.

6.2.1 MINCOND-P application

Consider the hypothetical manufacturing industry of automobile front panels of Section 5.2. Now suppose the resource power of machine 3 has failed. The mechanic in chief has to serialize machines 1 and 3 so that they can share the same resource power. Under this new scenario, machines 1 and 3 can not run in parallel anymore. The production manager needs to compute a new sequence to the jobs. That is here that SMC shows its versatility. It can face this unpredictable event by specifying a new model property or changind the model. In this specific case we can run MINCOND by expressing the property below in the NuSMV main module

```
COMPUTE MINCOND_P [begJS, endJS, (!m1_running | !m3_running)]
```

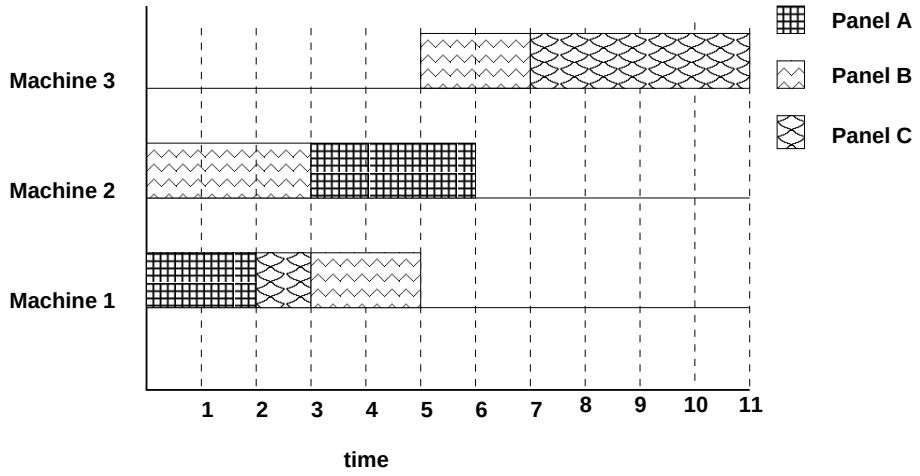


Figure 6.5: Gantt chart for the 3×3 job-shop problem example, considering that machines 1 and 3 can not run in parallel anymore.

where m1_running and m3_running represent boolean variables of the model. These variables indicate whether machines m1 and m3 are running or not. The sign "!" represents the unary negation operator. The proposition (!m1_running | !m3_running) corresponds to the condition that must be held during the minimum makespan computation. This condition relates to the situation that m1 and m3 can not run in parallel anymore. Figure 6.2.1 presents an illustration of a new sequence that MINCOND is able to compute given the above restriction. In this case the new minimum makespan is 11.

6.3 MINCOUNT-P Algorithm

Consider $\pi_1, \pi_2, \ldots, \pi_p$ being all paths between states I and F. MINCOUNT [22] searches for states that satisfies a condition C in π_i, $1 < i < p$. If there is no path between these states, MINCOUNT returns the special value NOPATH. Otherwise, it returns $m = min(m_1, m_2, \ldots, m_p)$ such that m_i is the number of states satisfying C in the path π_i.

However, in some cases, we need to present the path related to m. MINCOUNT-P extends MINCOUNT in this way. In other words, MINCOUNT-P returns a path corresponding to m. Figure 6.6 presents MINCOUNT-P algorithm and its behavior. It has two phases. The first one is the running of MINCOUNT algorithm just as proposed by Campos (line 2). If MINCOUNT returns NOPATH, then MINCOUNT-P returns ∞ (infinity). If MINCOUNT returns $m = 0$, no state satisfying C was found, and MINCOUNT_P returns the special value NOPATH. Otherwise, second phase (lines 5-12) begins.

```
1  proc MINCOUNT-P (I,C,F) {
2      mincountResult = MINCOUNT(I, C, F);
3      if mincountResult = NOPATH return ∞;
4      if mincountResult = 0 return NOPATH;
5      M = M ∩ ¬F;
6      R = ∅; R' = F; k = 0;
7      loop
8          R = R';
9          R' = T^{-1}(R) ∩ M;
10         printSet (R');
11         if (R' ⋂ I ≠ ∅) exit loop;
12     end loop
13     return R'
14 }
```

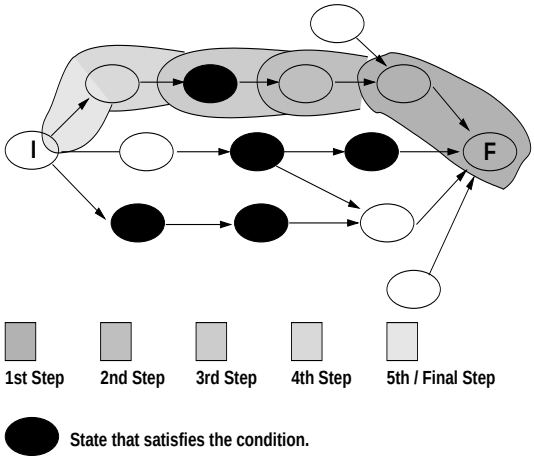


Figure 6.6: MINCOUNT-P algorithm returns a path between state I and F, such that in this path m states satisfies C. m is the least number of states satisfying C over all paths leading from I to F. If there is no path between I and F, MINCOUNT-P returns the special value NOPATH. If $m = 0$, then MINCOUNT-P returns ∞ (infinity). The illustration shows the behavior of the algorithm. It walks backward reaching states over paths between I and F. In this illustration, we can see that I is reached in 5 steps.

Let M be the set of states reached from I through F in the first phase, when $m > 0$. We remove F from M (line 5), as an initialization step of the second phase for performance purpose. MINCOUNT-P walks backward over M (line 9). Formally, $T^{-1}(X) = \{s \mid \rho(s, s')\ for\ some\ s' \in S\}$, where X is a set of states. At each backward step it reaches the set of states from the previous one. This set is intersected with M, because we are not interested in all predecessor states from the previous one, but only those states that had been reached in the first phase. The **printSet** function prints all states in R' (line 10). Since MINCOUNT (first phase) returned $m > 0$, it is guaranteed that MINCOUNT-P exits the loop (line 11), when it reaches a state satisfying I.

We will prove the correctness of the algorithm by showing that the MINCOUNT-P terminates and the following loop invariants hold:

- $R = S(k)$;
- $R' = S(k + 1)$.

where $S(k)$ is the set of all states in M that can be reached in a backward walk from F in

k steps or less.

Lemma 6.7 *MINCOUNT-P satisfies the loop invariants.*

Proof: Considering MINCOUNT-P moves backward and $\emptyset$ as the set of states previous to F, when $k = 0$, $S = \emptyset$ and $S' = F$, then the invariants hold. When $k > 0$, $R = R'$ (line 8) what is equivalent to $S(k + 1)$, then the invariant $R = S(k)$ holds for new values of k. Also, R' is set to its predecessors in M what is equivalent, by construction, to $R' = S(k+2)$. Then, the invariant $R' = S(k + 1)$ holds for new values of k too.

Lemma 6.8 *MINCOUNT-P terminates.*

Proof: In the first phase, Campos proved that MINCOUNT terminates [22]. Then, MINCOUNT-P exits by line 3 or 4, otherwise the second phase begins. It is guaranteed by the first phase that M has at least one path leading from I to F, such that there is m states satisfying C [22]. MINCOUNT-P starts by the set of states in F. At each loop round, MINCOUNT-P moves backward reaching all states in M that lead to the current set of states. Inevitably, MINCOUNT-P will reach some state in I and the loop test will be satisfied.

Lemma 6.9 *MINCOUNT-P returns a path between I and F such that this path contains the minimum number of states that satisfy C.*

Proof: Let m be the minimum number of states satisfying C along a path between I and F. All the states that compound this path are in M. By contruction (by MINCOUNT), the only paths from I to F that does exist in M are those relating to m. Consequently, finding this path is just a case of choosing a start point and walking through the opposite point direction. Inevitably this point will be reached.

6.3.1 MINCOUNT-P application

We will present an application to MINCOUNT-P somewhat out of the context of the JSP. Remembering Section 4.1, scheduling problems can be stated by $\alpha|\beta|\gamma$ fields. In scenarios where field α is characterized by P, Q or R, the jobs can be processed by any one

of the machines. If we want to determine a sequence for the jobs over the machines, such that the makespan is minimum and some specific machine is used as less as possible, then the MINCOUNT-P does this task. The right side of Figure 6.6 can illustrate this case, being the black nodes the machine we want to use the least minimum of times, and the nodes I and F being the initial and the final of the jobs. MINCOUNT-P is able to cope with this problem and present the desired route, if such route exists between I and F.

6.4 Conclusion

All the algorithms presented in this chapter enhance the power of SMC in solving scheduling problems. The algorithm MINCOUNT-P, for example, is very useful in a plant, when we want to minimize the use of some machines. In the case of this algorithm, we can obtain similar result with some mathematical programming tool, such as CPLEX. However, the mathematical model would be multi-criteria and the modeling would be more complex than the ones we presented in Appendix B.

Chapter 7

Integration between Production and Operational Planning

Planning is related to making decisions based on information. Nowadays, important information about production process comes from computer systems. Over the years, computer systems have been used to support different phases of production process. They improve the production process in many aspects, one of them relates to the information they generate based on datum they collect.

Many techniques, such as PERT/CPM (Program Evaluation Review Technique / Critical Path Method), MRP II (Manufacturing Resource Planning), have been used to provide information to support planning and controlling of the production. However, these techniques in general, deal with tactical and operational decisions in isolation. That is, information of both levels are not integrated. Our interest in this work relates to the ones which is concerned with this integration [60].

In this chapter we present how SMC can deal with planning problems. At first, we give an introduction to the problem of integrating tactical and operational plannings. The next sections describe the multi-period scheduling problem and how we have modeled this problem in SMV. Finaly, we present the result we have obtained solving an hipothetical problem and the conclusion about this study.

7.1 Integrating Information between Tactical and Operational Levels

Planning consists of one of the most important component in the economic life of enterprises. Considering a manufacturing plant, usually the planning process is hierarchically organized (from top to bottom) into three levels: strategic, tactical, and operational [49]. One planning level determines the targets to be achieved by the immediately level below.

Strategic level is the higher level of planning. It concerns with establishing long term goals to the enterprise. In general it involves the establishment of enterprise politics, types of products or services to be offered, construction and location of new plants or branch offices. The source of information at this level, in general, is external to the enterprise.

Tactical level is the middle level of planning. It concerns with production planning to accomplish the clients demands/contracts. In manufacturing industry, it relates to planning the production over periods of time (weeks or months), minimizing the production costs, for example. It involves establishing mid term goals, such as how much to produce in each period of time. The source of information at this level are both external and internal to the enterprise.

Operational level is the lower level of planning. It concerns with accomplishing tactical goals by means of scheduling. The planning at this level involves plans of short term goals and its source of information is internal to the enterprise.

Classical models for production (tactical) and scheduling (operational) plannings concerns with different types of information. The former works with more general information such as quantity of products to be produced over periods of time (weeks or months). The latter works with more detailed information, such as jobs, machines, running time of jobs over machines. Tactical managers need to know whether their production planning is feasible at the operational level. In other words, if there exist a scheduling of production (operational planning) such that the production planning is feasible. Due to the lack of operational informations at tactical level, it is not always the case.

When a production request is sent to the operational level, usually, this request was made without taking into consideration certain operational aspects such as, for example, idle time of the machines [36]. The idle time appears in operational level in consequence of the technological precedence of each job. For instance, a certain machine remains idle waiting for the conclusion of a job by another machine. Figure 7.1 illustrates this case. This figure presents a JSP 3×3 and its makespan. The JSP is represented by a disjunctive

graph. The makespan is presented as an optimal sequence of jobs over machines by a Gantt chart. We can see by the Gantt chart that there is an idle time in machine 3 at time 1. This idle time corresponds to the time that this machine waits for machine 1 finishing Panel C.

At first sight, one approach to solve the gap between tactical and operational levels of decision is migrating all information of operational level to the tactical level. However this approach has two drawbacks: it eliminates the decisions from operational level (concentrating all of them into the tactical level); and production planning becomes much more complex or even impossible to model [36].

One alternative approach to assure the compatibility between production planning and scheduling is by integrating tactical and operational levels of decision [34, 60]. The main purpose of this integration is to obtain a production plan that is feasible at operational level. This feasibility is obtained after some iterations between both levels, during which they exchange information. Figure 7.2 illustrates this process.

However, the integration between production and scheduling planning models is difficult. The production planning model is *continuous* under time and scheduling planning model is *discrete*. As we mentioned above, managers at tactical level are concerned with planning the production for weeks or months (*how much must be produced*). Meanwhile, managers at operational level are concerned with sequencing the jobs of a day work over the machines (*when and how to produce*) [42]. One alternative to overcome this difficulty is to split the continuous model into periods of discrete time of production. This alternative is known as *Multi-Period Scheduling* [34, 60].

7.2 Multi-Period Scheduling Problem

Consider that at an operational level we have a classical JSP modeled as stated in Section 4.1.1. A multi-period scheduling problem for a JSP consists of replicating the JSP into P periods of discrete time. We will modify the previous notation of JSP to the one described below:

dp_p is the duration of period $p \in P$;
J is a partition of T. Each $J_l \in J$ defines a job of the JSP;
o_{ilpk} is the i-th task of the job J_l in the period $p \in P$ running over machine $k \in M$;
d_{ilpk} is the demanded time for running o_{ilpk};

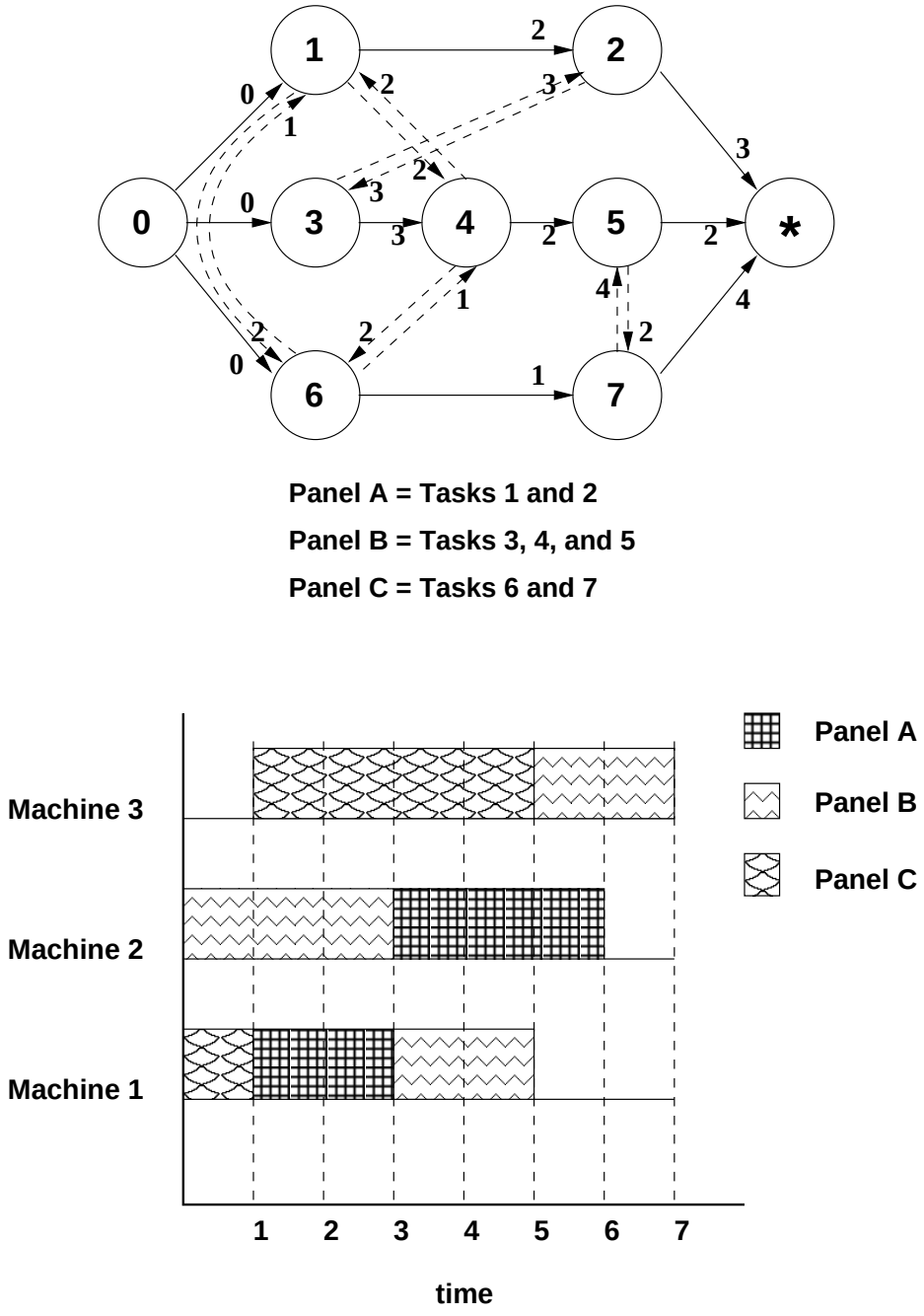


Figure 7.1: A 3×3 JSP represented as disjunctive graph (on top) and an optimal sequence for the corresponding makespan represented by a Gantt chart (in the bottom).

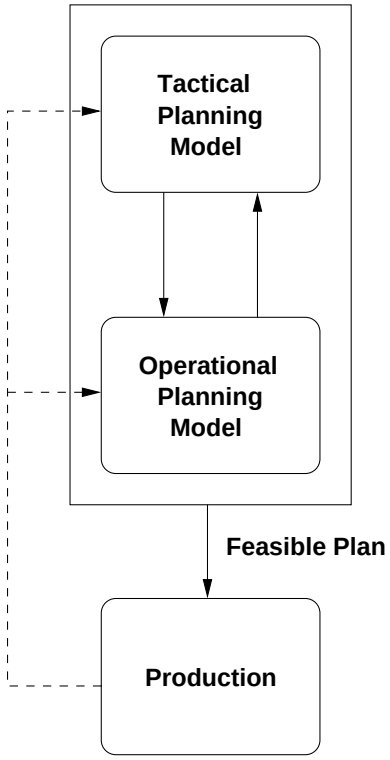


Figure 7.2: Model for exchange of information between tactical and operational level.

t_{ilpk} is the start time of o_{ilpk};
$L = \{o_{ilpk} : i = |J_l|\}$ is the set of the last task of each job;
Q_{ip} quantity of product i produced in period p.

The makespan problem of multi-period JSP consists of determining the start time of t_{ilpk} of each task. There are some different precedence relations among tasks of diferent periods that we can adopt [37]. We have considered the one that a task $o_{ilpk} \in L$ must be concluded at the end of p. That is, if the job must be finished in period p, it can be started in p or in a previous period. Other considerations that we have made to simplify the model, however not loosing the generality [36] are the following: (a) the setup time of machines is zero; (b) the technological precedence of each J_l is the same for every p; (c) $|J_l|$ is constant for all p. Therefore, the definition for this problem can be stated as

$$\min T_*$$

$$\begin{cases} t_{jlpk} - t_{ilpk} \geq d_{ilpk}.Q_{ip}, & (o_{ilpk}, o_{jlpk}) \in A & (1) \\ t_{ilpk} \geq 0, & i \in N & (2) \\ t_{jlpk} - t_{ilpk} \geq d_{ilpk}.Q_{ip} & or & \\ t_{ilpk} - t_{jlpk} \geq d_{jlpk}.Q_{jp}, & (o_{ilpk}, o_{jlpk}) \in D_k, k \in M & (3) \\ t_{ilpk} + d_{ilpk}.Q_{ip} \leq \sum_{q=1}^{p} dp_q & o_{ilpk} \in L & (4) \end{cases}$$

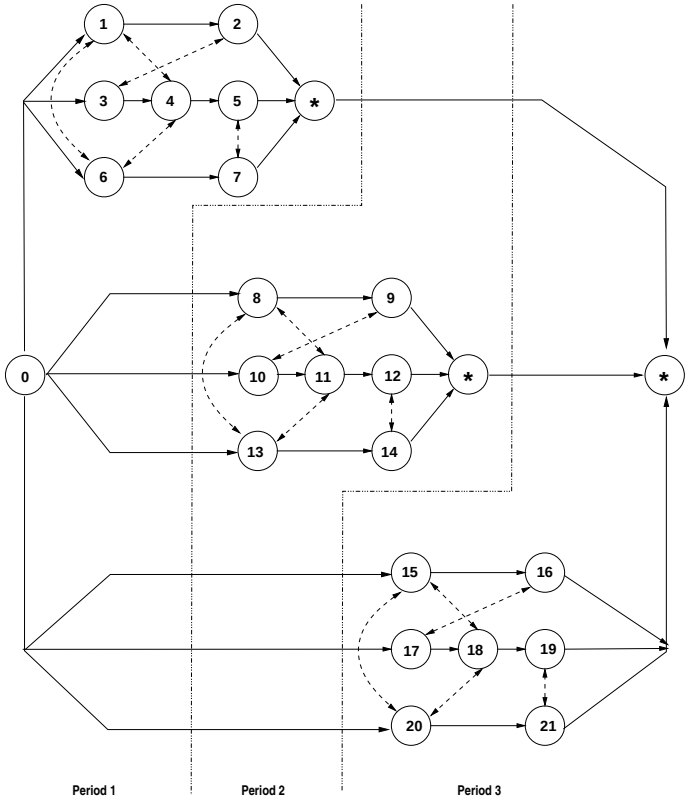


Figure 7.3: The representation of a multi-period JSP 3×3.

The restriction (1) imposes the technological precedence among tasks. The restriction (2) guarantees the completion of all tasks, and (3) assures the non preemption of machine $k \in M$. The restriction (4) assures that the last task of each job finishes at the end of the appropriate period. Figure 7.3 illustrates a JSP 3×3 replicated into 3 periods.

7.3 Modeling Multi-Period Job-Shop into SMV

Consider the manufacturing industry of car panels, described in Section 5.2. Consider that the tactical level wants to determine the quantity of panels to be produced along three periods of time. We have modeled this problem using SMV and a simplified excerpt of the main module is presented in Figure 7.4.

The variable NP (line 4) represents the number of periods of the multi-period JSP. In this case the number of periods is three. Lines 5-11 present the definition of the tasks of the problem. The lines 12-14 declare the variables m1, m2, and m3 that represent the machines of the multi-period JSP. These variables are instances of module sched that indeed implements the technological precedence and period precedence. For instance, line

```
1  MODULE main
2  VAR
3    boolean: begJS, endJS;
4    NP: 3..3;
5    t_1: task (1, 2, m1.j);
6    t_2: task (1, 3, m2.j);

7    t_3: task (2, 3, m2.j);
8    t_4: task (2, 2, m1.j);
9    t_5: task (2, 2, m3.j);

10   t_6: task (3, 1, m1.j);
11   t_7: task (3, 4, m3.j);


12   m1: sched3 (t_1, TRUE, NP, t_4, t_3.done, t_3.p, t_6, TRUE, NP);
13   m2: sched2 (t_2, t_1.done, t_1.p, t_3, TRUE, NP);
14   m3: sched2 (t_5, t_4.done, t_4.p, t_7, t_6.done, t_6.p);

15 ASSIGN
16   endJS := (t_2.p = NP) & (t_5.p = NP) & (t_7.p = NP);
```

Figure 7.4: A simplified excerpt of the main module of our SMV model for a multi-period JSP. To control the multi-periodicity, we have added variables such as NP and t_y.p where y=$\{1,3,4,6\}$. Variable t_1.p, for example, indicates in which period t_1 has been done.

14 declares machine m3. This machine is defined to run tasks t_5 and t_7. The technological precedence to these tasks are represented by t_4.done and t_6.done, respectively. These variables are boolean and they are set to TRUE when they have been concluded by their respective machines.

Besides the boolean variables to control technological precedence described above, we have to add one more parameter to each task of a machine, due to the multi-periodicity of the problem. This extra parameter indicates whether the technological precedence for each task has been concluded in a previous period. It works like a stock of almost-finished products. This means that a machine can run a task if its technological precedence has been concluded in a previous period. For example, m2 can run t_2 if t1 has been concluded in the current or previous period. This is represented in the model by the boolean variable t_1.done and by integer variable t_1.p, respectively. t_1.done and t_1.p are set and added appropriately in task module.

Finally, at line 16, the variable endJS represents the end of the multi-period JSP. Now, in contrast to the model presented in Figure 5.2, this variable is only set to TRUE when the last task of each job has been concluded at the third period.

Figure 7.5 presents an excerpt of the module task definition. This module models a machine executing a task. It has three parameters: **me** indicates job identity to whom the task belongs; time corresponds to the demanded execution time of the task; and sj relates to the job identity chosen to run - this choice is made by module sched. This modules uses two counters: i is a time counter; and p is period counter that indicates which period the task has been concluded.

Lines 6-10 models the execution of a task by a machine. In the beginning the time counter of all tasks are initiated with their respective time (line 6). When me is equal to sj the respective task runs. The task runs until the counter i is set to zero. Setting zero to i module task triggers the actions below at the same time:

- the variable done is set to TRUE to indicate the task completion (line 17);
- p is incremented by one to indicate that the task is one period ahead. p is incremented until all periods have been processed (lines 12-15);
- the counter i is reinitialized to time, indicating that is ready to the next period. Lines 7-10 model this described procedure.

The machines in a multi-period JSP must choose a task to run taking into considerations

```
1  MODULE task (me, time, sj)
2  VAR
3     i : 0..4;
4     p : 0..NP;

5  ASSIGN
6     init(i) := time;
7     next(i) := case
8                   (me = sj) & (i > 0): i - 1;
9                   1 : time;
10             esac;

11    init(p) := 0;
12    next(p) := case
13                  (i = 0) & (p < NP): p + 1;
14                  1 : p;
15             esac;

16 DEFINE
17    done := (i = 0);
```

Figure 7.5: Excerpt of our SMV model for a task of a multi-period JSP.

two points: (1) the conclusion of tecnological precedence of the task, and (2) the period of the JSP in which the tecnological precedence was concluded. Figure 7.6 presents an excerpt of a SMV program that models a machine that is supposed to run two tasks.

```
1  MODULE sched_m2 (t1, pt1_d, pt1_p, t2, pt2_d, pt2_p)

2  VAR
3    j : 0..3;
4    j_aux: 0..3;

5  ASSIGN
6    init (j_aux) := 0;
7    next (j_aux) := j;

8    j := case
9      t1.p < 3 & !t1.running & (pt1_d — pt1_p > t1.p) &
10     t2.p < 3 & !t2.running & (pt2_d — pt2_p > t2.p) : t1_id, t2_id;

11     (t1.p > 2 — !(pt1_p > t1.p)) &
12     t2.p < 3 & !t2.running & (pt2_d — pt2_p > t2.p) : t2_id;

13     (t2.p > 2 — !(pt2_p > t2.p)) &
14     t1.p < 3 & !t1.running & (pt1_d — pt1_p > t1.p) : t1_id;

15     1 : j_aux;
16   esac;

17   DEFINE
18     t1_id := t1.id;
19     t2_id := t2.id;
```

Figure 7.6: Excerpt of our SMV model for a machine that runs two tasks in an multi-period JSP.

The formal parameters t1 and t2 corresponds to the job identity to whom these task belong. The parameters pt1_d and pt1_p correspond to the technological precedence of task t_1 and the period in which pt1_d was finished, respectively. Lines 8–16 relates to the process of choice of a task to run made by the machine.

7.4 Conclusion

We have applied the MIN algorithm to compute the minimum makespan of the multi-period JSP relating the Figure 7.4. An optimal sequence for this problem is illustrated in Figure 7.7. This model has taken 4.9 seconds to be created and solved in a GNU/Linux platform running under Pentium Celeron 300 MHz with 128 Megabytes.

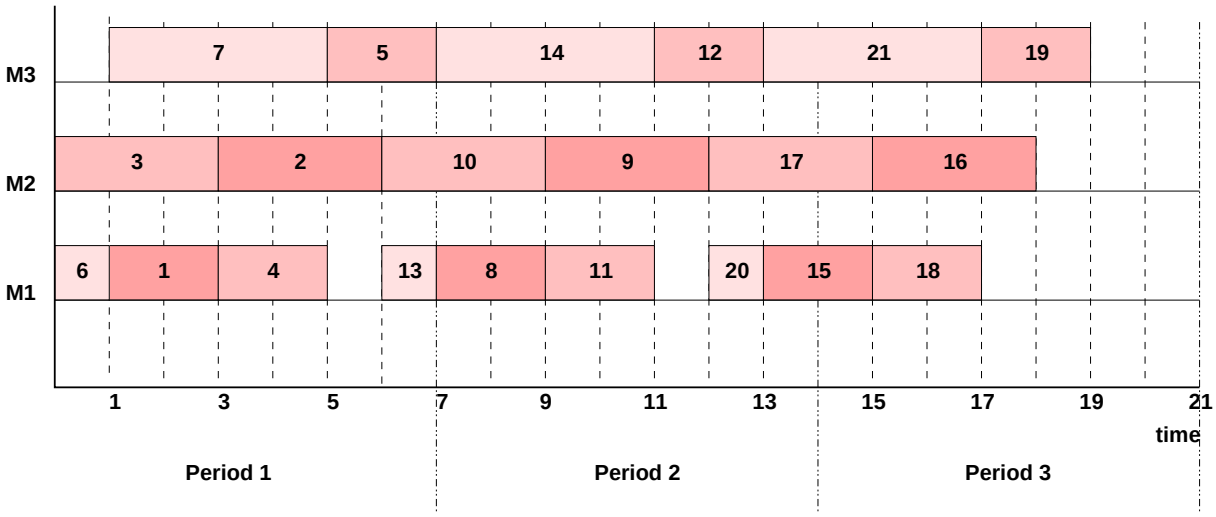


Figure 7.7: Makespan for a JSP 3×3 of 3 periods.

The purpose of this study was to show the viability of SMC in dealing with JSP mutli-period. SMC is an approach completely different from other works, such as Drumond [42], Carvalho [37] and Camargo [36]. The work of Drumond consists of implementing a decision support tool for production planning in the context of a port using CPLEX. Camargo and Carvalho solves integration problem by Benders decomposition. All these works express the restrictions of the problem by linear equations.

SMV approach consists of creating a model as we create computer program. The restrictions of the problem are expressed by modules of SMV created for this purpose. This approach, in enterprises that lack from people with mathematical skills, can be more comfortable than expressing restriction by mathematical equations.

Chapter 8

Conclusion and Future Works

Traditionally, exact solutions for deterministic scheduling problems have been obtained by using branch-and-bound (B&B) technique. The main approach of B&B technique is to find a feasible solution avoiding to explore some paths of the solution tree that certainly can not lead to a feasible one. Indeed avoiding certain paths implies in gaining of time/space performance.

On the other hand, formal verification techniques explore all paths (states) of the model being verified to assert properties about the model. The exploration of all states is a necessity because we can only assure a certain property if we explore all states of the model. In this work we have presented SMC (Symbolic Model Checking) based on OBDD (Ordered Binary Decision Diagram) as an exact solver for scheduling problems.

SMC is a formal verification technique originally proposed to verify concurrent finite states systems. Applying SMC to solving scheduling problems is a new approach considering the traditional ones, such as B&B (branch-and-bound) technique. There are several advantages of OBDD-SMC approach to the scheduling problems:

- the temporal logic of SMC can be used to reason about start and finish time of jobs relating to a certain makespan. That is, we can compute feasible schedules since we want a specific job finishes in a certain time. We can also search for feasible schedule given certain restrictions, such as: a specific job finishes in a certain time; or the makespan is less than or equal to a given units of time;
- the quantitative algorithms proposed here improve the potential of SMC in dealing with unpredictable changes in the problem specification. As an example, consider the following scenario. We have to compute a feasible schedule for a manufacturing plant

so that this schedule results in the minimum completion time of all jobs. After we have computed a feasible schedule an unpredictable event occurs: "two machines can not run in parallel anymore". This event changes the original problem specification. SMC is able to compute a new feasible schedule under this new problem configuration without the necessity of changes in the model or in any algorithm. It is just the case of specifying an appropriate CTL formulas and/or launching the appropriate algorithm.

In this work, we have applied OBDD-SMC to different scheduling scenarios. In each case, we have been successful in showing the practical applicability of this tool for solving such problems:

- the Video on Demand problem is related to assign client users to a video server. Efficiency in this kind of system is measured in terms of number of users served at the same time. In the case studied in this thesis, there was an implemented version of the server but there was no parameter about how close to the optimum efficiency the server was. We have applied OBDD-SMC to this problem and we have computed lower and upper bounds to the server. These bounds have pointed out errors in the implemented version. After the correction of them, the efficiency of the system server raised 40%;
- we have determined the minimum makespan of some instances of flow-shop (FSP) and job-shop problem (JSP). The computed makespan values were checked against a B&B tool to assure the results. We have modeled and computed correctly the makespan for several instances of JSP and FSP. Besides, we have changed the instances problem configuration and we have also successfully computed a new related makespan without any change in the modeled problem. It is an important feature of our approach. Being a general problem solver, OBDD-SMC can deal with some unexpected changes in the problem configuration more easily than certain specific tools, that in general are constructed with determined problem configuration in mind;
- we have presented a template for modeling deterministic scheduling problems via SMV, an OBDD-SMC tool. Besides, there is an available tool written in Java especially created for this purpose that automatically generates SMV models;
- the same template above can be used to deal with another class of problem: integration of production and scheduling plans. Integrating production and scheduling plannings is difficult because each plan is concerned with different kinds of information. Due to this difference not always we can find a scheduling plan that satisfies the correspondent production plan. Our model was applied to an hypothetical case and has determined correctly the makespan showing the feasibility of the approach.

The results of SMV (an OBDD-SMC tool) in computing the minimum makespan of JSP and FSP instances were compared against to the results obtained by CPLEX. The comparison was related to the processing time demanded by these tools to compute makespan. Considering that CPLEX is a commercial software product and has been receiving a lot of optimization since it was launched, the time performance of OBDD-SMC is very much comparable to CPLEX, mainly if we consider FSPs. These results points out that FSP is the class of scheduling problem that is more adherent to the OBDD-SMC approach.

Considering the 6×6 FSP instance, SMV was very much faster than CPLEX. This result occurred because the B&B technique of CPLEX had great difficulty to prove the optimality of the solution. This difficulty was provided by the configuration of this instance, whose demanded time of all tasks is the same value. That is, all tasks of the job demand the same amount of time. In this scenario, B&B technique has great difficulty to cut-off some paths of the solution tree. On the other hand, SMV was very fast in solving this problem. The time response for computing the makespan of this instance enhances our expectation that indeed FSP is the niche of application of OBDD-SMC approach.

On the other hand, at the present moment, SMV has some difficulty to deal with very complex JSP instances (above instances 6×6). This difficulty is related to the size of the OBDD. This size is a consequence of the number of the variables of the model and the ordering of the model along the OBDD. Depending this ordering, the size of OBDD can vary from linear to exponential to the number of the variables. Since the efficiency of OBDD-SMC is related to the size of OBDD, finding a good order for the variables of the model is very important. However, finding a good ordering depends of some heuristics that are difficult to implement in JSPs.

8.1 Future Works

The fact that SMV was much more efficient than CPLEX in solving complex FSP instances indicates that this class of problems adheres to OBDD-SMC more conveniently than JSP. We must investigate FSP instances greater than 6×6. That is, we need to model more complex FSP instances to evaluate the performance of the OBDD-SMC approach under these instances. In addition, further research is needed to characterize the FSP instances more accurately in terms of the time of the tasks and the related effect over B&B technique and OBDD-SMC itself.

Besides, we need to investigate how to tune OBDD-SMC to deal with JSP. For one side,

we can study alternatives to order variables of JSP models along the OBDD. This study can lead to more efficient sizes of OBDD relating to complex JSP instances. By other side, we can investigate the merging of some combinatorial optimization techniques to OBDD-SMC, such as the branch-and-bound techniques to prune some parts of the OBDD that contains paths that can not lead to a feasible solution.

In a another variant of research, consider all paths between states S and F in a Kripke model. Campos [22] has proposed the MAXCOUNT algorithm that returns the maximum number of events between these two states. We can extend this algorithm by printing the path corresponding to the result presented by MAXCOUNT. Let name the extended algorithm by MAXCOUNT-P. MAXCOUNT-P can be very useful in computing and presenting a feasible schedule so that the maximum number of jobs is finished in a given amount of time. Nowadays this information can be obtained in an indirect way by the counter-example.

Appendix A

Instances of JSP and FSP

This appendix presents the configuration of the job-shop (JSP) and flow-shop (FSP) we have computed in Section 5.3. Each instance identification is followed by its respective configuration. Each configuration is composed by as much lines as the number of jobs. Each configuration line presents the technological precedence of the respective job. This precedence is stated as list of $m_i(t)$ meaning that the job requires t units of time of machine i. For example, the job-shop instance 3×2 is a problem consisting of three jobs on two machines. The configuration has three lines. The first line states that job one (j1) runs at first on machine two (m2) taking eleven (11) units of time, then in the following it runs on machine 1 (m1) taking seven (7) units of time.

Job-Shops

JSP 3×2

j1: m2(11) m1(7)
j2: m1(10) m2(5)
j3: m1(9) m2(8)

JSP 3×3

j1: m1(3) m2(2) m3(2)
j2: m2(4) m3(3) m1(1)
j3: m2(1) m3(2) m1(3)

JSP 3×4

j1: m1(3) m2(2) m3(5)
j2: m2(4) m3(3) m4(6)
j3: m2(1) m3(2) m1(9) m4(8)

JSP 4×3

j1:	m2(1)	m1(2)	m3(3)
j2:	m3(5)	m1(4)	m2(5)
j3:	m1(3)	m3(4)	m2(4)
j4:	m2(6)	m3(2)	m1(2)

JSP 4×4

j1:	m1(2)	m2(3)	m3(3)	m4(2)
j2:	m1(3)	m3(3)	m2(2)	m4(3)
j3:	m1(1)	m2(3)	m4(2)	m3(1)
j4:	m1(4)	m4(1)	m3(3)	m2(3)

JSP 5×4

j1:	m1(2)	m2(3)		
j2:	m1(3)	m3(3)	m2(2)	
j3:	m1(1)	m2(3)	m4(2)	
j4:	m1(4)	m4(1)	m3(3)	
j5:	m4(4)	m3(4)		

JSP 5×5

j1:	m1(3)	m3(3)	m2(2)	m4(1)	m5(1)
j2:	m2(2)	m1(3)	m3(1)	m4(1)	m5(4)
j3:	m2(3)	m1(2)	m3(2)	m5(1)	m4(1)
j4:	m1(1)	m2(1)	m4(3)	m3(2)	m5(3)
j5:	m1(3)	m2(2)	m3(1)	m4(3)	m5(2)

JSP 6×6

j1:	m3(1)	m1(1)	m2(1)	m4(1)	m6(1)	m5(1)
j2:	m2(1)	m3(1)	m5(1)	m6(1)	m1(1)	m4(1)
j3:	m3(1)	m4(1)	m6(1)	m1(1)	m2(1)	m5(1)
j4:	m2(1)	m1(1)	m3(1)	m4(1)	m5(1)	m6(1)
j5:	m3(1)	m2(1)	m5(1)	m6(1)	m1(1)	m4(1)
j6:	m2(1)	m4(1)	m6(1)	m1(1)	m5(1)	m3(1)

Flow-Shops

FSP 3×2

j1:	m1(7)	m2(11)	
j2:	m1(10)	m2(5)	
j3:	m1(9)	m2(8)	

FSP 3×3

j1:	m1(3)	m2(2)	m3(2)
j2:	m1(1)	m2(4)	m3(3)
j3:	m1(3)	m2(1)	m3(2)

FSP 3×4

j1:	m1(3)	m2(2)	m3(5)	
j2:	m1(4)	m2(3)	m3(6)	
j3:	m1(1)	m2(2)	m3(9)	m4(8)

FSP 4×3

j1:	m1(2)	m2(1)	m3(3)
j2:	m1(4)	m2(5)	m3(5)
j3:	m1(3)	m2(4)	m3(4)
j4:	m1(2)	m2(6)	m3(2)

FSP 4×4

j1:	m1(2)	m2(3)	m3(3)	m4(2)
j2:	m1(3)	m2(2)	m3(3)	m4(3)
j3:	m1(1)	m2(3)	m3(1)	m4(2)
j4:	m1(4)	m2(3)	m3(3)	m4(1)

FSP 5×4

j1:	m1(2)	m2(3)	
j2:	m1(3)	m2(2)	m3(3)
j3:	m1(1)	m2(3)	m4(2)
j4:	m1(4)	m3(3)	m4(1)
j5:	m3(4)	m4(4)	

FSP 5×5

j1:	m1(3)	m2(3)	m3(2)	m4(1)	m5(1)
j2:	m1(2)	m2(3)	m3(1)	m4(1)	m5(4)
j3:	m1(3)	m2(2)	m3(2)	m4(1)	m5(1)
j4:	m1(1)	m2(1)	m3(3)	m4(2)	m5(3)
j5:	m1(3)	m2(2)	m3(1)	m4(3)	m5(2)

FSP 6×6

j1:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)
j2:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)
j3:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)
j4:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)
j5:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)
j6:	m1(1)	m2(1)	m3(1)	m4(1)	m5(1)	m6(1)

Appendix B

Mixed Integer and Linear Programming Model for JSP

Recall JSP definition in Section 4.1.1. JSP can be represented by a *disjunctive* graph $G = \{N, A, D\}$, such that: $N = T \cup \{0, *\}$ is the set of nodes of the graph; A is the set of arcs; and D is the set of disjunctive arcs. d_i is the fixed processing time of task $i \in T$, and t_i represents the earliest possible start time of task i, that has to be determined during optimization. The mathematical formulation for this problem is presented below:

$$\begin{cases} min\ t_* & & & & \\ & t_j - t_i & \geq & d_i, & (i,j) \in A & (1) \\ & t_i & \geq & 0, & i \in N & (2) \\ t_j - t_i \geq d_i \vee t_i - t_j & & \geq & d_j, & (i,j) \in D_k, k \in M & (3) \end{cases}$$

where d_i is the fixed processing time of task i (all arcs (i,j) of G are labeled by d_i), and t_i represents the earliest possible start time of task i, that has to be determined during optimization.

The restriction (1) imposes an order of execution among tasks of each job. The restriction (2) guarantees the completion of all tasks, and (3) assures the non preemption of machine $k \in M$. Figure 4.4 illustrates a disjunctive graph representing job-shop 3×3 (3 jobs over 3 machines).

We have used AMPL [1] to express the mathematical formulation of JSP. Figure B.1 presents a mixed integer-linear (MILP) model for a JSP 3×2. The sets Tom1 and Tom2 (lines 3 and 4) correspond to the sets D_k, and set Prec correspond to set A, in the mathematical formulation of JSP above.

```
1   set M := {1 .. NumMachines};      # machines
2   set T := {0 .. NumTasks+1};       # tasks
3   set Tom1 within {T};              # tasks that must be executed by machine 1
4   set Tom2 within {T};              # tasks that must be executed by machine 2
5   set Prec within {T cross T};      # technological precedence

6   param NumMachines > 0 integer;    # number of machines
7   param NumTasks > 0 integer;       # number of tasks
8   param d{i in T} >= 0;             # processing time of tasks
9   param z > 0;                      # high value constant

10  var t{i in T} >= 0;               # the beginning of task t(i)
11  var x1{(i,j) in Tom1 cross Tom1} binary; # mutual exclusion for machine 1
12  var x2{(i,j) in Tom2 cross Tom2} binary; # mutual exclusion for machine 2

13  minimize obj: t[NumTasks +1];     # Objective function

################
# Restriction for a JSP 3 X 2 #
################
14  subject to prec_job {(i,j) in Prec}: t[j] - t[i] >= d[i];
15  subject to preempt1_1 {i in Tom1, j in Tom1: i < j}: t[i] + x1[i,j]*d[i] - (1 - x1[i,j])*z <= t[j];
16  subject to preempt1_2 {i in Tom1, j in Tom1: i < j}: t[j] + (1 - x1[i,j])*d[j] - x1[i,j]*z <= t[i];
17  subject to preempt2_1 {i in Tom2, j in Tom2: i < j}: t[i] + x2[i,j]*d[i] - (1 - x2[i,j])*z <= t[j];
18  subject to preempt2_2 {i in Tom2, j in Tom2: i < j}: t[j] + (1 - x2[i,j])*d[j] - x2[i,j]*z <= t[i];
```

Figure B.1: MILP model for JSP 3×2.

```
############
# Data for a JSP 3 X 2 #
############
1   param NumMachines := 2;
2   param NumTasks := 6;
3   param z:= 1000;
4   param d := 0 0   1 11   2 7    3 10   4 5    5 9    6 8    7 0;

5   set Prec := (0,1) (1,2) (2,7) (0,3) (3,4) (4,7) (0,5) (5,6) (6,7);
6   set Tom1 := 2 3 5;
7   set Tom2 := 1 4 6;
```

Figure B.2: The data for the MILP model for JSP 3×2

Line 14 corresponds to the restriction 1 relating to technological precedence. Lines 15 through 18 correspond to the restriction 3 relating to the preemption. The disjunction of this restriction is modeled by integer variables (lines 11 and 12) that can assume values 0 or 1.

The data to the CPLEX model is presented in Figure B.2. The processing time for each task is given line 4. In this line, for example, task 1 takes 11 units of time. Line 5 declares the technological precedence. Tasks 0 and 7 correspond to the dummy tasks of the disjunctive graph, respectively, 0 and $*$. For example, the sequence (0,1) (1,2) (2,7) says that task 1 has no technological precedence, task 2 depends of task 1, and the (dummy) task 7 depends of task 2. Lines 6 and 7 declare the tasks that must run on its respective machine.

Adapting this model to other JSP instances implies in defining appropriately the lines 1–4, 11–12, and 15–18, in Figure B.1. It is also necessary to change lines 4–7, in Figure B.2.

Appendix C

Automatic Generator for SMV Programs

Figure 5.4 presented the code to assure the choice of only one task to run and that this choice occurs when the machine is idle. However, this codification grows exponentially with the number of machines modeled by the module. Precisely, the number of lines in the case is $2^m - 1$, where m is the number of machines being modeled. Depending the size of the JSP instance, it is a tedious and error-prone activity to made by hand. For this reason, we have created a program to make this task automatic.

The program receives a file with the description of a JSP and generates a correspondent SMV program. The program was written in Java and can be obtained by email (autran@dcc.ufmg.br). The format of the input file for a JSP 3×2 is presented below:

```
j1: m1(7) m2(11)
j2: m2(10) m1(5)
j3: m1(9) m2(8)
```

Each line corresponds to a job. The description of a job begins with its identification followed by the tasks and the respective processing time. The tasks correspond to the identity of the machines that must process the job. In the example above, the first line, for instance, declares that the identity of the job is j1. This job must be processed by two machines: m1 taking 7 units of time, and m2 that takes 11 units of time.

Bibliography

[1] AMPL. http://www.ampl.com.

[2] Adams, J., Balas, E., and Zawack, D. The shifting bottleneck procedure for job shop scheduling. *Management Science 34*, 3 (March 1988), 391–401.

[3] Alur, R., Courcourbetis, C., and Dill, D. Model-checking for real-time systems. In *5th Symposium on Logic in Computer Science* (1990).

[4] Anderson, E., Glass, C., and Potts, C. Local search in combinatorial optimization: applications in machine scheduling. Research report OR56, University of Southampton, 1995.

[5] Applegate, D., and Cook, W. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing 3* (1991), 149–156.

[6] Baker, K. R. *Introduction to sequencing and scheduling.* John Wiley & Sons, 1974.

[7] Balas, E. Machine sequencing via disjuntive graphs: an implicit enumeration algorithm. *Operational Research 17* (1969), 941–957.

[8] Balas, E., Lenstra, J., and Vazacopoulos, A. One machine scheduling with delayed precedence constraints. *Management Science 41* (1995), 94–109.

[9] Barnes, J., Laguna, M., and Glover, F. An overview of tabu search approaches to production scheduling problems. In *Symposium of Intelligent Scheduling Systems* (1992), pp. 30–50.

[10] Berson, S., Muntz, R., Ghandeharizadeh, S., and Ju, X. Staggered striping in multimedia information systems. In *ACM SIGMOD Conference* (1994), pp. 79–90.

[11] Bertini, L., Campos, S., Jamil, G. L., Macêdo, A., Ribeiro-Neto, B., dos Santos, C. F., and dos Santos, D. A. S. Análise de desempenho do servidor de vídeo almadem-vod. In *SBMIDIA'99 - Simpósio Brasileiro de Sistemas Multimídia*

e Hipermídia (Goiânia, GO, Jun. 1999), SBC - Sociedade Brasileira de Computação, pp. 259–276. (in Portuguese).

[12] Blażewicz, J. Selected topics in scheduling theory. *Annals of Discrete Mathematics 31* (1987), 1–60.

[13] Blażewicz, J., Domschke, W., and Pesch, E. The job shop scheduling problem: conventional and new solution techniques. *European Journal of Operational Research 93* (1996), 1–33.

[14] Blażewicz, J., Ecker, K. H., Pesch, E., Schimidt, G., and Węglarz, I. *Scheduling computer and manufacturing processes*. Springer-Verlag, 1996.

[15] Brucker, P., Jurisch, B., and Krämer, A. The job-shop problem and immediate selection. *Annals of Operations Research 50* (1996), 73–114.

[16] Brucker, P., Jurisch, B., and Sievers, B. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics 49* (1994), 107–127.

[17] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers C-35*, 8 (August 1986), 677–691.

[18] Campos, S., Clarke, E., Marrero, W., and Minea, M. Verifying the performance of the PCI local bus using symbolic techniques. In *International Conference on Computer Design* (1995), IEEE, pp. 72–78.

[19] Campos, S., Clarke, E., Marrero, W., and Minea, M. Verus: a tool for quantitative analysis of finite-state real-time systems. *ACM SIGPLAN Notices 30*, 11 (1995), 70–78.

[20] Campos, S., Clarke, E., Marrero, W., Minea, M., and Hiraishi, H. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium* (1994).

[21] Campos, S., Ribeiro-Neto, B., Macêdo, A., and Bertini, L. Formal verification and analysis of multimedia systems. In *ACM Multimedia* (Orlando, MI, USA, Oct. 1999), ACM, pp. 419–430.

[22] Campos, S. V. A. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, Carnegie Mellon University, September 1996. CMU-CS-96-199.

[23] Carlier, J., and Pinson, E. An algorithm for solving the job-shop problem. *Management Science 35*, 2 (February 1989), 164–176.

[24] Carlier, J., and Pinzon, E. A practical use of jackson's preemptive schedule for solving job shop problem. *Annals of Operations Research 26* (1990), 269–287.

[25] Carlier, J., and Pinzon, E. Adjustments of heads and tails for the job-shop problem. *European Journal of Operational Research 78* (1994), 146–161.

[26] Cavada, R., Cimatti, A., Olivetti, E., Roveri, M., and Pistore, M. *NuSMV 2.0 - User's Manual.* Istituto per la Ricerca Scientifica e Tecnologica, http://nusmv.itrst.itc.it. Email: nusmv@irst.itc.it.

[27] Chua, T., Li, J., Ooi, B., and Tan, K. Disk striping strategies for large video-on-demand servers. In *ACM International Multimedia Conference* (1996), pp. 297–306.

[28] Clarke, E., Gupta, A., Kukula, J., and Strichman, O. SAT based abstraction-refinement using ILP and machine learning techniques. In *14th International Conference on Computer-Aided Verification - CAV'02* (2002).

[29] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs* (1981), vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 52–71.

[30] Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D., McMillan, K. L., and Ness, L. A. Verification of the Futurebus+ cache coherence protocol. In *International Symposium on Computer Hardware Description Languages and their Applications* (April 1993), L. Claesen, Ed., North-Holland.

[31] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking.* MIT Press, 1999.

[32] Clarke, E. M., and Wing, J. M. Formal methods: state of art and future directions. Tech. Rep. CMU-CS-96-178, Carnegie Mellon University, Computer Science Department, Sep. 1996. http://www.cs.cmu.edu/Reports/1996.html.

[33] Corbett, J. C., and Avrunin, G. S. Using integer programming to verify general safety and liveness properties. *Formal Methods in Systems Design 6*, 1 (Jan. 1995), 97–123.

[34] Dauzère-Pérès, S., and Lassere, J.-B. An integrated approach in production planning and scheduling. *Lecture Notes in Economics and Mathematical Systems 411* (1994).

[35] Davis, L. Job shop scheduling with genetic algorithms. In *International Conference on Genetic Algorithms and their Applications* (1985), J. Grefenstette, Ed., pp. 136–140.

[36] de Camargo, R. S. *Contribuições para Modelagem da Integração dos Problemas de Planejamento Tático e Operacional*. PhD thesis, Engenharia de Produção - Universidade Federal de Minas Gerais, 2002.

[37] de Carvalho, C. R. V. *Une Proposition D'Intégration de La Planification et L'Ordennancement de Production: Application de La Méthode de Benders*. PhD thesis, Université Blaise Pascal - Clermont II, 1998.

[38] de Carvalho, C. R. V. Organização e controle de chão de fábrica. Course material (in Portuguese), 1999.

[39] Déharbe, D., and Borrione, D. Semantics of a verification-oriented subset of VHDL. In *CHARME'95 - Correct Hardware Design and Verification Methods* (1995), P. Camurati and H. Eveking, Eds., vol. 987 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 293–310.

[40] Della Croce, F., Tadei, R., and Volta, G. A genetic algorithm for the job shop problem. *Computers & Operations Research 22* (1995), 231–252.

[41] Dorndorf, U., Pesch, E., and Huy, T. P. Recent developments in scheduling. In *International Conference on Operations Research* (Aug. 1998), Springer-Verlag.

[42] Drumond, F. P. *Sistema de Apoio à Decisão para Planejamento da Produção de um Terminal Portuário*. PhD thesis, Pós Graduação em Ciência da Computação - Universidade Federal de Minas Gerais, 1998.

[43] Fisher, H., and Thompson, G. Probabilistic learning combinations of local job-shop scheduling rules. In *Industrial Scheduling* (New Jersey, 1963), J. Muth and G. Thompson, Eds., Prentice Hall, Englewood Cliffs.

[44] Freedman, C., and DeWitt, D. The SPIFFI scalable video-on-demand system. In *ACM International Multimedia Conference* (1995), ACM, pp. 352–363.

[45] Garey, M. R., and Johnson, D. S. *Computers and Intractability : A Guide to the Theory of Np-Completeness*. Books in the Mathematical Sciences. W.H. Freeman & Co, 1979.

[46] Ghandeharizadeh, S., Kim, S. H., and Shahabi, C. On configuring a single disk continuous media server. *ACM Sigmetrics Performance 3* (May 1995), 43–52. Springer Verlag ed.

[47] Gonzales, T., and Sahni, S. Open shop scheduling to minimize finish time. *Journal of ACM 23* (1976), 665–679.

[48] Graham, R., Lawler, E., Lenstra, J., and Rinnooy Kan, A. Optimization and approximation in deterministic sequencing and scheduling theory: A survey. *Annals of Discrete Mathematics 5* (1979), 287–326.

[49] G.R.Bitran, and D.Tirupati. *Handbook in Operations Research and Management Science*. S.C. Graves and A.H.G. Rinooy Kan and P.H. Zipkin, 1993, ch. Hierarchically Production Planning.

[50] Hartonas-Garmhausen, V., Clarke, E. M., and Campos, S. Deadlock prevention in flexible manufacturing systems using symbolic model checking. In *Intl. Conf. on Robotics and Automation* (1996).

[51] Hennessy, J. L., and Patterson, D. A. *Computer Architecture A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, 1996.

[52] Henzinger, T., Ho, P., and Wong-Toi, W. Hytech: the next generation. In *IEEE Real-Time Systems Symposium* (1995).

[53] Hoare, C. *Communicating Sequential Process*. Prentice-Hall, 1985.

[54] Holland, J. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[55] Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[56] ILOG. CPLEX. http://www.ilog.com.

[57] Johnson, S. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly 1* (1954), 61–68.

[58] Jones, C. B. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[59] Jurisch, B. *Scheduling jobs in shops with multi-purpose machines*. PhD thesis, Universität Osnabrück, 1992. Fachbereich Mathematik/Informatik.

[60] Lassere, J.-B. An integrated model for job-shop planning and scheduling. *Management Science 38*, 2 (1992), 1201–1211.

[61] Lawler, E., and Martel, C. Preemptive scheduling of two uniform machines to minimize the number of late jobs. *Operations Research 37* (1989), 314–318.

[62] Lee, C.-Y., Lei, L., and Pinedo, M. Current trends in deterministic scheduling. *Annals of Operations Research 70* (1997), 1–41.

[63] Macêdo, A., Campos, S., and Carvalho, C. Symbolic model checking: A new approach for solving scheduling problems. In *X CLAIO - Latin-Ibero-American Conference on Operations Research and Systems.* (Sept. 2000), IMSIO - Mexican Institute for Operations Research and Systems. digital proceedings with no page numbering.

[64] Manna, Z., and Pnueli, A. *The temporal logic of reactive and concurrent systems: specification.* Springer-Verlag, 1981.

[65] Martin, P., and Shmoys, D. A new approach to computing optimal schedules for job-shop scheduling problem. In *5th Conference on Integer Programming and Combinatorial Optimization - IPCO* (1996).

[66] McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[67] Moving Picture Experts Group. MPEG-1. ISO/IEC JTC1/SC29/WG11, June 1996. http://mpeg.telecomitalialab.com/standards/mpeg-1/mpeg-1.htm.

[68] Nakano, R., and Yamada, T. Conventional genetic algorithms for job shop problems. In *4th International Conference on Genetic Algorithm* (1991), R. Belew and L. Booker, Eds., Morgan Kaufmann, pp. 474–479.

[69] on Demand Lab., V. Projects. http://www.vod.dcc.ufmg.br/vod/projetos.jsp.

[70] Ozden, B., Rastogi, R., and Silberchatz, A. On the design of a lowcost video-on-demandstorage system. In *ACM International Multimedia Conference* (1996), ACM, pp. 40–54.

[71] Parker, R. G. *Deterministic Scheduling Theory.* Chapman & Hall, 1995.

[72] P.Bellini, R.Mattolini, and P.Nesi. Temporal logic for real-time system specification. *ACM Computing Surveys 32*, 1 (2000), 12–42.

[73] Pinedo, M. *Scheduling: theory, algorithms and systems.* Prentice Hall, 1995.

[74] Queille, J., and Sifakis, J. Specification and verification of concurrent systems in caesar. In *5th International Symposium on Programming* (1981), vol. 137 of *Lecture Notes in Computer Science*, pp. 337,351.

[75] Roy, B., and Sussman, B. Les problèmes d'ordonnancement avec constraintes disjonctives, 1964. Note DS n^o 9 bis, SEMA, Paris.

[76] Santos, J., and Muntz, R. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In *ACM International Multimedia Conference* (1998), pp. 303–308.

[77] Spivey, J. M. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[78] Trade Association, S. Small Computer System Interface. http://www.scsita.org.

[79] University, C. M. Formal methods - model checking. www-2.cs.cmu.edu/~modelcheck.

[80] Vernick, M., Venkatramani, C., and Chiueh, T. Adventures in building the stony brook video server. In *ACM International Multimedia Conference* (1996), ACM, pp. 287–295.

[81] Wing, J. M. A specifier's introduction to formal methods. *IEEE Computer 23*, 9 (September 1990), 8–24.