ABSTRAÇÕES SEMIAUTOMÁTICAS NA VERIFICAÇÃO DE MODELOS SIMÉTRICOS

PEDRO DE CARVALHO GOMES

VERIFICATION OF SYMMETRIC MODELS USING SEMIAUTOMATIC ABSTRACTIONS

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: SÉRGIO VALE DE AGUIAR CAMPOS

Belo Horizonte

June 2010

PEDRO DE CARVALHO GOMES

ABSTRAÇÕES SEMIAUTOMÁTICAS NA VERIFICAÇÃO DE MODELOS SIMÉTRICOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Sérgio Vale de Aguiar Campos

Belo Horizonte

Junho de 2010

Gomes, Pedro de Carvalho.

G633a Abstrações semiautomáticas na verificação de modelos simétricos. / Pedro de Carvalho Gomes. — Belo Horizonte, 2010.
xxiv, 64 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Prof. Sérgio Vale de Aguiar Campos.

1. Redes de Computadores - Teses. 2. Redes P2P- Teses. 3. Verificação Formal - Teses. 4. Verificação de Modelos - Teses. 5. Simetria - Teses.
I. Orientador. II. Título.

CDU 519.6*22(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Abstrações semiautomáticas na verificação de modelos simétricos

PEDRO DE CARVALHO GOMES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. SÉRGIO VALE AGUIAR CAMPOS - Orientador
Departamento de Ciência da Computação - UFMG

PROF. DILIAN GUROV
KTH, Royal Institute of Technology - Suécia

PROF. ANTONIO ALFREDO FERREIRA LOUREIRO
Departamento de Ciência da Computação - UFMG

PROF. MARK ALAN JUNHO SONG
Departamento de Ciência da Computação - PUC-MG

Belo Horizonte, 21 de junho de 2010.

Dedico este trabalho à minha família e aos amigos que me incentivaram durante esta jornada.

I dedicate this work to my family and friends which supported me along this journey.

Acknowledgments

Agradeço primeiramente a minha família, que esteve junto comigo durante todo este tempo. Só eles sabem das coisas que abri mão para fazer o mestrado. Ao professor Sérgio Campos por me ensinar a técnica de Model Checking, pela atenção e enorme paciência. Ao agora doutor Alex Borges, que foi um grande companheiro de pesquisa. A todos os amigos do laboratório LUAR, que me acompanharam durante esta jornada. A todos os amigos, sejam do CT, do DCC, Avacáia, da rua ou de Recife. Apesar de não darem sossego, compreenderam a necessidade da minha reclusão por vários momento. Outras pessoas tiveram participação no meu mestrado, seja na escolha de mudança de rumos da minha vida profissional ou no suporte a ela. Obrigado a todos.

First I want to thank my family, which stood by me during all this time. Only they know the things I had to give up to make the M.Sc. To professor Sergio Campos for teaching me Model Checking, for the attention and gigantic patience. To the now PhD Alex Borges, which was a great research fellow. To all my friends from LUAR, which where together along this journey. To all other friends, may them be from CT, DCC, Avacaia, from my neighborhood or Recife. Even though they didn't give me a break they comprehended my need for reclusion in many moments. Other people had participated on the Master, may it be on the change of direction if my professional career or supporting the decision. Thank you all.

"The roots of education are bitter, but the fruit is sweet."
(Aristotle)

Resumo

A Verificação de Modelos é uma técnica poderosa de verificação automática de sistemas concorrentes. Ela explora automaticamente os estados de um modelo que representa o sistema para provar sua correção com relação a especificações formais, descritas usando alguma lógica temporal. Apesar de sua importância e ampla aplicação, a Verificação de Modelos sofre com o problema da explosão de estados: o número de estados do modelo é exponencial ao seu tamanho; isto limita o tamanho dos modelos possíveis de serem verificados.

Diversas técnicas foram propostas para contornar o problema. Dentre elas, o uso de abstrações é considerada uma das mais genéricas e eficientes. A adoção de abstrações consiste em gerar um modelo reduzido a partir do modelo original através da fusão ou remoção de estados que supõe-se irrelevantes com relação à propriedade sendo verificada. Outra técnica é a redução por simetria. Ela baseia-se na observação que diversos sistemas apresentam considerável grau de simetria, e estados considerados equivalentes podem ser agrupados. Assim o espaço dos estados a ser considerado é significantemente menor e a exploração de apenas um dos estados do mesmo grupo é suficiente para provar a correção de alguma propriedade.

Este trabalho combina ambas as técnicas para produzir modelos reduzidos, que podem ser verificados em tempo factível. É apresentada uma metodologia para gerar abstrações semiautomáticas, baseada na simetria do modelo. A ideia chave é que, na verificação de certas propriedades, a remoção de componentes simétricos de um modelo tem um impacto pequeno na perda de informação causada pelas abstrações já que a contra-parte simétrica ainda está presente. A metodologia define premissas de modelagem para tornar a adoção das abstrações semiautomática, ou seja, sem a necessidade de alterar a descrição do modelo. Além disso, são apresentados padrões de abstrações baseados na simetria do sistema e mostra-se quais especificações são consistentes com cada padrão.

As técnicas apresentadas neste trabalho são especialmente úteis na verificação de sistemas de computação que apresentam uma considerável replicação de estrutura.

Tal característica pode ser observada em memórias, caches, protocolos de barramento, programas com vários processos e protocolos de rede. Foi implementado no trabalho o modelo de uma rede *P2P Live Streaming* para validar a metodologia. Neste modelo cada participante recebe e encaminha dados para seus parceiros para reconstruir o conteúdo ao vivo original. O fato de todos os participantes serem processos distintos que compartilham o mesmo código torna este modelo altamente simétrico e assim um exemplo válido.

A redução obtida com a metodologia provou ser bastante significativa. Por exemplo, o cálculo do número de estados alcançáveis do modelo original, de um total de aproximadamente 2^{73} estados possíveis, não terminou após mais de duas semanas de computação intensa. Em contrapartida, a mesma computação para os modelos reduzidos terminou em menos de três minutos em todos os casos e o número máximo encontrado de estados alcançáveis foi de aproximadamente 2^{19}.

Palavras-chave: Verificação Formal, Verificação de Modelos, Simetria, Abstração.

Abstract

Model Checking is a powerful method for the formal verification of concurrent systems. It explores automatically the state-space of a model that represents the system to prove its correctness in relation to formal specifications, which are described using some temporal logic. Despite its importance and wide application, Model Checking suffers with the state-space explosion: the number of states of a model is exponential to its size; thus it limits the size of the models that may be verified.

Many techniques were proposed to overcome this problem. Among them, the use of abstractions is considered one of the most general and efficient. The adoption of abstractions consists of generating a reduced model from the original model by merging or removing states that are irrelevant to the specification being verified. Another technique is the symmetry reduction. It is based on the observation that several models present some level of symmetry, and states considered equivalent can be grouped. Thus, the state-space to be considered is significantly smaller and the exploration of only one of the states of the same group is sufficient to prove the correctness of some propriety.

This work combines both techniques to produce reduced models that can be verified at feasible time. It presents a methodology to generate semiautomatic abstractions, based on the model symmetry. The key idea is that, for the verification of certain proprieties, the removal of symmetric components of a model has a small impact on information loss caused by the abstractions since its symmetric counterpart is still represented. The methodology defines modeling premises to make the abstraction adoption semiautomatic, i.e., without the need to alter the model description. Moreover, it presents abstraction patterns based on the system symmetry and shows which specifications are consistent with each pattern.

The techniques presented in this work are specially useful on the verification of computation systems that present considerable replication of structures. This characteristic can be observed in memories, caches, bus protocols, multi-processes applications and network protocols. In this work the model of a P2P Live Streaming application was implemented to validate the methodology. At this model each participant receives

and forwards data to its partners to reconstruct the original live stream. The fact that all peers are distinct processes that share the same code makes this model highly symmetric and thus a valid example.

The reductions obtained by the methodology proved to be very significant. I.e, the calculation of the number of reachable states of the original model, from a total of approximately 2^{73} possible states, has not finished after more than two weeks of intensive computation. In contrast, the same computation for the reduced models finished in less than two minutes in all cases and the maximum number of reachable states found was approximately 2^{19}.

Keywords: Formal Verification, Model Checking, Symmetry, Abstraction.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Formal Methods refer to the processes of formal specification and verification of computation systems. In particular, **Formal Verification** refers to the process of analyzing a system using some formal technique in order to prove whether the system meets its formal specifications. In contrast to other validation approaches, such as testing and simulation, Formal Verification fully guarantees the correctness of a system with respect to its specifications.

Among the techniques of Formal Verification, **Model Checking** is considered one of the most popular and consolidated. In Model Checking, systems are modeled as a finite state machine and the formal specifications are enunciated as a formula in some temporal logic. After implementing the model and providing the specifications, the verification of correctness is totally automatic. A software called model checker explores exhaustively the state-space of the model to check if it satisfies its properties. In case of failure, the model checker provides a counterexample, which is extremely useful for debugging the non-compliance found.

Despite being a very popular technique and being successfully adopted on the verification of several hardware systems, software systems and communication protocols over decades, Model Checking coexists with the problem of **state-space explosion**. The problem is characterized by the fact that the number of states in a model is exponentially proportional to the number of variables it has. In other words, models that contain a considerable number of variables have an enormous amount of states to explore, which can make its verification impractical. The consequence is that the state-space explosion limits the complexity of models that can be verified.

Techniques from the symbolic representation of models gave a significant growth in the number of possible states that can be explored. These techniques use propositional formulas to represent the transition relation between states in contrast to the

use of adjacency lists, used in the explicit representation. Such formulas can be implemented using extremely compact data structures, which increases significantly the size of a model that can be verified.

Despite the considerable progress achieved by the symbolic representation, many real systems are still too complex to be verified with existing computing resources. Several complementary techniques have emerged in the last two decades to overcome this limitation and increase the size of models that can be verified. Many of these focus on the controlled reduction of the model of the system. That is, these techniques remove or collapse parts of the model without changing the result of the property being verified.

The use of abstractions is one of the main techniques of model reduction. It consists on the generation of a smaller abstract model, removing or grouping states from the original model. There are two approaches to abstract information from the original model. The first one is called over-approximation, where states are grouped. The second one is called under-approximation, where states and transitions are eliminated. The two approaches differ on the way they handle the information loss and also the kind of properties that can be verified without interfering in the result. The use of abstractions is considered one of the most general and effective techniques of model reduction.

Another technique for reducing the number of states to be considered is the symmetry reduction. It is widely adopted to reduce models that have large amount of replicated structures, such as a bank of registers. The idea behind this approach is that some states present symmetrical behavior for the verification of some properties. So, instead of exploring all states, the exploration of only one of these symmetric states is sufficient.

Although both techniques have been widely explored, and have even been combined, the adoption of both demands high computing cost. It is desirable to minimize such cost. The current work aims to eliminate much of this cost by defining a methodology to assist the person who will check the model to exploit the symmetry of a model in a simple way. Thus eliminating unnecessary calculation.

This work contributes to the study of model reduction presenting a semiautomatic technique for the definition of abstractions in reactive models. Reactive systems are those were its components react to stimulus from the environment. We call semiautomatic abstractions the removals of parts of the model by human intervention, but without changing its description. We present modeling techniques that allow the abstraction of entire components of the model without the need of code editing. Furthermore, we propose a technique for the formal identification of the symmetry in the

original model. Then we use the formal symmetry to generate reduced models by under-approximation. We also present the description of which properties are compliant to each abstract model.

Several computer systems have the symmetrical structure that is explored in this work. Some examples are circuits with multiple identical components, programs with more than one execution flow, bus protocols and network protocols. We chose the model of the GridMedia network to validate this work, which is a P2P Live Streaming system. In this system, a special node called server generates the transmission and split it into small data chunks, which are distributed to the nodes on the network. Each node reconstructs the original transmission information by requesting to other participants the chunks of data that it does not have, and forwarding the ones that it has received. All participants of the network, except the server, share the same code, which makes the model highly symmetric and therefore a valid example.

After implementing the model and defining the formal properties, we verified the model using the proposed methodology. The results show that the abstractions used in this work achieved considerable reductions for both verification time and size of the model. For example, it was not possible to complete the computation of the number of reachable states for the original model server after more than two weeks running on a dedicated processing. The same computation finished in less than three minutes for all reduced models. Also the maximum number of reachable states found was less than 2_{20}, many orders of magnitude lower than the approximately 2^{73} possible states, which demonstrates the power of model reduction through elimination of components.

1.1 Organization

The text is divided into two parts. The first one contextualizes the work and presents the theoretical basis needed for the comprehension of the text. Chapter 2 presents the related works studied during the literature review. Chapter 3 presents the main concepts of Model Checking, such as the representation of the finite state machine, temporal logics, the state-space explosion problem and symbolic representation. Chapter 4 describes the techniques of model reduction used in this work; first it describes the concepts of abstraction; following it presents the concepts of group theory and used it to define symmetry reduction.

The second part presents the performed activities, results, conclusions and future work. Chapter 5 presents the methodology proposed in the present work and defines the modeling assumptions that are mandatory to generate abstract models based on

symmetry in a semiautomatic way. Chapter 6 describes aspects that were necessary to model the GridMedia network, such as the functioning of P2P networks and the model description language. It also describes the implementation of the model and its abstractions. Chapter 7 describes the computational resources used, presents the process of identifying the symmetry, three examples of model reduction and verification, and a quantitative comparison of the reductions. Chapter 8 presents the conclusions and lists possible directions to explore in future works.

Chapter 2

Related Work

The use of abstractions has been studied for a long time in Computer Science. Cousot and Cousot [1977] were pioneer in proposing the concept of abstract interpretations to generate partial and/or inaccurate information about program structures to be used by compilers. The first work in Model Checking that used abstractions is considerably more recent and was based on the same concept of abstract interpretations. Clarke et al. [1994] shows a method to create abstract models and uses the $\forall CTL*$ logic, a subset of the $CTL*$ logic introduced by Grumberg and Long [1991], to ensure the correctness of the results in model checking with over-approximation abstractions. [Lee et al., 1996] extends the technique to the verification of reduced models with under-approximation abstractions and introduces the $\exists CTL$ logic, a subset of the CTL logic that ensures the correctness of the results for this kind of abstraction. [Clarke et al., 2003] focuses on over-approximation abstractions again by proposing a technique for the automatic adoption of abstractions that uses the temporal logic $\forall CTL$, subset of $\forall CTL*$ logic, to ensure the correctness of the verification. In case of inconclusive results, the technique performs successive automatic refinements in the model. More recently Drager et al. [2009] uses the consolidated techniques of abstractions with Directed Model Checking, which is a new approach to minimize the problem of the state-space explosion where the exploration of the graph is oriented towards states that are considered special.

The exploitation of the symmetry in models was proposed at the same time by Ip and Dill [1993] and Emerson et al. [1993]. Both works use the concept of permutations over the states of a model to determine its symmetry. The first one focuses on the identification of symmetry through the use of special languages. The second one focuses on the analysis of models with many isomorphic components and proposes the reduction by grouping states that have the same valuation for symmetrical variables. Next, Jha [1996] extended the use of permutations over the set of states and formalized the

theoretical framework for the symmetry, based on Group Theory. Clarke et al. [1996] combines the reduction of symmetry with the symbolic model checking, and shows how the technique impacts the size of the symbolic representation using BDDs. Sistla et al. [2000] proposes a semiautomatic way to discover the symmetry in the model by using a special modeling language. It also defines the temporal logic $SCTL$, a subset of CTL that contains only formulas that do not break the model symmetry. Finally, Miller et al. [2006] present a compilation of the most important articles about symmetry reduction.

Some works merge the model reductions techniques to overcome the state-space explosion. Emerson and Trefler [1999] combined the concepts of symmetry and over-approximation abstractions to check properties that do not preserve the symmetry of the models and used the $\forall CTL$ logic. Sistla and Godefroid [2004] continues and extends the previous approach by allowing the verification of $CTL*$ formulas, even on models with low degree of symmetry. [Barner and Grumberg, 2005] combines under-approximation abstractions and symmetry reduction, focusing on the falsification of properties. The technique is applied to On-the-fly Model Checking, which is an approach where the model representation is constructed while the checking computations happen. This work also introduces the concept of maximal boolean subformulas, which are boolean formulas that preserve the symmetry of the model and allow larger reductions.

Finally, I mention that although there are several works of formal verification of network protocols, there are few that focus on P2P networks. We found only two during literature review. [Borgström et al., 2004] and [Bakhshi and Gurov, 2007] use theorem proving to verify a distributed database over P2P. We did not find any work about formal verification of P2P Live Streaming Systems or about the verification of P2P networks using Model Checking, which brings a pioneering aspect to this work.

Chapter 3

Model Checking

The task of Model Checking can be described as, given a model M that represents the system being verified, and a formal specification ϕ, find out which states from set S of reachable states satisfy ϕ. Formally, $\{s \in S | M, s \models \phi\}$.

The following sections present the definition of the Kripke structure, which is a mathematical model used to represent the system being verified. Then we introduce the concept of temporal logics, which are formalisms used to represent the formal specifications to be verified. Following, we formalize the problem of state-space explosion. Finally, we describe the technique of symbolic representation, which uses special data structures to represent the model in a compact way.

3.1 Kripke Structure

Kripke structure is a type of non-deterministic finite state machine used by Model Checking to describe the systems to be verified. It is represented as a directed graph where the vertices represent states of the model, and the edges represent transitions between states. A mapping function labels which properties are true in each state. Paths along the Kripke structure model computations of a system. Formally, a Kripke structure is defined as below:

Let AP be a set of atomic propositions. A Kripke structure M over AP is a 4-tuple $M = (S, S_0, R, L)$, where:

- S is the finite set of states
- $S_0 \subseteq S$ is the set of initial states

- $R \subseteq S \times S$ is the transition relation, which must be total, i.e., for each state $s \in S$, there shall exist a state $s' \in S$ such that $R(s, s')$.

- $L : S \longrightarrow 2^{AP}$ is the function that maps each state to the subset of atomic propositions that are true on it.

A **path** in a Kripke structure M, beginning from a state $s_0 \in S$, is an infinite sequence of states $\pi = s_0 s_1 s_2 ...$ such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. The fact that the transition relation is total, i.e., there are no states that have no outgoing transitions, makes the construction of infinite paths always possible.

3.2 Temporal Logics

Temporal logic is the term used to describe any system of rules and symbols used to represent and reason about propositions qualified in terms of time. It is used in Model Checking to describe sequences of transitions between states of a model. Time is not explicitly referenced in temporal logics. Instead, it presents operators such as "may", "always" or "never", which combined with boolean operators allows to verify if certain conditions occur at some point along the time.

All temporal logics allow reasoning about time lines. There are two types of logics in Model Checking based on this concept: linear time logics and branching time logics. The first one describes events over a single time line, and the second one describes events over multiple time lines. Linear Time Logic(LTL) is the best known logic of linear time. It was initially proposed by Pnueli [1977]. Computation Tree Logic (CTL) is the main logic of branching time and was initially proposed by Clarke and Emerson [1982]. Both logics are subsets of temporal logic $CTL*$ and differ by the operators they have. It is proved [Emerson and Halpern, 1986] that each one of these logics has a distinct capacity of expressiveness. The CTL logic has been chosen for this work because it has the existential path quantifier operator E, necessary to verify abstract models by under-approximation. In addition, the main related work in the area utilize it fully, or a subset of it Emerson et al. [1993]Clarke et al. [1994]Jha [1996].

3.2.1 CTL

Conceptually, the CTL formulas describe properties of *computation trees*. The tree is generated by selecting a state $s_0 \in S$ of the Kripke structure as the initial state. Then we expand the structure into an infinite tree with s_0 as root, as in 3.1.

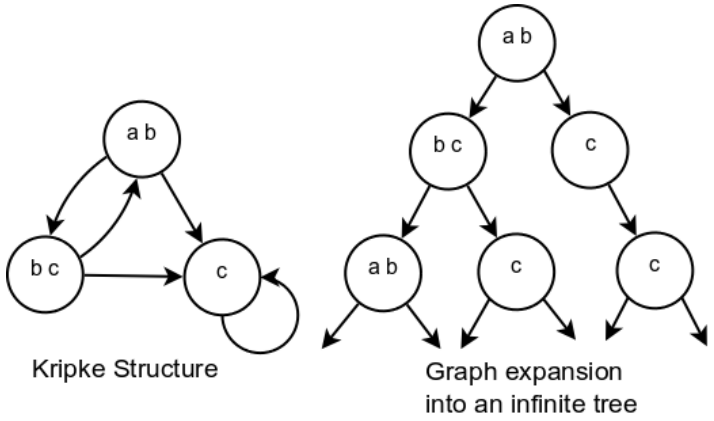

Figure 3.1. Expansion of a Kripke Structure into an infinite tree

Formulas in CTL are composed of CTL operators and propositional formulas. CTL operators are formed by the combination of a path quantifier and a temporal operator. **Path quantifiers** describe the branched structure of the computation tree; it specifies if the properties occur in all paths or at some paths. The **temporal operators** describe the linear structure of a path in the infinite tree.

The path quantifiers are:

- A ϕ - All: Defines that ϕ has to occur on all paths starting from the current state.
- E ϕ - Exists: Defines that ϕ has to occur in at least one path stating from the current state.

The temporal operators are:

- G ϕ - Globally: ϕ has to be true in all states along the path.
- F ϕ - Finally: ϕ has to be true in at least on state along the path.
- X ϕ - Next: ϕ must be true in the state following the current state.
- ϕ U ψ - Until: ϕ must be true until ψ is true. Implies that ψ will occur in the future.
- ϕ W ψ - Weak until: ϕ must be true until ψ is true, but does not imply that ψ occurs.

Figure 3.2 shows examples of infinite trees where the CTL formulas $AG\ p$, $AF\ p$, $EG\ p$ and $EF\ p$ are true and p is a proposition formula.

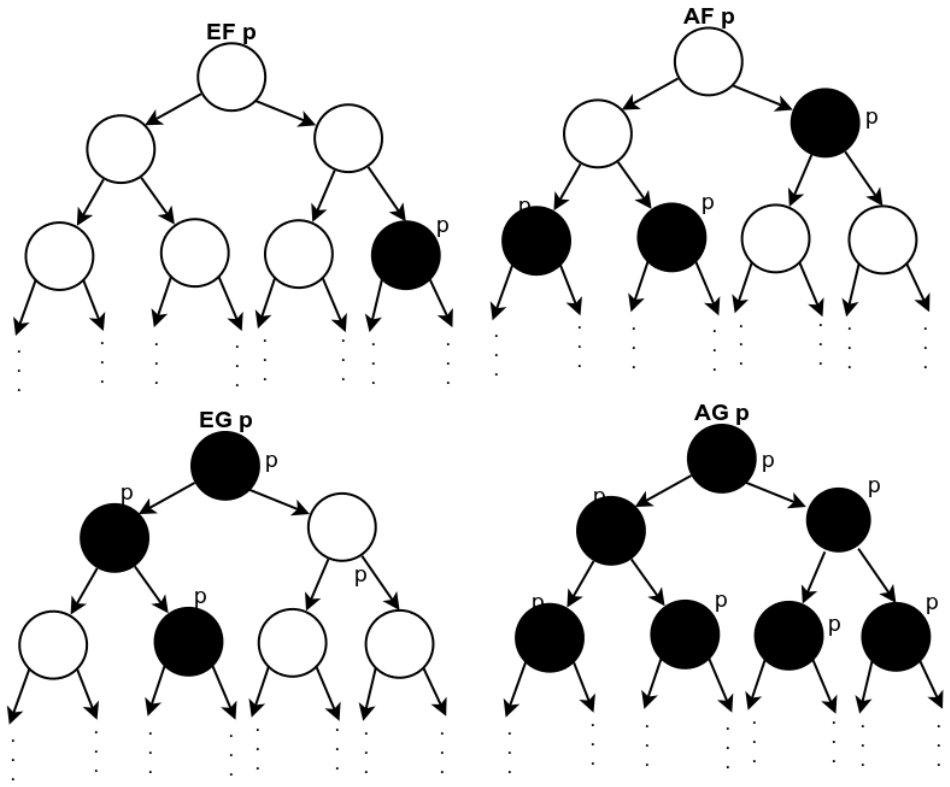

Figure 3.2. *CTL* operators over a property p

3.3 The state-space explosion

The main challenge in Model Checking is the **state-space explosion**: as the model complexity increases, characterized by the number of variables on the model, the number of states of the Kripke structure grows exponentially with respect to the number of variables. For example, a system composed of four processes, each with three variables that can assume five different values, results in a state-space of approximately 250 million states. The problem is aggravated in the verification of concurrent systems because the possible interleaves between asynchronous processes generate a gigantic amount of transitions between states.

We can define the problem of state-space explosion formally through the variables that compose the model. Let $V = \{v_1, v_2, v_3, ..., v_n\}$ be the set of variables of a model and D be the finite set representing the domain of V. A **valuation** of V is a function that associates a value in D into each variable $v \in V$. Formally, $s : V \rightarrow D, \forall v \in V$. A state of the model is defined as a unique valuation of all variables $v \in V$. If we consider a model where all the variables belonging in the set V have the same domain D, it is easy to see that the total number of states that the model will have is $|D|^{|V|}$. That is, the number of possible states is exponential to the number of variables $|V|$. In

general, the number of possible states is given by $\prod_{v_i \in V} |D_i|$, where $|D_i|$ is the number of elements from the domain of the variable v_i.

The main studies on Model Checking research ways of trying to circumvent the state-space explosion. Among the most important ones, there is the Symbolic Model Checking, which presents compact ways to represent the transition relation using propositional formulas. It is detailed in section 3.4. Other techniques target the reduction of the model and consequently the reduction of the states and transitions to be considered. One of them is the use of abstractions, which consists of removing or merging states irrelevant to the property being verified. Another one is the symmetry reduction, which groups symmetrical states. Both are used in the present work and are detailed in chapter 4.

3.4 Symbolic Model Checking

Symbolic Model Checking is the name given to the techniques where the graph representing the model is not constructed explicitly, i.e., not described by adjacency lists. McMillan [1992] started the symbolic representation proposing the description of the transition relation between the states of the Kripke structure using first-order propositional formulas. The implementation and handling of these formulas is made using BDDs, which are data structures that discard redundant information in the representation of boolean formulas and have efficient algorithms for their manipulation. It enabled a compact representation of the model without limiting its verification. This single contribution elevated at once the size of verifiable models from the magnitude of 10^{10} to the magnitude of 10^{20} states [Burch et al., 1990].

In this section we present the symbolic representation using BDDs. The technique was chosen for this work because the major related works in the area also use it. However, the reductions presented in this work are independent of the type of representation adopted and can easily be extended to explicit representation or symbolic representation using SAT solvers, which is a symbolic technique that uses propositional satisfiability solvers to evaluate models.

3.4.1 Symbolic Representation

As mentioned in section 3.3, a valuation over the domain D of the set of variables V determines a state of the model. Given a valuation, we can write a formula that is exactly true for that valuation. For example, let $V = \{v_1, v_2, v_3\}$ and the valuation

$(v_1 \leftarrow 4, v_2 \leftarrow 5, v_4 \leftarrow 7)$. We can derive the formula $(v_1 = 4) \wedge (v_2 = 5) \wedge (v_3 = 7)$, which represents the state of the model where the variables have exactly those values.

Besides states, the transitions relation set can also be represented by first-order formulas. The idea presented above is extended and we represent a transition as an ordered pair of states. First we introduce a second set V' where for all $v_i \in V$ there is a corresponding variable $v'_i \in V'$ with the same domain $D_i = D'_i$. Each valuation over V is considered a current state and each valuation over V' determines a next state. A pair formed by a valuation of V and a valuation of V' can be seen as a transition and we can represent it using formulas. The formula $(v_1 = 4) \wedge (v_2 = 5) \wedge (v_3 = 7) \wedge (v'_1 = 1) \wedge (v'_2 = 2) \wedge (v'_3 = 0)$ is an example of a transition represented by a propositional formula.

To specify properties of the system, we must define a set of atomic propositions AP, which has the form $v = d$, where $v \in V$ and $d \in D$. A formula that represents the subset of atomic propositions that are true in a state s is a conjunction of these propositions.

Thus we can derive a Kripke structure $M = (S, S_0, R, L)$ from first-order formulas that describe the system as below:

- The set of variables S are all possible combinations of values (valuation) of the variables $v_i \in V$, where each combination is represented by a first-order formula, such as $(v_1 = value_1 \wedge v_2 = value_2 \wedge v_3 = value_3 \wedge ... \wedge v_i = value_i)$.
- The set of initial states S_0 is a set of first-order formulas that meet a property ρ_0.
- The set of transitions R is composed of first-order formulas that map to true when the state $s \in S$ and the next state $s \ \in S'$ has its first-order formulas also true.
- The set L maps each state to a first-order formula that contains a conjunction of atomic propositions true in that state.

The above definitions contain predicates of the various theories and must be converted to pure boolean first-order logic. For example, the formula $(v_1 = 4) \wedge (v_2 = 5) \wedge (v_3 = 7)$ contains predicates of the Number Theory for integers. The conversion is done by calculating the number j of bits needed to encode the D domain of each variable v_i in binary representation and partitioning v_i into j boolean variables. For example, suppose that the v_i variables assume only integer values ranging from 0 to 7, i.e., $D = \{0, 1, 2, 3, 4, 5, 6, 7\}$. It takes three bits to encode each variable v_i.

Thus, the predicate $(v_2 = 5)$ can be replaced by the conjunction of three new boolean variables $(v_{2.1} = 1 \wedge v_{2.2} = 0 \wedge v_{2.3} = 1)$, where each variable represents a bit of binary representation of 5.

To illustrate the symbolic representation of a Kripke structure using first-order formula, considers the two-state structure shown in figure 3.3. This example has two boolean state variables, a and b. We introduce two new variables a' and b' to encode the successor states. Thus we represent the transition from state s_1 to s_2 as the conjunction $(a \wedge b \wedge a' \wedge \neg b')$. The boolean formula that represents the complete transition relation contains three disjunctions, representing the three transitions that the Kripke structure has. It is represented as below:

$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$

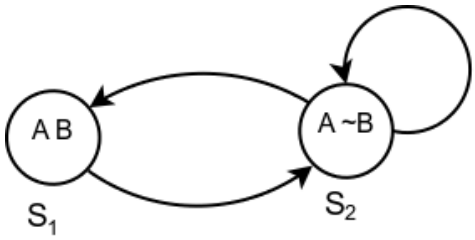

Figure 3.3. Two-states Kripke structure

3.4.2 BDDs

Binary Decision Diagram (BDD) is a compact way of representing boolean formulas which usually are smaller than the traditional conjunctive normal form (CNF) and the disjunctive normal formal (DNF). Its origin comes from the representation of boolean functions using binary decision trees, which is a directed tree with two types of vertices: terminals and non-terminals. Non-terminal vertices are labeled by a boolean variable $var(v)$ and have two successors: $zero(v)$ when $v = 0$ and $one(v)$ when $v = 1$. Terminal vertices are labeled 0 or 1, according to the value of function $valor(v)$. Figure 3.4 shows an example of a binary decision tree for a 2-bit comparator, given by the boolean formula $f(A1, A2, B1, B2) = (A1 \leftrightarrow B1) \wedge (A1 \leftrightarrow B1)$.

A walk through the tree beginning from the root vertex, and the choice of $zero(v)$ or $one(v)$ for each variable $var(v)$ will lead to a terminal vertex that represents the evaluation of the boolean function. In the example of the 2-bit comparator, the attribution $(A1 := 1, A2 := 0, B1 := 1, B2 := 1)$ leads to the terminal labeled with 0. Thus, the expression is false for this assignment.

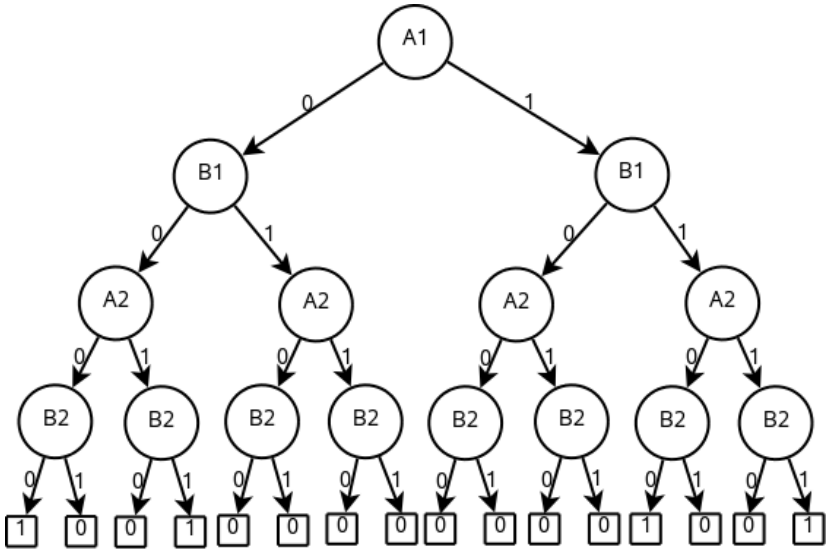

Figure 3.4. Binary decision tree for the 2-bit comparator

The binary decision trees have a lot of redundant information. In the example in Figure 3.4, there are eight subtrees labeled by $var(v) = B2$, but only three are distinct. If we group the isomorphic subtrees it is possible to get a more concise representation of the boolean formula. This will result in a directed acyclic graph called binary decision diagram.

In practice it is desirable to obtain a canonical representation for boolean functions. Such representation must have the property that two functions are logically equivalent if and only if they have an isomorphic representation. Two BDDs are isomorphic if there is an injective function h that maps terminals and non-terminal to one another. Bryant [1986] presented a method to obtain the canonical representation of boolean functions by placing restrictions on two BDDs. First, the variables must always appear in the same order along all paths starting from the root to a terminal. Second, there may not occur isomorphic subtrees or redundant vertices in the diagram.

The first part is obtained by fixing a total ordering for the variables ($var_1 < var_2 < var_3 < ... < var_i$), i.e., if a vertex u has a non-terminal successor v, then the variable u precedes the variable v in the ordering ($var(u) < var(v)$). The second part is obtained by successively applying the three transformation rules below in the diagram, which does not change the represented boolean function:

- Remove duplicate terminals: Keep only one terminal representing the 0 terminal and one terminal representing the 1. Redirect incoming edges to one of these.

- Remove duplicated non-terminals: If two non-terminal u and v represent the same

variable ($var(u) = var(v)$), and lead to the same vertices ($zero(u) = zero(v) \wedge um(u) = um(v)$), eliminate v and redirect incoming edges to u.

- Remove redundant tests: If a non-terminal v has $zero(v) = one(v)$, then delete v and redirect all incoming edges to the vertex indicated by $zero(v)$.

The canonical form is obtained stating with a BDD that meets the defined ordering and then applying the transformation rules until the diagram can no longer be reduced. The term ordered binary decision diagram (OBDD) describes the graphs obtained in this way. Figure 3.5 shows the OBDD for the boolean function in the example of a 2-bit comparator, considering the order $A1 < B1 < A2 < B2$.

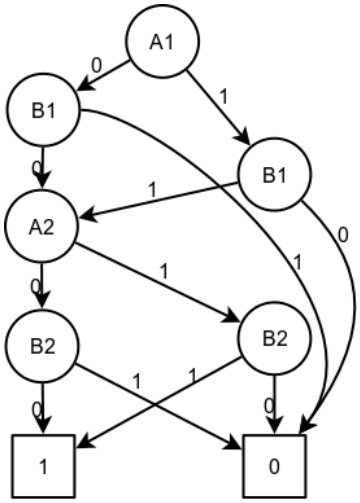

Figure 3.5. OBDD for the 2-bit comparator

The size of the OBDD depends critically on the variables ordering. Using the same example of the two-bit comparator, we obtain the OBDD in figure 3.6 for the order $A1 < A2 < A3 < A4$. This OBDD has eleven vertices, while the OBDD in figure 3.5 has eight. In case of a n-bit comparator it can be shown that if the variables are ordered such as $A1 < B1 < A2 < B2 < ... < An < Bn$ the number of vertices of the OBDD would be $3n + 2$, but if we choose the order $A1 < A2 < ... < An < ... < B1 < B2 < ... < Bn$ the number of vertices will be $3.2^n - 1$.

In general, finding the optimal ordering for the variables is not feasible. Bryant [1992] showed that the problem of determining whether an ordering is optimal is NP-Complete. Moreover, there are boolean functions that have exponential size for the OBDD for all possible variable orderings. However empirical observation has shown that OBDDs tend to be smaller when related variables are placed near. Several heuristics use this observation to generate orderings that lead to smaller OBDDs.

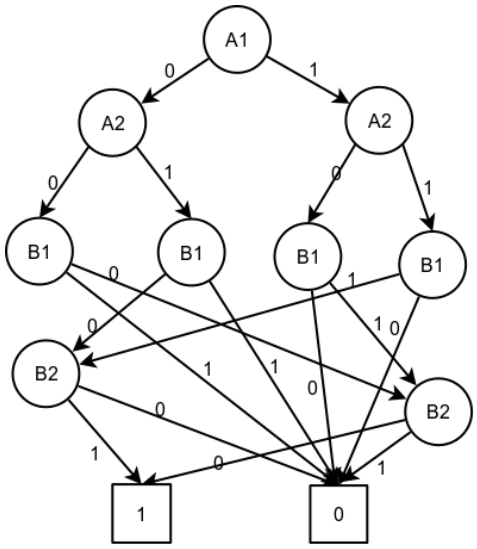

Figure 3.6. OBDD for the 2-bit comparator. Ordering: $A_1 < A_2 < B_1 < B_2$

Chapter 4

Model Reduction

This chapter presents the techniques explored in this work for reducing models. The first part shows the reduction using abstractions, which removes parts of the model that are considered irrelevant to the property being verified. There are types of abstractions: over-approximation, where states and transitions are grouped, and under-approximation, where states and transitions are discarded. We also show how the information loss impacts the properties that can be verified for each of these types of abstractions. More specifically we show the temporal operators that can be used in each case without interfering in the results of verification.

The second part describes the method of model reduction by symmetry. It shows how to identify the symmetry between the states and components, and how to determine if a formula in temporal logic can be checked in the reduced model. The symmetry reduction can be seen as a special case of abstraction by over-approximation as it groups states and transitions, but it does not limit the temporal operators that a verifiable formula can have.

4.1 Abstractions in Model Checking

The use of abstractions simplifies the original model removing or merging behaviors that are irrelevant to the property being checked. Although there are attempts to automate the implementation of abstractions on the models, as in [Clarke et al., 2003], typically the task is manual and requires creativity and intuition, which adds a new possible error factor. Moreover, the information loss caused by the abstractions on the model can lead to incorrect results on the verification.

There are two basic abstractions methods, which differ in the way they control the errors brought by the information loss. The first one is over-approximation,

where states and transitions are grouped. In an abstract model created by over-approximation, all the previously existing paths from the original model remain; however new invalid paths are created. The other method is abstractions by under-approximation, where states and transitions are eliminated. In this approach some existing paths in the original model no longer exist in the abstract model; however all paths in the abstract model necessarily exist in the original model.

4.1.1 Over-approximation Abstractions

Abstractions by **over-approximation** groups states and transitions. Intuitively, an abstract model $\widehat{M}$ is generated by partitioning the set of states S of the original model M, and each partition will be a state in the abstract states set $\widehat{S}$. The definition of which states are in each partition are given by the abstraction function $h : S \to \widehat{S}$, which is a surjection. Formally, an abstract Kripke structure $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L})$ with respect to an abstraction function h is defined as below:

- $\widehat{S}$: for all $s \in S$, $h(s) \in \widehat{S}$.
- $\widehat{S_0}$: if $s \in S_0$ then $h(s) \in \widehat{S_0}$.
- $\widehat{R}$: if $(s, s') \in R$ then $(h(s), h(s')) \in \widehat{R}$.
- $\widehat{L} = \bigcup_{h(s)=\widehat{s}} L(s)$.

Figure 4.1 shows the example of generation of an abstract model by over-approximation. The first graph in the figure represents the original model, which has seven states. The dashed lines delimit three partitions on the set of states S. The partitions generate a set of abstract states $\widehat{S} = \{(A, B), (C, D, E), (E, F)\}$. The abstraction function h is defined as (A, B) for $h(A)$ and $h(B)$; (C, D, E) for $h(C)$, $h(D)$ and $h(E)$; and (F, G) for $h(f)$ and $h(g)$. The second graph in the figure represents the abstract model generated by the partitions, where all the transitions in R have some correspondent in $\widehat{R}$.

It is easy to see that all states and all transitions at the original model have a match in the abstract model. Therefore the abstract model contains all possible paths in the original model. However the grouping of states and transitions generate paths that do not exist in the original model. For example, the path $\pi_1 = A, G, F, ...$ is absent in the original model but exists in the abstract model. Another example is the path $\pi_2 = B, D, E, ...$ that exists on the abstract model and contains the state E, which is unreachable in the original model.

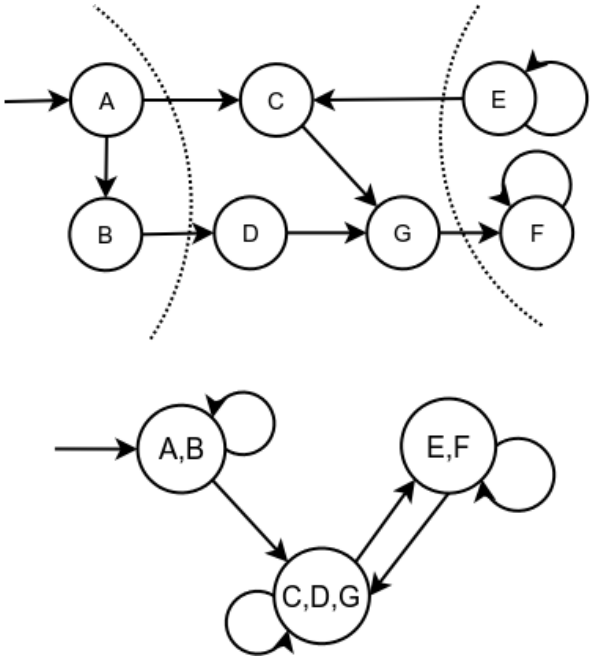

Figure 4.1. Generation of the Abstract Model by Over-Approximation

The verification of abstract models with over-approximations can be made using the temporal logic $\forall CTL$, which is a subset of CTL that contains only operators with the universal path quantifier A and do not accept the negation of subformulas that contain temporal operators. If a property ϕ in $\forall CTL$ is true on the abstract model, it will also necessarily be true in the original model [Clarke et al., 1994]. Intuitively, this conclusion is true because the verification of ϕ was satisfied for all paths on the abstract model. The fact that the abstract model has all paths from the original model, plus the invalid paths generated by the over-approximations implies that the property is satisfied for all paths at the original model.

For similar reason, if the verification of ϕ is false, the counterexample provided by the model checker may be invalid. After all, the counterexamples is a path along the abstract Kripke structure $\widehat{M}$. Thus the path can either be an existing path in the original model or an invalid path generated by the over-approximation abstractions.

4.1.2 Under-approximation Abstractions

The method of abstractions by **under-approximation** systematically eliminates states and transitions. It creates abstract models which, although have smaller number of behaviors, do not admit behaviors that do not exists in the original model. Unlike the over-approximation abstractions, there is no correspondence between all states on the original in the abstract model. That is, there is not an abstract function h that

maps each state of S in $\widehat{S}$. Instead of a function, the definition of the abstractions is done by the subsets $F \subseteq S$ of states and $E \subseteq R$ of edges, which are eliminated. Formally, an abstract Kripke structure $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L})$ with respect to the set of states F and set of edges E is defined as below:

- $\widehat{S} = S - F$.
- $\widehat{S_0} = S_0 - (S_0 \cap F)$.
- $\widehat{R} = R - E$.
- $\widehat{L} = L$.

Figure 4.2 shows an example of abstraction by under-approximation using the same original model of section 4.1.1. The set of removed states is $F = \{B, D\}$ and the set of removed edges is $E = \{(A, B), (B, D), (D, G)\}$. The generated abstract model is shown by the second graph on the figure. The path $\pi = A, C, G, F$ is highlighted. It exists in the original model and is preserved in the abstract model.

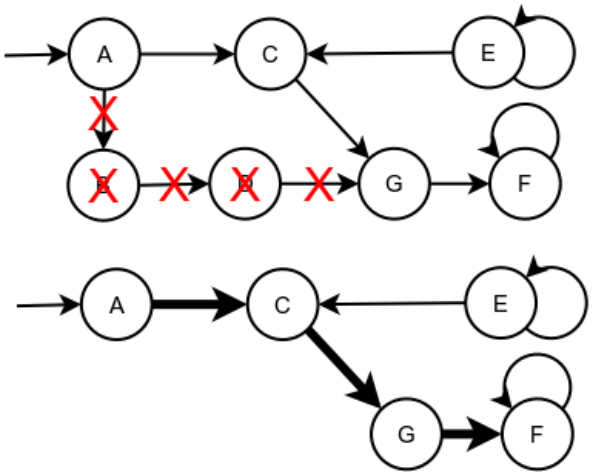

Figure 4.2. Abstract model by under-approximation

It is trivial to notice that, in contrast to over-approximations, not all paths in the original model also exist in the abstract models with under-approximations. Conversely, if a path exists in an abstract model it necessarily also exists in the original model. Lee et al. [1996] introduced the $\exists CTL$ logic, a subset of CTL that contains only operators which have the existential path quantifier E and do not accept negation of subformulas that contain temporal operators. It guarantees the accuracy of model checking with abstractions by under-approximation. Intuitively, if the verification of a $\exists CTL$ formula ϕ proves to be correct it implies that the verification found at least one

path in the abstract model that satisfies ϕ, and all the paths in an abstract model also exist in the original model.

4.2 Symmetry in Model Checking

The symmetry reduction relies on the concepts of permutations over finite sets combined with Group Theory. It is based on the observation that the existence of symmetry implies that there are sets of equivalent states. Those states preserve both the transition relation R and the labeling function L of atomic propositions when swapped. The presence of sets of equivalent states allows the creation of reduced models, called quotient models, choosing only one representative from each set.

Besides the fact that the quotient model is smaller, it is proved that the verification of any $CTL*$ property over it implies an identical result to verification of the same property on the original model. The symmetry reduction can be seen as a special case of abstraction by over-approximation, since the equivalent states are collapsed in the representative state, but where the information loss does not limit the temporal logic that can be used.

4.2.1 Permutation Groups

The use of symmetry in Model Checking is associated with concepts of Group Theory and permutation that are presented now. A **permutation** σ over a set A is a bijective function defined as $\sigma : A \to A$. The identity permutation e is the one that maps each element of the set A into itself. A permutation like $a_1 \mapsto a_2, a_2 \mapsto a_3, ..., a_{k-1} \mapsto a_k, a_k \mapsto a_1$ is called a cycle and is written as $(a_1 a_2 a_3 ... a_k)$. Any permutation can be written as the functional composition of disjoint cycles[Lane and Birkhoff, 1988]. Recall that functional composition is the application of the output of a function as input to another. All the papers about symmetry in Model Checking represent the functional composition by the concatenation operation, and the present work follows this convention. Therefore, $f(g(x)) = f \circ g = fg$.

A **permutation group** G is a set of permutations associated with the operation of functional composition, such that:

- The identity permutation $e \in G$.
- For every permutation $\sigma \in G$ exists a permutation $\sigma^{-1} \in G$ such that $\sigma\sigma^{-1} = e$.
- For all permutations $\sigma_1, \sigma_2 \in G$, $\sigma_1\sigma_2$ is also in G.

The **closure** of a set under any operation is defined as the smallest superset containing all elements resulting from the application of the operation for all elements in the set. For example, the closure of the set $\mathbb{N}$ of the natural numbers under the subtraction operation is the set $\mathbb{Z}$ of integers. The permutations $\sigma_1, \sigma_2, \sigma_3, ..., \sigma_k$ are said to be the **generators** of a permutation group G, if G is the closure of the set $\{\sigma_1, \sigma_2, \sigma_3, ..., \sigma_k\}$ under the operation of functional composition. Formally, $G = \langle(\sigma_1, \sigma_2, \sigma_3, ..., \sigma_k)\rangle$.

In this paper we use permutations over the finite set of states S of the Kripke structure M that represents the system. However a permutation can also be stated in terms of the components of the model. Let p_i and p_j be two identical components of a system. The function $\sigma(p_i) = p_j$ indicates the permutation of all the states where the variables of p_i and p_j have the same values. The permutation function σ can also be used over a first-order boolean formula β to indicate the equivalent function after permutation.

A permutation σ that preserves the transition relation and the initial states of a Kripke structure M is called an **automorphism**. If all the permutations of the permutation group G are automorphisms of M, then G is also a **symmetry group** (or automorphism group). Formally, $\forall s, s' \in S, \forall \sigma \in G[(s, s') \in R \iff (\sigma(s), \sigma(s')) \in R \wedge s \in S_0 \iff \sigma(s) \in S_0]$.

We exemplify the presented definitions through the Kripke structure M of a system with three identical processes: p_1, p_2 and p_3. Each process p_i has a set of variables $\bar{V_i}$ and each state of the system is a valuation over the set of variables $\bar{V} = \bar{V_1}\bar{V_2}\bar{V_3}$. A permutation that exchanges the values of $\bar{V_i}$ and $\bar{V_j}$, i.e., exchange states where the valuation of each variable of a process is identical to the same valuation of the variables of the other process is written as $(p_i p_j)$. These permutations maintain the transition relation and initial states since the processes are identical. Thus the group $G = \{e, (p_1p_2), (p_2p_3), (p_1p_3), (p_1p_3p_2), (p_1p_2p_3)\}$ is a symmetry group of M and the permutations $\{(p_1p_2), (p_2p_3)\}$ are the set of generators of G. Finally, if a permutation exchanges the values of p_1 and p_2, a boolean formula $\beta = (p_1.state = critical)$ is also mapped to the equivalent formula after applying the function $\sigma(\beta) = (p_2.state = critical)$.

4.2.2 Quotient Model

Given a symmetry group G and a Kripke structure M, we can partition the set of states S into equivalence classes called orbits. Formally, the **orbit** of a state s is a set of states $\theta(s) = \{t | \exists \sigma \in G$, where $\sigma(s) = t\}$. Thus, we can generate a reduced model

from the original model by selecting one **representative** from each orbit θ, which we call $rep(\theta(s))$. This reduced model is known as model quotient.

Formally, the quotient model of a Kripke structure M, with respect to a symmetry group G is defined as $M_G = (S_G, S_G^0, R_G, L_G)$, where:

- $S_G = \{\theta(s) | s \in S\}$ is the set of orbits of the state in S.
- $S_G^0 = \{\theta(s) | s \in S_0 \wedge s = rep(\theta(s))\}$.
- $R_G = \{(\theta(s), \theta(s')) | (s, s') \in R\}$.
- $L_G(\theta(s)) = L(rep(\theta(s))$.

The fact that G is a symmetry group makes all initial states $s \in S_0$ at M have a representative of its orbit also in S_G^0. For the same reason R_G is well-defined and independent of the chosen representatives for each orbit. In contrast, the definition of L_G is dependent on the choice of representative states of each orbit, as atomic propositions can be true in one state but not at another.

To avoid problems related to the choice of representatives, the generation of quotients models is restricted to symmetry groups that are also **invariance groups**. In addition to the transition relation and initial states, the permutations of these groups preserve the set of propositions that are true in each state. Formally, let BS be a set of formulas that represent propositional formulas over the states of the model. A symmetry group G with respect to a Kripke structure M, is also a invariance group with respect to BS if for each $\sigma \in G$, and each $s \in S$, and each $\beta \in BS$, $\beta \in L(s) \iff \beta \in L(\sigma(s))$.

A proposition $\beta \in BS$ is called an **invariant** over G and is said to preserve the symmetry of a permutation σ if for every $s \in S$, $[M, s \models \beta \iff M, \sigma(s) \models \beta]$; conversely, β breaks the symmetry of a permutation σ if β does not preserve the symmetry in σ.

In [Emerson et al., 1993][Jha, 1996][E M Clarke, 1999] it was proved that if a specification ϕ in $CTL*$ contains only invariant propositions, then the result of the verification of ϕ over the quotient model is also valid for the original model. Formally, $M \models \phi \iff M_G, \models \phi$. Thus, the formula ϕ to verified must contain only invariant propositions since the set BS is build of the propositions in ϕ.

Barner and Grumberg [2005] shows that atomic propositions break the symmetry of permutations and proposes the generation of the BS by extracting the **maximal boolean subformulas** from ϕ. Formally, a formula β is a maximal boolean subformula of a temporal formula ϕ if β is a boolean subformula of ϕ, and for all subformulas β'

of ϕ, if β is a subformula of β' it implies that β' is not boolean. For example, the temporal formula $\phi = AG\ (p1.state = req \rightarrow AF\ p1.state = req\ \&\ p2.state = idle)$ generates the set BS containing two maximal boolean subformulas: $(p1.state = req)$ and $(p1.state = req\ \&\ p2.state = idle)$.

4.2.3 Example

The illustration of the definitions presented is done using the Kripke structure of a token ring network consisting of two identical processes. The internal representation of the state machine of the processes is shown in figure 4.3. Each process has three states: n, while it is awaiting the arrival of the token; t, when it holds the token, but is not in the critical section; and c, when it holds the token and is at the critical section. The actions r and e indicate respectively the reception or delivery of the token to another process.

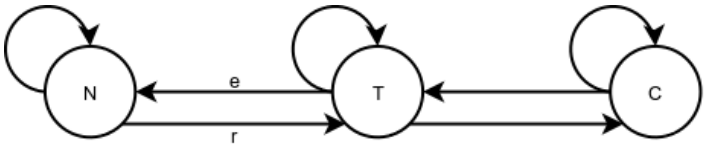

Figure 4.3. Representation of the state machine of a process p_i

In this example the network has two participating processes, p_1 and p_2. The Kripke structure resulting from the composition $p_1||p_2$ is shown in figure 4.4. Each of the four reachable states is determined by a valuation of the state variables v_i of each process, whose domain D is $\{N, T, C\}$. For the sake of simplification, all states s are also initial states.

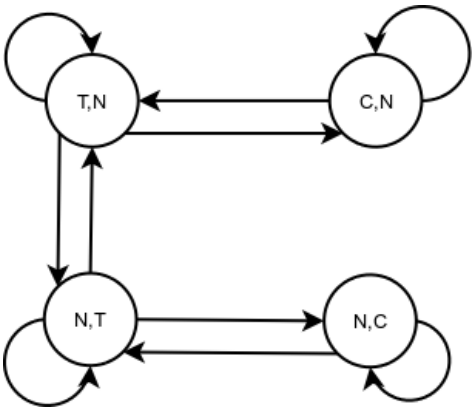

Figure 4.4. Kripke structure for $p_1||p_2$

A permutation that exchanges the state variable of a process for the equivalent variables of another can be written using the notation presented in section 4.2.1 as (p_1p_2). We can easily verify that (p_1p_2) is an automorphism of M. For example, there is a transition (T, N) to (C, N) and also one from $\sigma((T, N))$ to $\sigma((C, N))$. Any other transition at $p_1||p_2$ can be examined similarly.

Consider a symmetry group G where the only generator is (p_1p_2). The orbits induced by $G = \langle(p_1p_2)\rangle$ are $\{(T, N), (N, T)\}$ and $\{(C, N), (N, C)\}$. If we choose the states (T, N) and (C, N) as the representatives of each orbit, we have a quotient model with two states, as in figure 4.5. Note that this choice can only be made because it maintains the transition relation.

Now we want to check if there is violation of mutual exclusion in the model, indicated by the CTL formula $\phi = AG(v_1 = C \rightarrow \neg v_2 = C) \wedge (v_2 = C \rightarrow \neg v_1 = C)$. In this example the set BS of propositions contains only the maximal boolean subformula $\beta = (v_1 = C \rightarrow \neg v_2 = C) \wedge (v_2 = C \rightarrow \neg v_1 = C)$ It's easy to see that β is true in all states exchanged by the permutations of G. That is, the symmetry group G is also an invariance group of M with respect to BS. Thus, based on the proof referenced in section 4.2.2, a $CTL*$ formula which contains only boolean subformulas from BS can be checked in the model without changing the result of verification. In this example, if the verification of ϕ over the quotient model is true, it guarantees that two processes will never be in the critical section simultaneously in the original model.

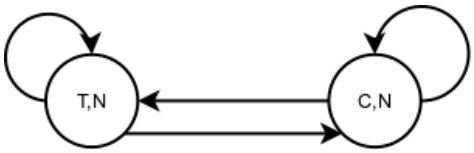

Figure 4.5. Quotient model for $p_1||p_2$

The presented example has only two processes, but can be extended easily to more. Consider the same token ring network with i being the number of processes, as in figure 4.3. The process p_k receives the token from the process p_{k-1} on the right and forwards it to the process p_{k+1} on its left. Figure 4.6 presents this network. Just like in the example with two processes, each state is determined by the valuation of the state variables of the Kripke structure generated by the composition $p_1||p_2||...||p_i$. The number of reachable states is $2 \times i$.

Let $G = \langle(p_1p_2...p_i)\rangle$ be the symmetry group with respect to $p_1||p_2||...||p_i$. As in the case of $p_1||p_2$ G generates two orbits:

$\{(T, N, ..., N), (N, T, .., N), ..., (N, N, ..., T)\}$ and $\{(C, N, ..., N), (N, C, .., N), ..., (N, N, ..., C)\}$.

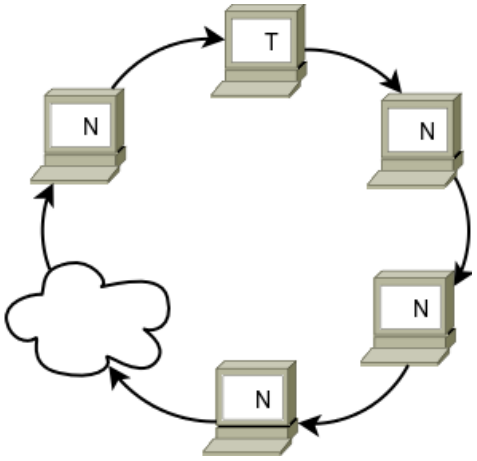

Figure 4.6. Token ring network represented by the $p_1||p_2||...||p_i$ model

Therefore, the quotient model for $p_1||p_2||...||p_i$ is identical to quotient model for $p_1||p_2$, shown in figure 4.5. This example clearly shows how the exploitation of symmetry can result in significantly smaller models.

Chapter 5

Methodology

The development of a methodology of semiautomatic reductions based on symmetry aims to avoid the computational work required by the automated techniques that remove states. It is semiautomatic as it needs human intervention to be defined, but does not require the edition of the description of the model to be implemented.

The methodology is divided in three parts: the modeling procedure, which defines the premises to make the use abstractions semiautomatic; the method of determining the symmetry in the model through the generation of symmetry groups; and the specification of properties in temporal logic that can be verified on the abstract model.

5.1 Types of model specification

Despite the results of this study be independent of the choice of description language of the state machine, there is the assumption that the chosen language shall provides the required semantic resources for the model description. In this case, the language should support the description of the model both by assignment and by direct specification. This is the case of SMV language, which is used in this work and that will be detailed in section 6.2.

The **specification by assignment** is the most common and intuitive method to define the model. On it, the reasoning is made over the values that the variables of the model will assume, not about the states that these values define. The semantics is similar to that of imperative languages such as C++, where variables are declared, initialized and assigned. Below there is a simple example of definition by assignment in SMV language. The variable `myint` is an integer and can assume values from zero to three. The first line defines that the `myint` will always be initialized with one or two. The second line defines that the value of a `myint` in the next states will always

be equal to the rest of the division of the increment of the current value by four.

```
init(myint) := 1,2;
next(myint) := (myint+1) mod 4;
```

Direct specifications reason about states and transitions defined by the values of variables, and not about how values are assigned. The definitions are enunciated through propositional formulas, and only the states where these formulas are true are valid. The example below, defined by direct specification, is equivalent to the example above, defined by assignments. The first line defines that the initial states will be only those states where the value of `myint` variable ranges from one to two. The second line defines that the only valid transitions between states are those where, if the value of `myint` is equal to three in the current state then in the next state it should be zero; and when its value is different from three then the value in the next state will the be increment of the current value.

```
INIT (myint < 3) & (myint > 0);
TRANS (myint != 3 -> (next(myint) = myint+1)) & (myint = 3 -> next(myint)=0);
```

The use of direct specifications should be made consciously. Its use allows the definition of inadmissible state machines, where the set of initial states may be empty, or where the transition relation is not complete (i.e., there are states without successors). The example below illustrates the two cases. The first line defines that only states where `myint` is different from `myint` will be initial states. Obviously such states do not exist, and the set of initial states is empty. The second line defines that only transitions where the value of `myint` in the current state is greater than `myint` are valid. Again, such states do not exist, and consequently there are no valid transitions.

```
INIT (myint != myint);
TRANS (myint > myint);
```

5.2 Modeling

This work applies to the verification of models of reactive systems that have symmetry in its description. This feature is present in models that have replication of its structures, i.e., contains several instances of the same types of components. Some examples

of highly symmetric systems are a bank of registers, a network with several identical participants or crawler robots of a search engine.

In all examples, the components are identical and share the same scheme of decision making. Moreover, each model has a self-organization of which components communicate with each other. The observation of this pattern of organization allows to divide the definition of models into three parts: representation of the types of components, called the internal representation; representation of how the components interact, defined by the communication interfaces; and the organization of which components communicate to each other, called the external representation. The diagram of figure 5.1 illustrates the role of each party in the model definition.

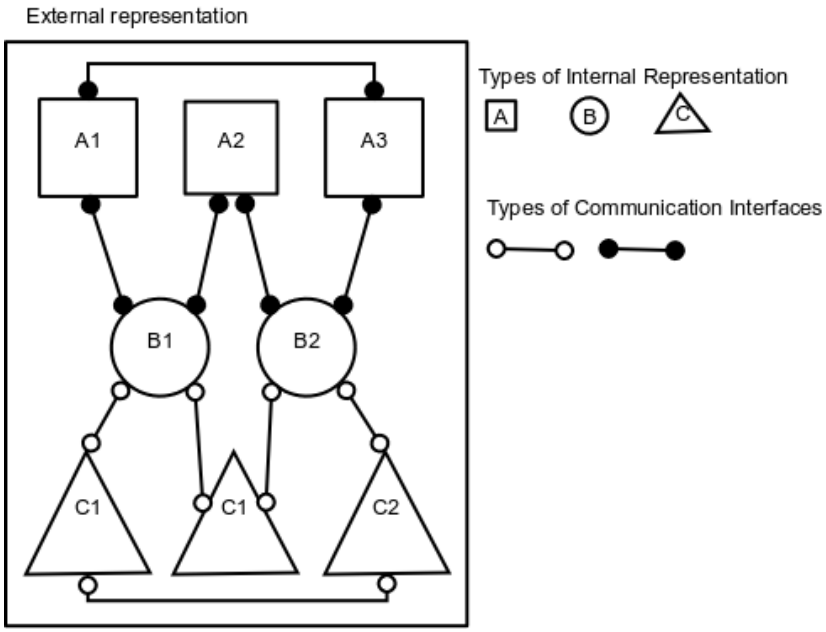

Figure 5.1. Scheme of the division the modeling

5.2.1 Internal Representation

The internal representation refers to the description of finite state machines of the types of components in the model. It contains the declarations of state variables and the definition of the transitions of the values of the variable, based on the next states and the current state.

Its definition is directly related to the concept of semiautomatic abstractions because the removal of a component from the model defines that the transitions of its state variables are always made to the same values. Section 5.4 the assumptions that the internal representation must meet to enable the semiautomatic abstractions.

The methodology of this paper defines the internal representation of the types of components has to be made through the method of assignment, using a modular structure. In the SMV language, for example, this is done using the primitive `MODULE`, which is similar to the declaration of a function in a programming language. Although it is possible to use direct specifications in this case, the method is not recommended because it is less intuitive. Moreover, the semiautomatic abstractions are defined by direct specifications. If the internal representation would also use direct specifications as well, the model would be less readable.

5.2.2 Communication Interfaces

Communication interfaces define how the components of the model interact. They are a common medium between the internal and external representation. Typically they are implemented by changing the value of a state variable that is common to the components that communicate. The input of a logic signal in a circuit or a communication message on a network are examples that illustrate the concept of interface.

The way the components interact depends on the system being represented. So, the methodology of this work only states that any communication is done through reading and writing to state variables that are visible to the components being considered. However it is recommended, if the representation of the system enables, that the modeling of the communication between components is made by writing in the variables of each other, and that the reference to these variables is passed in the declaration of the component. An example would be the declaration of a module that contains a state variable for receiving data More, its signature defines that it should receive the reference of a variable from another component to write data by when the module is instantiated.

This makes the definition and change of the external representation of the model easier, in addition to make it more readable. It also facilitates the implementation of semiautomatic abstractions since it allows the simulation of disconnection between two components by limiting the value that a communication variable can assume to a value that indicates lack of communication. This is the case of the model of the GridMedia network, which will be detailed in section 6.6.

5.2.3 External Representation

The external representation of a model refers to the declaration of the components of the model and how they are organized. Some examples are the connections of logical

components in a circuit or the topology of a network. The analysis of the external representation is what determines the symmetry between the components.

The methodology of this work determines that the external representation of the model must be defined using assignment specifications. If the communication interfaces support, it should use the modular structures to pass the variables references and define the external representation in a modular and readable way. The methodology also expects that the external representation will be changed by the semiautomatic abstractions, defined with direct specifications. These changes are detailed in Section 5.4.

5.3 Generation of the symmetry group

As it will be detailed in section 5.4, the semiautomatic abstractions remove information that impacts the verification of the model and the biggest advantage of the symmetry reduction is no longer true, which is the validity of the results in the verification of $CTL*$ formulas that preserve the symmetry. In this work the use of symmetry serves as an orientation of which parts should be removed. Its use is justified by the smaller loss of behaviors it causes. Despite the removal of one of the symmetrical components eliminate states that do not contain valuations without the default values, at least all possible valuations for the state variables are present on the component that was kept.

The definition of symmetry between the components of the model is made by generating the symmetry groups. As mentioned in section 4.2.1, these groups are sets of permutations over the set of states of a model that preserve the transition relation. It has also been shown that the concept of permutation can be easily extended to the processes that are part of the model.

In all related work, the creation of the symmetry group is performed manually. This is because the task of determining automatically whether a permutation is an automorphism has a high computational cost. The problem is the same as the classical problem of isomorphism in two graphs [E M Clarke, 1999], and should be repeated for all possible permutations. Thus, the manual determination takes advantage of the knowledge that the programmer has about the model to generate the symmetry group and overcomes this high computational cost.

Our methodology analyzes the external representation of the model to determine the symmetry group. That is, it simply determine the symmetry between the components of the model, instead of trying to determine the symmetry between the states. If the system has been modeled as described in section 5.2 the task becomes easier

because the model can be viewed as a graph, where the components of the model are the vertices and connections between components are the edges. If a permutation σ of two components maintains the transition relation in the graph of the external representation of the model, then this permutation will be part of the symmetry group. Recall that a permutation between two components means the permutation among the states that have symmetric valuations for the state variables of both, as defined in section 4.2.1.

To illustrate, we will use the example of the token ring network of section 4.2.3. Figure 5.2 shows the graph of the network, composed asynchronously by the processes P_1 and P_2. The symmetry can be easily perceived and is highlighted by the dotted axis. In this case the permutation σ that exchanges the states where the processes P_1 and P_2 have the same values for the single state variable is (P_1P_2), which is also the generator of the symmetry group $G = \{e, (P_1P_2)\}$.

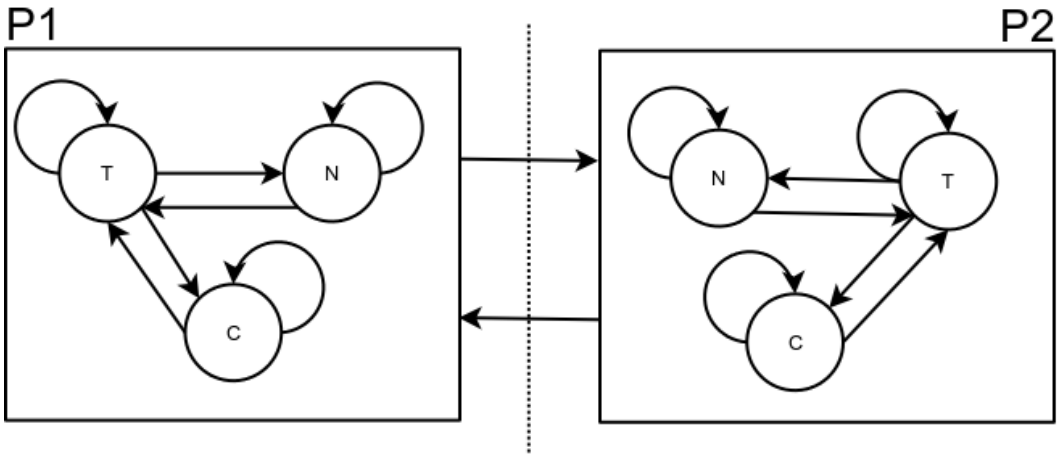

Figure 5.2. Symmetry in a Token Ring network with two processes

5.4 Semiautomatic abstractions

The implementation of abstractions in Model Checking is usually done in two ways: manual or automatic. In the first one the description of the model is edited to abstract some component. It requires intuition and is subject to impact the result of the verification if done naively. In contrast, the automatic techniques group or remove states without human intervention. In Clarke et al. [2003], for example, either the abstractions and the refinements on the model are made without human intervention.

The concept of semiautomatic abstractions defines that abstractions are implemented by human intervention, but there is not edition on the description of the model.

```
next(inbox) := {empty,message};
TRANS next(inbox) = empty;
```

Figure 5.3. Example of Semiautomatic Abstraction

The abstractions are made in terms of components, in contrast to the automatic methods, which make the reductions in terms of states. This is done using direct specifications that limit the transitions of the state variables of a component to only the transitions that do not alter the values of variables. The fact that the values of the state variables of a component will not be changes implies that logically the component is no longer part of the model. Consequently the number of reachable states is reduced as the state variables of the removed component do not contribute to the number of possible valuations in the model.

The methodology of this paper defines that the internal representation of the components should provide a default value for the communication variable, which limits the transitions of the other state variables. This is a valid and fairly common situation in many reactive systems, where components may only alter the values of its variables after receiving some information from another component. That is, if there is no communication, the component will always keep the same values for the state variables and hence the states defined by other valuations of these variables become unreachable. Not all models have this structure. In other cases, the definition of the model shall have default values for all the state variables, usually keeping the initial values.

The example in SMV language in figure 5.4 illustrates the definition of a communication variable that has a default value. The first line shows that the variable `inbox` can assume two values, `empty` and `message`, assigned non-deterministically. The first value indicates that there was no communication. The second indicates the arrival of data. If the direct specification on the second line is present, communication is interrupted because the only valid transition to the variable is the one that changes its value to `empty`.

The implementation of semiautomatic abstractions leads to the discard of states and transitions that impact on the verification of the model. It happens despite the fact the selection of which parts of the model will be abstracted is based on symmetry and symmetry reduction has no information loss. The reason is that because forcing the state variables of a component for values that will not change eliminates all the states whose valuations do not have the default values for the abstracted component. We can illustrate the information loss using the example in figure 5.4. If there are

two identical processes with the same variable `inbox` and one of them is abstracted by limiting the value of `inbox` to `empty` then the states were both variables are valuated to `message` is no longer reachable. So the abstractions presented in this work are under-approximations and as mentioned in section 4.1.2, only the positive results for the verification of formulas in the $\exists CTL$ logic are valid.

So, in case the verification of a property is correct on the abstract model, we conclude that it will also be correct in the original model. If the verification of a property is false, the result is inconclusive. In this situation the abstractions should be removed one by one and the check should be performed again until one of the following situations occurs: the property is proved to be true, which is a valid result; there are no more abstractions to be removed and the property will be checked over the original model; or the check does not end after a time limit.

Finally, only formulas that do not contain reference to the component being removed are valid. Although the semiautomatic abstractions logically remove a component, in fact it is still present in the model. So, if the $\exists CTL$ formula being verified has any reference to variables of the abstracted component, the model checker will judge it a valid formula. Consequently it may yield incorrect results because it contains reference to a component that will never change the values of its variables.

Chapter 6

The GridMedia Model

A model of the GridMedia network was implemented in the SMV language to validate the proposed methodology. The internal modeling represents the internal state machines of the types of the nodes participating in the network. In this model we have implemented only the server and client. The bootstrap was not considered. Each type of node is represented by a module. It contains the declaration of the state variables of the processes that compose the model and the local decisions of each node. The model implements only the data exchange on demand (pull).

The external modeling represents the topology of a P2P Live Streaming network. It contains the definition of the asynchronous composition of processes in the model and which nodes communicate with each other. The considered topology is static and there is no explicit disconnection between two nodes. However the fault situation is considered in the internal representation of modules, simulated by not receiving new messages.

6.1 P2P Live Streaming

The distribution of content using the client-server architecture is the traditional way of delivering content on the Internet. On it, a central computer called server waits for requests from computers called clients and replies them providing the requested service. Despite its popularity, the architecture has several limitations. The first one is its scalability. The centralization of resources such as bandwidth and processing in a single computer limits the number of users that can be served at any given time. There is also a safety issue. The server can be attacked and become unable to provide data and services. One example is the denial of service attack, where multiple malicious clients simultaneously forge requests to the server to exhaust its resources. The server can not

easily distinguish legitimate requests from the malicious requests. The consequence is that it may no longer be able to provide its services to valid users.

The Peer-to-Peer (P2P) architecture for data distribution is an alternative to client-server model and its limitations. On it, each participant performs network partnerships with other clients to exchange information. In other words, each client also acts as a server, and is responsible for providing the received information to other clients. In this way, the entry of a new participant in the P2P network increases the total bandwidth for data transmission, which makes the model highly scalable. Moreover, the fact that there is not a centralized point of resources reduces the chances of a successful malicious attack.

Applications based on the P2P concept have been adopted with great success for more than a decade to deliver various types of content over the Internet. Initially they were successfully implemented for files distribution. The main examples of such applications are Napster [Carlsson and Gustavsson, 2001], Kazaa [kazaa, 2007] and more recently BitTorrent [Bittorrent, 2007]. Other services based on P2P networks have also gained considerable popularity, such as the Distributed Hash Tables (DHT) [Tewari et al., 1998][Plaxton et al., 1997], which are distributed databases, where each network participant stores parts of the database.

More recently, the P2P architecture began to be used by applications broadcasting live content. Such applications are called P2P Live Streaming. They inherit all the characteristics of P2P systems for file sharing, in addition to a strong constraint of low-latency on the transmission. These systems are gaining great importance nowadays due to increasing demand for live video broadcasts on the Internet. Although each application has its own communication protocol, most of them shares the scheme of dissemination of information based on the BitTorrent network. In this scheme, the original information is fragmented into several pieces and distributed among the participants of the network; then the participants try to rebuild the original information by requesting and forwarding data to its partners. Several applications of this type have emerged in recent years. Among the most popular are the PPLive [PPlive, 2007], SopCast [Sopcast, 2007], CoolStreaming [Zhang et al., 2005b] and GridMedia [GridMedia, 2007]. Unlike other applications, the specifications of the GridMedia network were openly released by its authors in [Zhao et al., 2005] [Zhang et al., 2005a][Tang et al., 2006]. These specifications are crucial to model correctly the network participants and the overlay network; then the GridMedia network was chosen for this work due to open access to its specifications.

6.1.1 GridMedia Network

The topology of GridMedia network is of the mesh type, where participants create partnerships with other nodes without a rigid structure. This overlay network is established over the Internet. Figure 6.1 presents an example of the network organization.

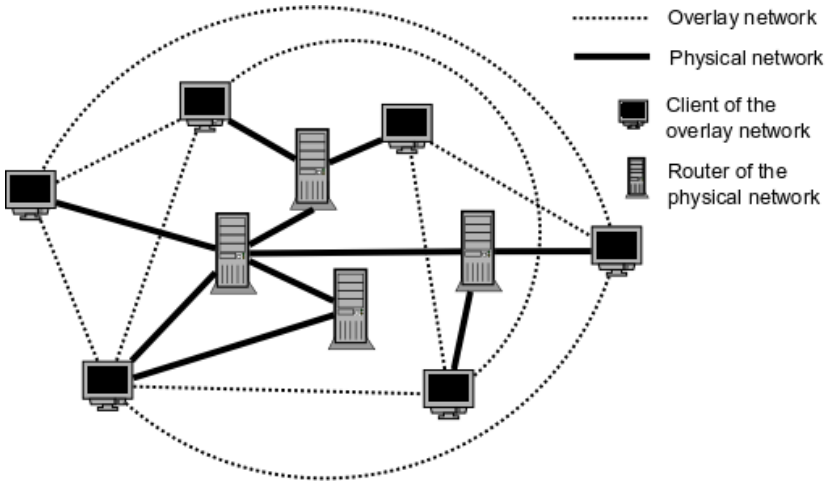

Figure 6.1. Organization of the overlay mesh network

In the GridMedia network, information is generated solely by a special node, called server. Unlike the concept of same name in the client-server model, this node does not serve all participants. It just generates the information of the live transmission, divides into small pieces called chunks and distributes each chunk to few participants. Each participant tries to reconstruct the original information by requesting to its neighbors the missing chunks, and forwards to its neighbors the chunks it has, but they have not.

The formation of topology occurs as follows. Each new participant in the transmission contacts a special node, called the bootstrap, which holds information on which clients are participating on the transmission at any given time. The bootstrap draws randomly a list containing a subset of the nodes that are in transmission, transfers to the new participant and also registers it, so the new client can be drawn on lists forwarded to the potential new participants. Then the new client attempts to start partnerships with the nodes on list it has received. Each client must report from time to time the bootstrap to inform it is still alive and involved in transmission.

The exchanges of data begin after the establishment of partnerships. GridMedia is a hybrid system and data transmissions can happen by two methods: by request (pull) or forwarding (push). In the pull method, the deliver of the information takes place after three steps. First, each client informs its partners constantly which chunks

it has and which chunks it needs through a data structure called chunk map. After receiving the chunk map of a partner, the client asks for a chunk that he does not have. The partner receives the request and then replies sending the requested chunk. Figure 6.2 illustrates the pull scheme in three steps.

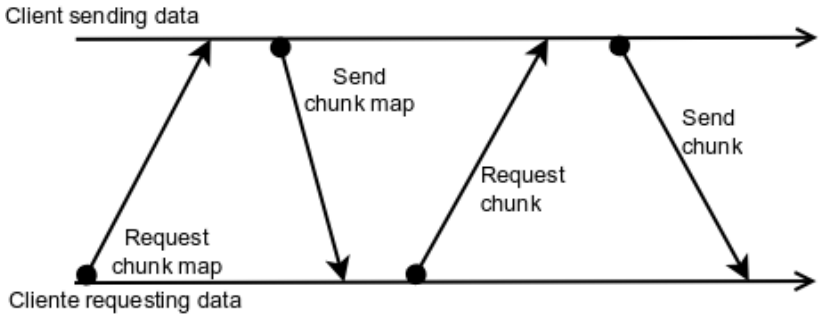

Figure 6.2. Data delivery by the pull method

The push method is complementary to the pull method. In push, each participant actively analyzes its partner's chunk maps. If it notices that one of its partners does not have an old chunk that it already owns, the participant forwards this chunk to the partner without request. The goal of this approach is to reduce the latency of the system by avoiding the three steps of the pull approach. Unfortunately the push approach enables some kinds of denial of service attacks, and most of the applications choose not to implement it. In this work the push approach was not considered. Only the pull method for data exchange was modeled, as described in section 6.3.

6.2 SMV Language

The SMV language [Cavada et al., 2005] is the description language implemented by NuSMV, which is the model checker used in this work. It allows the description of the state machine either by assignments or by direct specification [Pace et al., 2007]. The following sections detail the primitives for each type of model specification and provide the example of a simple bit inverters to illustrate their use in both cases.

6.2.1 Primitives for specification by assignment

The first primitive we detail is `MODULE`, which starts the declaration of a reusable component in the model, or the special module `main`, which can be instantiated only once. Each module has its own declaration of state variables, initiated by the primitive `VAR`. This field defines the names and types of the variables, which can be either of

```
MODULE inverter
VAR
   thebit: boolean;
ASSIGN
   init(thebit) := {0,1};
   next(thebit) := !thebit;

MODULE main
VAR
   myinversor: inverter;
```

Figure 6.3. Example of a bit inverter

atomic types, like integer or boolean, or data structures such as an enumerable set of values. The primitive `ASSIGN` marks the beginning of the assignments for variables, which are made through the primitive `init()` for the possible initial states, and `next()` to set the values of variables in the next state. In both cases it is possible to make non-deterministic assignments to variables.

Figure 6.2.1 shows an example of the use of the SMV assignment primitives. It defines the `inverter` module, which has only one state variable: `mybit`. The variable is of boolean type, has its initial value assigned non-deterministically and the value in the next state is always the negation of the value in the current state.

Each instance of a module is declared as a variable in the `VAR` field of the module that contains it. It is possible to pass the reference of a state variable from another instance as a parameter on the module declaration. The use of the `process` primitive before the declaration of the instance of a module indicates that the composition between the processes of the model is asynchronous. The field started by the primitive `FAIRNESS` contains one or more boolean formulas and indicates that only paths in which these formulas are infinitely true must be considered in check. This prevents that valid paths in the model, but impossible in a real situation, are considered.

The example in figure 6.2.1 presents an extended version of the inverter in figure 6.2.1. In this example, the module receives the variable to be reversed as a parameter, instead of modifying its own variable. Moreover, two asynchronous processes are instantiated, where each one reverses the value of the `thebit` variable of the other one when it is its turn to execute. In NuSMV, each process has a special internal input variable called `running` and when its value is true, indicates that it is the turn of the process to make its transitions. Finally, the declaration of `FAIRNESS` defines that only paths where the `running` variable of each process is infinitely true must be considered.

```
MODULE inverter (otherbit)
VAR
   thebit: boolean;
ASSIGN
   init(thebit) := {0,1};
   next(otherbit) := !otherbit;
FAIRNESS
   running

MODULE main
VAR
   inverter_a: process inverter( inverter_b.thebit);
   inverter_b: process inverter( inverter_a.thebit);
```

Figure 6.4. Example of two asynchronous inverters

```
INIT inverter_a.thebit = 1 & inverter_b.thebit = 0;
TRANS (inverter_a.thebit = 0 -> next(inverter_b.thebit = 1));
TRANS (inverter_a.thebit = 1 -> next(inverter_b.thebit = 0));
```

Figure 6.5. Example of direct specifications

6.2.2 Primitives for direct specification

The primitives for direct specification are `INIT` and `TRANS`. They can be used in more than one declaration and the initial states and possible transitions are those where the conjunction of all the specifications is true. Statements that use the `INIT` primitive constrain the initial states to those where the propositional formulas are true. `TRANS` restricts the transition relation to the transitions where the given propositional formulas are true.

Figure 6.2.2 provides examples of such specifications, which might be applied to the example in figure 6.2.1. The first statement, which contains the `INIT` primitive, restricts the initial states to those where the variable of the `inverter_a` process equals to one and the variable of the `inverter_b` process equals to zero. The lines containing the `TRANS` primitives restrict the valid transitions to those where the variable of the `inverter_b` process in the next state is the negation of the variable of the `inverter_a` process in the current state.

6.3 Internal Representation

The internal representation refers to the description of the state machines of the network participants. The model of the GridMedia network contains two types of participants: the server, that generates the content to be distributed, splits into pieces called chunks and distributes to the clients; and the client, which receives and forwards chunks to its neighbors to rebuild the original information.

6.3.1 Client

As mentioned in section 6.4, the module `connectionHandler` defines the state machine of a connection in the client. Its declaration expects the passage of three parameters: `my_chunk_buffer`, which is a reference to the variable `chunk_buffer` of the client itself; `outgoing_event`, a reference to the partner's message box, and `remote_chunk_buffer`, which is a reference to the `chunk_buffer` variable of its partner.

The only variable passed by reference which is altered by the module `connectionHandler` is `outgoing_event`. Any attribution can only be made if the partner's message box is empty. That is, when the value of `outgoing_event` equals `null`. This prevents that a second message overwrites the first received message before it is handled by the partner. The decision making to set the value of `outgoing_event` is based on the request model shown in figure 6.2. The client responds to requests made by its partner for chunks and chunk maps, and make the same requests to complete its own buffer.

Figure 6.6 shows the SMV code for the decision-making for variable `outgoing_event`. Note that in some cases the assignment is non-deterministic. For example, if the partner has two chunks that the client wants, he can request either `chunk1` or `chunk2`.

The module has two state variables: `event`, which is the incoming message box of the connection, and `last_chunk_map`, which stores the last view of which chunks the partner has. The variable `last_chunk_map` is initialized to `00`, indicating that the partner's chunk map is empty and/or it has not been requested yet. Its value may change when the incoming message is `chunk_map`. The new value will be a copy of the neighbor's buffer, received through the parameter `remote_chunk_buffer`. The `event` variable is always initialized to `null`, indicating that the connection has not received any messages. Also its value in the next state will always be set to `null`, indicating that the last received message has already been handled.

Each network client shares the same internal representation with others. All are

```
next(outgoing_event) := case
   ---- Uploading requests
   (event = request_map) & (outgoing_event = null) : {chunk_map, outgoing_event};
   (event = request_chunk1) & (outgoing_event = null) : chunk1;
   (event = request_chunk2) & (outgoing_event = null) : chunk2;
   ---- Downloading requests
   ((last_chunk_map & !my_chunk_buffer) = 0B2_11) &
    (outgoing_event = null): {request_chunk1, request_chunk2};
   ((last_chunk_map & !my_chunk_buffer) = 0B2_01) &
    (outgoing_event = null): request_chunk1;
   ((last_chunk_map & !my_chunk_buffer) = 0B2_10) &
     (outgoing_event = null): request_chunk2;
   -- Request chunk map only if my chunk buffer isn't full
   (my_chunk_buffer != 0B2_11) &
      (outgoing_event = null) : {request_map, outgoing_event};
   1: outgoing_event;
esac;
```

Figure 6.6. Assignment to variable outgoing_event in connectionHandler

instances of the `node` module, which defines the state machine of the clients. The signature of the module defines the receiving of six parameters, which are variables from the three processes that represent its partners: `neighbor_{x,y,z}_buffer` are the chunk maps; and `neighbor_{x,y,z}_event` are the message boxes.

The declaration of variables in the `node` module has only one state variable: `chunk_buffer`, which represents the chunk buffer that the client has in a given moment. The buffer is presented as a two-bit vector, where the rightmost bit represents the first produced chunk, and the leftmost bit represents the second chunk generated in the transmission. The value of `chunk_buffer` changes when one of the communication variables `neighbor_{x,y,z}_event`, received as a parameter, contains a new chunk, indicated by the messages `chunk1` or `chunk2`.

The `node` module also contains the declaration of three instances of the `connectionHandler` module, representing the connections that each client keeps with exactly three partners. The composition between the instances of the module is synchronous. That is, when it is time of an instance of `note` to run, all the variables of the three instances of `connectionHandler` module and `chunk_buffer` variable will be altered simultaneously.

```
next(outgoing_event) := case
   ---- Uploading requests
   (event = request_map) & (outgoing_event = null) : chunk_map;
   (event = request_chunk1) & (outgoing_event = null): chunk1;
   (event = request_chunk2) & (outgoing_event = null): chunk2;

   1: outgoing_event;
esac;
```

Figure 6.7. Assigning values to outgoing_event in connectionHandlerServer

6.3.2 Server

Likewise the `connectionHandler` module defines the state machine of a connection in a client, the module `connectionHandlerServer` defines a connection in the server. In fact, `connectionHandlerServer` is a subset of `connectionHandler` because the server only produce chunks and do not request them makes. Figure 6.7 presents the SMV code of the assigning scheme to the variable `outgoing_event`, where the server only replies to requests from clients. Since the server does not send requests to clients, there is no need for a variable that stores the last view of the partner's chunk map, like the `last_chunk_map` variable in `connectionHandler`.

The `server` module that describes the server has similarities with the `node` module. First, the interface of both is similar. The `server` module expects four parameters, which are references to variables of the clients that are directly connected to it: `node_{a,b}_buffer` are the chunk maps; and `event_node_{a,b}` are the message boxes. Note that, different from the `node` module which receives six parameters, `server` receives only four. This is because the server maintains connection with only two nodes, different from clients that have three connections each.

Similar to the `node` module, the variable declaration of `server` has only one state variable: `chunk_buffer`. It is defined as a two-bit vector and represents the chunk buffer of the server at a given moment. Unlike the clients, which fill out their buffers whenever a new chunk is sent by a partner, the server completes its buffer creating sequentially chunks, represented in the model by bits. The first chunk is the rightmost bit. There is also the declaration of two instances of the `connectionHandlerServer` module, which represents the two connections of the server. The composition between these instances is synchronous.

6.4 Communication Interface

Each connection is represented as an instance of the `connectionHandler` module for the clients and as an instance of the `connectionHandlerServer` module for the server. These modules will be detailed in sections 6.3.1 and 6.3.2 respectively. Both modules contain the declaration of the state variable `event`, which simulates a box of incoming messages. Thus, writing on the variable of a process that will receive the message simulates sending a message to it. This holds both for the communication between clients and between clients and server.

The model defines a six types of messages. Along with the special value `null`, which indicates that no message was received, the state variable `event` can assume seven values, as follows:

- `null`: the message box is empty and can receive new messages.
- `request_map`: request the chunk map to a partner.
- `chunk_map`: sending of the chunk map.
- `request_chunk1`: request chunk number 1.
- `request_chunk2`: request chunk number 2.
- `Chunk1`: sending the chunk number 1.
- `Chunk2`: sending the chunk number 2.

6.5 External Representation

External representation refers to the modeling of the network topology. The model generated in this work comprises the most relevant aspects of possible interactions between participants of the GridMedia network. The first is the direct communication between the server and clients. The second aspect is the communication between clients that communicate directly to server, and clients that do not. Finally, the third aspect is the communication between two clients that are on the same distance to the server.

The topology of the GridMedia network implemented this work is shown in figure 6.8. It is easy to see that this topology is highly symmetrical. It has the *server* connected to nodes A and B in a first level. In a second level there are the nodes C and D, which have connections to both A and B, and between themselves. Note that all relevant aspects of the interactions are represented: the direct communication

between the server and clients is represented by ($server \rightarrow A$) and ($server \rightarrow B$); the communication between clients on the first level with clients on the second level happens in ($A \leftrightarrow C$),($A \leftrightarrow D$),($B \leftrightarrow C$) e ($B \leftrightarrow D$); finally, the connection between two clients at the same level is represented by ($C \leftrightarrow D$).

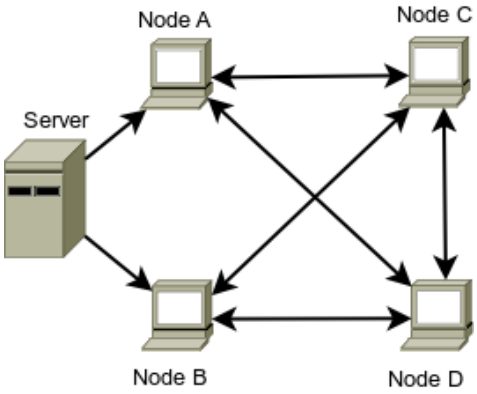

Figure 6.8. Topology of the GridMedia network

The modeling of the topology using the SMV language is made in the declaration of the processes that represent the network participants. It is composed of five processes. The use of the `process` primitive defines that the model is asynchronous. The passing of arguments provides a reference of the message boxes and the chunk buffers of each partner of a node. Figure 6.9 shows the SMV code for definition of the topology. The attributes `connection{1,2,3}` of each node is a reference to one of its instances of the `connectionHandlerServer` mode, in the case of the server, or `connectionHandler` in case of the clients.

6.6 Abstractions in the Model

The abstractions in this work refer to the removal of entire components from the model and the consequent removal of states. The removal of a component may be the elimination of a connection between two participants, represented by the pair of variables that simulate message boxes, or the removal of a participant in the network, represented by the removal of all the connections that it maintains.

The connection between two participants is abstracted by enabling the `TRANS` statements that restrict the valid transitions to those where the values of the communication variables is not changed. Figure 6.11 shows the removal of three connections of the model: between node A and node C, between C and D, and between node A and D. The `TRANS` statements are shown in figure 6.10.

```
server: process server( node_A.connection1.event, node_A.chunk_buffer,
                        node_B.connection1.event, node_B.chunk_buffer);

node_A: process node( server.connection1.event, server.chunk_buffer,
                      node_C.connection1.event, node_C.chunk_buffer,
                      node_D.connection1.event, node_D.chunk_buffer);

node_B: process node( server.connection2.event, server.chunk_buffer,
                      node_C.connection2.event, node_C.chunk_buffer,
                      node_D.connection2.event, node_D.chunk_buffer);

node_C: process node( node_A.connection2.event, node_A.chunk_buffer,
                      node_B.connection2.event, node_B.chunk_buffer,
                      node_D.connection3.event, node_D.chunk_buffer);

node_D: process node( node_A.connection3.event, node_A.chunk_buffer,
                      node_B.connection3.event, node_B.chunk_buffer,
                      node_C.connection3.event, node_C.chunk_buffer);
```

Figure 6.9. Definition of the topology in SMV language

```
TRANS (next(node_A.connection2.event) = null) &
      (next(node_C.connection1.event) = null)
TRANS (next(node_A.connection3.event) = null) &
      (next(node_D.connection1.event) = null)
TRANS (next(node_C.connection3.event) = null) &
      (next(node_D.connection3.event) = null)
```

Figure 6.10. Example of direct specifications that remove connections

The abstraction of a participant is made similarly. To remove it, simply enable the direct specifications that restrict any changes to all its state variables that simulate the receipt of a message. Figure 6.13 shows the example of the removal of the participant D. In this case all the `TRANS` statements that removed the connections to the nodes A, B and C were enabled, as shown in figure 6.12.

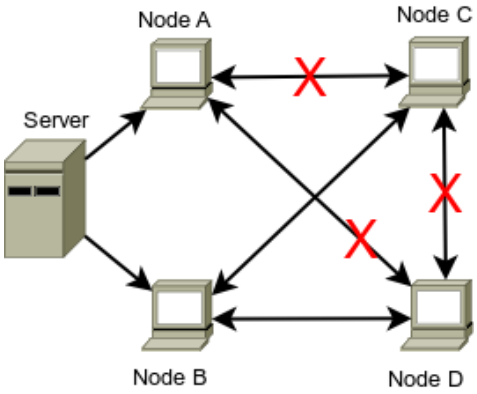

Figure 6.11. Example of removal of connections from the topology

```
TRANS (next(node_A.connection3.event) = null) &
      (next(node_D.connection1.event) = null)
TRANS (next(node_B.connection3.event) = null) &
      (next(node_D.connection2.event) = null)
TRANS (next(node_C.connection3.event) = null) &
      (next(node_D.connection3.event) = null)
```

Figure 6.12. Direct specifications that remove participant D

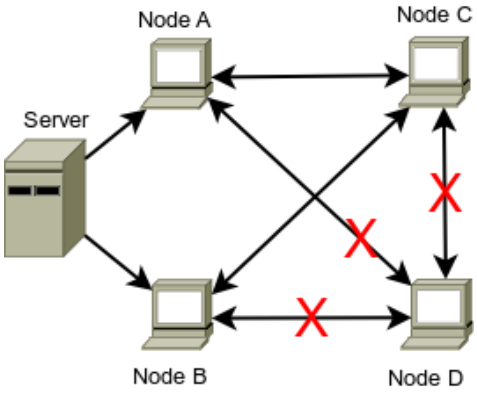

Figure 6.13. Example of abstraction of a participant in the network

Chapter 7

Experimental Results

The chapter begins presenting the description of the computing environment used at this work, composed of hardware and software. It is important to emphasize that the results are independent of the computational resources used, as it was the case of chosen type of symbolic representation used by the model checker.

Then we present the procedure of generating the symmetry group for the model of the GridMedia network, as described in chapter 6. Starting from the symmetry group, we generate three abstract models. Then we use these models to check three properties in $\exists CTL$ logic. Finally we show a quantitative comparison of the reductions obtained with the generated abstract models.

7.1 Verification Environment

All experiments were performed on the processing server of the Luar laboratory. Its configuration consists of an Intel Xeon X3323 Quad-core 2.5-GHz clock, 3 megabytes of cache per core and 16 Gigabytes of RAM. The operating system is OpenSuSE Linux 11.1 for the x86-64 architecture.

The chosen model checker was the NuSMV [Cimatti et al., 2000], version 2.4.3. It is a symbolic model checker originated from the re-engineering, reimplementation and extension of the pioneer SMV [McMillan, 1992], the first model checker based on BDDs. Its goal is to be the state of the art in symbolic verification, and address both the verification of models on an industrial scale as a tool for further studies in formal verification.

This new implementation has a modular architecture, open source and supports both verifications based on BDDs and on SAT solvers. In case of the verification using BDDs, the NuSMV uses the library CUDD [Somenzi, 1998] and is able to check

both properties in CTL and LTL. In the verification using techniques of propositional satisfiability, it integrates with the SAT solvers ZChaff [Fu et al., 2004] and MiniSAT [Sörensson and Eén, 2005] and can only verify LTL properties.

7.2 Symmetry Group

As mentioned in section 5.3, the formal definition of symmetry is made through the generation of symmetry group, which contains the permutations of components of the model where the transition relation is maintained. The section cites that the process is manual and requires knowledge of the model structure.

For the model of the GridMedia network, described in chapter 6, it is easy to see the symmetry between the clients that are the in same distances from the server. I.e., the clients A and B, which are directly connected to the server, are symmetrical with each other. Likewise the clients C and D, which are at a distance of size two away from the server. Figure 7.1 shows the axis of symmetry in the network model.

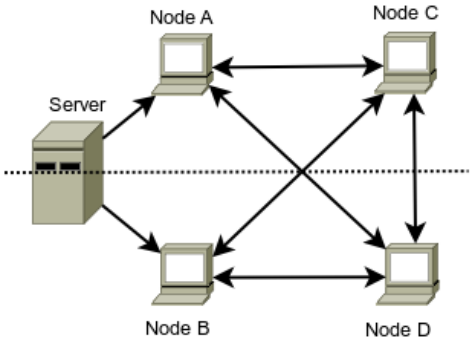

Figure 7.1. Symmetry in the original model

Thus, to generate the symmetry group we simply exchange the symmetrical components and verify if the transition relation model is preserved. The graph representing the GridMedia network contains five components, being four of them identical. They are connected by twelve edges only. Note that there are no incoming edges to the server; this is because the server only sends data and does not request anything. Table 7.2 shows the method for generating the symmetry group. Through its analysis is possible to notice that the generation of the symmetry group over the permutations of components demands a much smaller effort than the generation over the permutations of states.

Table 7.2 shows that the permutations between clients A and B, and between the clients C and D preserve the transition relation. I.e., all edges present

Edge	Permutation			
	$(Node_A Node_B)$	$(Node_C Node_D)$	$(Node_A Node_C)$	$(Node_A Node_B Node_C Node_D)$
$S \to A$	$S \to B$	$S \to A$	~~$S \to C$~~	$S \to B$
$S \to B$	$S \to A$	$S \to B$	$S \to B$	~~$S \to C$~~
$A \to C$	$B \to C$	$A \to D$	$C \to A$	$B \to D$
$C \to A$	$C \to B$	$D \to A$	$A \to C$	$D \to B$
$A \to D$	$B \to D$	$A \to C$	$C \to D$	~~$B \to A$~~
$D \to A$	$D \to B$	$C \to A$	$D \to C$	~~$A \to B$~~
$B \to C$	$A \to C$	$B \to D$	~~$B \to A$~~	$C \to D$
$C \to B$	$C \to A$	$D \to B$	~~$A \to B$~~	$D \to C$
$B \to D$	$A \to D$	$B \to C$	$B \to D$	$C \to A$
$D \to B$	$D \to A$	$C \to B$	$D \to B$	$A \to C$
$C \to D$	$C \to D$	$D \to C$	$A \to D$	$D \to A$
$D \to C$	$D \to C$	$C \to D$	$D \to A$	$A \to D$
Preserved the transition relation	Yes	Yes	No	No

Table 7.1. Generation of the symmetry group

in these permutations also exist in the original model. However, other permutations of identical components did not preserve the transition relation. This was the case of $(Node_A Node_C)$, shown in the table. The marked edges do not exist in transition relation in the original model. The permutations $(Node_A Node_D)$, $(Node_B Node_C)$ and $(Node_B Node_D)$ were omitted from the table, but it is easy to see that they do not preserve the transition relation by analogy to $(Node_A Node_C)$. The permutation $(Node_A Node_B Node_C Node_D)$ is shown in the table and it is seen that it does not preserve the relation transition. Thus we conclude that the permutations $\langle (Node_A Node_B), (Node_C NodoD) \rangle$ are generators of the symmetry group $G = \{e, (Node_A Node_B), (Node_C Node_D), (Node_A Node_B)(Node_C Node_D)\}$ with respect to the original model.

7.3 Generation of Abstract Models

Section 5.3 showed that the generation of the symmetry group will guide the abstractions in the GridMedia model. The verification process of the original model must first create an abstract model, based on permutations of the symmetry group G which does not remove components which are mentioned in the temporal formula ϕ to be verified. Also section 5.4 showed that only properties in $\exists CTL$ logic are valid in the verification of abstract models. Thus, if $\phi \in \exists CTL$, the abstract model can be checked against the formula and if the result is correct, we conclude that it will also be in the original model.

The following sections show the generation of abstract models for the verification of three properties over the model of the GridMedia network. The first checks if there is the case where two nodes are at the same distance of size two from the server, but one can fully recreate the original information before the other has even received any data yet. The second property checks if exists the possibility that clients which are in a distance of size two from the server can recreate the original information before clients that are directly connected to the server do it. That is, if the information can pass by two different paths. The third and last property tests if there is a path where both a client on the first level and client at the second level recreate the original information at any instant.

7.3.1 Property 1: Partial Data Reconstruction

The first property to check over the GridMedia model is the existence of the situation where a client that is in distance of size two to the server can recreate the original information before the other node, in the same distance to the server, receives any portion of the information. The formula ϕ_1 below verifies this case, represented by the data buffer of the node D containing the two bits that represent all the chunks of transmission, and the buffer of the node C being empty:

$$\begin{aligned} \phi_1 = \quad & EF \quad (node\_C.chunk\_buffer = 0B2\_00) \ \& \\ & \qquad (node\_D.chunk\_buffer = 0B2\_11) \end{aligned}$$

As described in section 5.4, only components that are not referenced in ϕ_1 can be abstracted. In this case, both the $Node\_C$ and $Node\_D$ are mentioned in the formula. Thus, the orbit generated by the permutation $\sigma = (Node_C Node_D)$ can not be used, since both components are part of the formula. Given the permutations of the symmetry group G, the permutation $\sigma = (Node_A Node_B)$ is the only valid for generating the abstract model and we can choose any one of the clients as the representative of the orbit generated by the permutation. In this example we chose $Node_A$ as the representative, and consequently $Node_B$ is abstracted. The abstract model is represented in figure 7.2.

The abstraction of component B is done semiautomatically through the direct specifications shown in figure 7.3. The `TRANS` statements limit the transitions of the `event` variables that are used to simulate communication with B. The transitions are restricted to only those that keep the values of the variables always as `null`. Consequently the states where there are transitions to other values of the internal variables of B, in exception of the initial values, will no longer be reachable since the internal representation defines that there are transitions of values only when there is communication

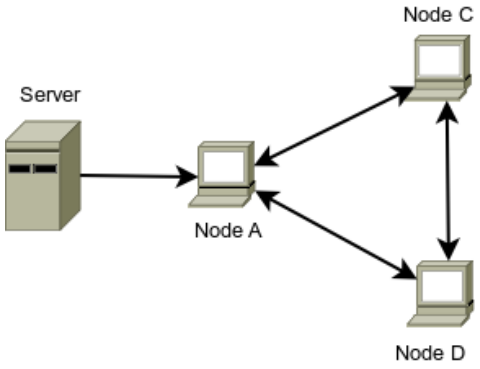

Figure 7.2. Abstract model of Example 1

```
TRANS (next(server.connection2.event) = null) &
      (next(node_B.connection1.event) = null)
TRANS (next(node_B.connection2.event) = null) &
      (next(node_C.connection2.event) = null)
TRANS (next(node_B.connection3.event) = null) &
      (next(node_D.connection2.event) = null)
```

Figure 7.3. Direct specifications that abstract node B

with other participants.

The verification of the formula ϕ_1 over the abstract model was correct, and we conclude that it is also correct if verified over the original model. That is, there is a situation where only one of the clients at the same distance of size two from server can recreate the transmission, but the other client has not received any part of the data.

7.3.2 Property 2: Multiple Paths

An expected property in P2P networks is that there exists the possibility where clients that are not directly connected to the server can recreate the original information before the clients connected to the server. This is a valid situation in P2P live streaming networks, as shown in section 6.1. This should be possible since the server partitions the original information and distributes different parts to the nodes connected directly to it. So it is expected that there is more than one path that information can take to reach any participant in the network. The ϕ_2 formula below checks the case where each client at the first level has a different part of the information, but a client on the second level already has the full information.

$$\begin{aligned}\phi_2 = \quad EF \quad & (node\_A.chunk\_buffer = 0B2\_10)\ \& \\ & (node\_B.chunk\_buffer = 0B2\_01)\ \& \\ & (node\_C.chunk\_buffer = 0B2\_11)\end{aligned}$$

Similar to section 7.3.1, we must first determine which components are referenced in the $\exists CTL$ formula, since only those who are not mentioned can be abstracted. In the case of ϕ_2, $Node_A$, $Node\_B$ and $Node\_C$ are part of the formula. The two elements of the permutation $\sigma = (Node_A Node_B)$ contains elements in ϕ_2, and then it can not be used to abstract components. In the permutation $\sigma = (Node_C Node_D)$ only $Node_C$ is mentioned in the formula. Then it is automatically chosen as the representative of the orbit generated by the permutation. Consequently the node D is abstracted. The representation of the abstract model is shown in figure 7.4.

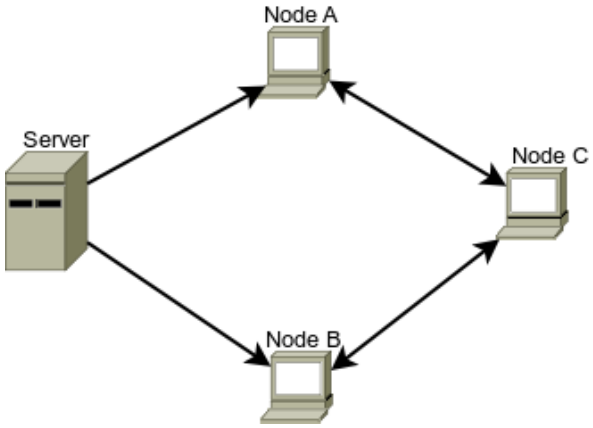

Figure 7.4. Abstract model of Example 2

The abstraction of node D and its states is done by the direct specifications shown in Figure 7.5. The `TRANS` statements restrict the transitions of the `event` variables, which simulate the communication between node D and its partners, to only the transitions that keeps its current value equal to `null`. This eliminates any communication with the node D. The internal representation of the clients defines that the other state variables will change their values only if there is communication with other network participants. Thus, the removal of communication implies that the variables of D will not suffer transitions.

The verification of ϕ_2 over the abstract model has proved to be correct, and we can conclude that ϕ_2 is also correct in the verification over the original model. Therefore, we conclude based on the model of the GridMedia network that clients that are not directly linked to the server can reconstruct the original transmission before clients in the first level.

```
TRANS (next(node_A.connection3.event) = null) &
      (next(node_D.connection1.event) = null)
TRANS (next(node_B.connection3.event) = null) &
      (next(node_D.connection2.event) = null)
TRANS (next(node_C.connection3.event) = null) &
      (next(node_D.connection3.event) = null)
```

Figure 7.5. Direct specifications that abstract the participant D

7.3.3 Property 3: Successful transmission

A desirable property in P2P systems like GridMedia is that participants in different levels are able to reconstruct the original information at some point in time. The formula ϕ_3 below checks if there is the situation where $Node_B$, at distance of size one from the server, and $Node_D$ at distance of size two, can both reconstruct the information along time.

$$\phi_3 = \quad EF \quad (((node\_B.chunk\_buffer = 0B2\_11)\& \\ (node\_D.chunk\_buffer = 0B2\_11))$$

Again, only components that are not referenced in ϕ_3 can be abstracted. In this case both $Node\_B$ and $Node\_D$ are part of the formula. The permutations $\sigma = (Node_A Node_B)$ and $\sigma = (Node_C Node_D)$ may be used to generate the abstract model, as both have members that are not in ϕ_3. The processes $Node_B$ and $Node_D$ are automatically selected as representatives of the orbits generated by permutation, as both are part of the formula. Consequently $Node_A$and $Node_C$ are abstracted. The abstract model is represented in figure 7.6.

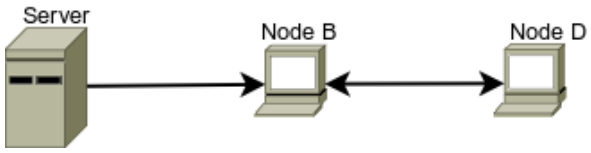

Figure 7.6. Abstract model of Example 3

Figure 7.7 shows the direct specifications that abstract semiautomatically the clients A and C. Again the `TRANS` statements restrict the transitions of the model to only those where the value of the `event` variables, which simulate the communication with $Node_A$ and $Node_C$, remain unchanged as `null`. Consequently all the state variables of $Node_A$ and $Node_C$ maintain always the same values.

The verification of ϕ_3 over the abstract model stated that property is correct, and we conclude it will also be correct over the original model. That is, there is a situation

```
TRANS (next(server.connection1.event) = null) &
      (next(node_A.connection1.event) = null)
TRANS (next(node_A.connection2.event) = null) &
      (next(node_C.connection1.event) = null)
TRANS (next(node_B.connection2.event) = null) &
      (next(node_C.connection2.event) = null)
TRANS (next(node_A.connection3.event) = null) &
      (next(node_D.connection1.event) = null)
TRANS (next(node_C.connection3.event) = null) &
      (next(node_D.connection3.event) = null)
```

Figure 7.7. Direct specifications that abstract nodes A and C

where the nodes B and D, which are at a distance of size one and two from the server respectively, are both able to fully recreate the information. By the symmetry of the system, we conclude that the same situation exists for nodes A and C. So we can conclude that there are paths that make any network client recreate the information in a given instant of time.

7.4 Comparison of the reductions

The original model has a total number of states in the order of 2^{73}, referring to all the possible valuations over the state variables. Its complexity has not allowed the calculation of the diameter of the graph model, nor the number of reachable states ended after more than two weeks of intensive computation running on the processing server of the Luar laboratory. The same happened in the verification of properties ϕ_1, ϕ_2 e ϕ_3 presented in sections 7.3.1,7.3.2,7.3.3 respectively.

By contrast the computations performed on all abstract models finished in a bit more than two minutes for all cases. Table 7.4 shows a quantitative comparison between the verification of abstract models. It presents the diameter of the obtained graph, the number of reachable states, the CPU time spent to verify the property and the total time between the beginning and the end of the NuSMV process, which include CPU time, plus context switches and rescheduling.

The most notable fact was the reduction of exponential order in the number of reachable states, from about 2^{19} in the case of Example 2 to approximately 2^9 in Example 3. It gives the dimension of the reduction in a model when a single extra component is abstracted. The same result can be observed by comparing the found diameters. In Examples 1 and 2, where only one process was abstracted, the diameter

Abstract Model	Diameter	Reachable States	Total Time (s)	CPU Time (s)
Example 1	41	500079	28,834	0,044
Example 2	41	814135	69,440	0,032
Example 3	28	793	0,222	0,012

Table 7.2. Comparison between the abstract models

was 41; in the Example 3, which had two processes removed, the value dropped to 28.

We can also observe a considerable growth of the CPU time on the verification, even when the numbers of reachable states were of the same order, as in the comparison of Example 1 and Example 2. It exemplifies the state-space explosion problem, and how the verification time is impacted by the number of states on the model. Due to this factor that the computations of the original model did not finish after more than two weeks of intensive computation.

Chapter 8

Conclusion

Model Checking, if carried out naively and without knowledge of modeling techniques, overloads the task of model checker and may limit the result of the verification. It is desirable, whenever possible, that the programmer of the model supply information to the model checker and behave actively to alleviate the problem of the state-space explosion. This work contributes in this direction by showing how to make reductions in symmetric models of reactive systems in a simple way, without compromising the verification result.

The methodology presented is based on the model reduction in terms of components, in contrast to the reduction by states. The reasoning about components is more intuitive since the modeling is done at this level. It is also more practical because it deals with a smaller amount of objects. Consequently the identification of symmetry is simpler, as well as the implementation of abstractions. Furthermore, the quantitative results showed that the removal of components can result in significant reduction in the number of reachable states and is capable of allowing the verification of models that were previously intractable. It was the case of our example, where the verification of properties in the original model did not finish after two weeks of intensive computation on a dedicated server. The same computations have ended in less than three minutes for all the abstract model.

The identification of symmetry proved to be a strong evidence of how to choose the components to be abstracted in the model. The abstractions remove information that makes it impossible to check properties in a temporal logic other than $\exists CTL$. In return, the exploitation of symmetry eliminated redundant behavior and minimizes the information loss. This can be noticed by the fact that it was not necessary to refine the abstract model in any situation because an inconclusive result occurred. The limitation of the technique is that it applies only to highly symmetric models of reactive systems.

Fortunately this is the case for several real models, like the example of GridMedia network.

The technique of semiautomatic abstractions also proved to be very useful. In addition to make the implementation of abstractions of components easier, it also avoids errors caused by the edition of the model description. Its utilization is especially useful in the verification of several properties over the same model, which generally require the generation of many different abstract models.

8.1 Future Work

I intend to continue with the use of semiautomatic abstractions in three ways. The first is the validation of the methodology based on symmetry in the verification of other models. Especially I want to check its usefulness in the verification of synchronous models. The second is to extend the use of semiautomatic abstractions by under-approximation to the study of dynamic aspects of P2P networks. In the model of the GridMedia network, i.e., you can remove a client or a connection to simulate scenarios of disconnection or even to check the process of forming the network topology. Third, the under-approximation approach of this work can be used not only to verify properties in $\exists CTL$, but also to falsify properties in $\forall CTL$, which means try to prove that some property does not hold on the model by finding a counter-example.

Then I intend to intensify the work on symmetry reduction, and conduct the verification of CTL and LTL properties in quotient models. So I shall use the concepts of symmetry presented in section 4.2. First, the method of generating the symmetry group can be extended to determine the invariance groups based on the maximal boolean subformulas. It will be necessary to create a method of choosing the representatives of the orbits without breaking the symmetry. Once the orbit relations and choice of the representatives are generated, I foresee two alternatives to the generation of the quotient model. The first is the automatic generation of `TRANS` statements that eliminate symmetric valuations using preprocessor macros or change the code of NuSMV for the removal of states that are not representatives of the orbits.

Finally, I want to change the model checker NuSMV, so it may allows the selection of `TRANS` statements present in the model by passing parameters in the program call, in a similar manner to what occurs with the specifications in temporal logic. This would avoid the need to edit the file containing the model to define the semiautomatic abstractions.

Bibliography

Bakhshi, R. and Gurov, D. (2007). Verification of peer-to-peer algorithms: A case study. *Electron. Notes Theor. Comput. Sci.*, 181:35--47.

Barner, S. and Grumberg, O. (2005). Combining symmetry reduction and under-approximation for symbolic model checking. *Form. Methods Syst. Des.*, 27(1-2):29--66.

Bittorrent (2007). Bittorrent. `http://www.bittorrent.com`.

Borgström, J., Nestmann, U., Alima, L. O., and Gurov, D. (2004). Verifying a Structured Peer-to-peer Overlay Network: The Static Case. Technical report, École Polytechnique Fédérale de Lausanne.

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

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293--318.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1990). Symbolic model checking: 10^{20} states and beyond. *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439.

Carlsson, B. and Gustavsson, R. (2001). The rise and fall of napster - an evolutionary approach. In *AMT '01: Proceedings of the 6th International Computer Science Conference on Active Media Technology*, pages 347--354, London, UK. Springer-Verlag.

Cavada, R., Cimatti, A., Jochim, C. A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., and Tchaltsev, A. (2005). *NuSMV 2.4 User Manual*.

Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (2000). Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752--794.

Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52--71, London, UK. Springer-Verlag.

Clarke, E. M., Enders, R., Filkorn, T., and Jha, S. (1996). Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77--104.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512--1542.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238--252, New York, NY, USA. ACM.

Drager, K., Finkbeiner, B., and Podelski, A. (2009). Directed model checking with distance-preserving abstractions. *Int. J. Softw. Tools Technol. Transf.*, 11(1):27--37.

E M Clarke, Grumberg O., P. D. (1999). *Model Checking*. The Mit Press.

Emerson, E. A. and Halpern, J. Y. (1986). "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151--178.

Emerson, E. A., Sistla, A. P., and Weyl, H. (1993). Symmetry and model checking. *5th International Conference on Computer-Aided Verification*, 697.

Emerson, E. A. and Trefler, R. J. (1999). From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *In Conference on Correct Hardware Design and Verification Methods*, pages 142--156. Springer.

Fu, Z., Mahajan, Y., and Malik, S. (2004). New features of sat'04 version of zchaff. In *The International Conference on Theory and Applications of Satisfiability Testing*.

GridMedia (2007). GridMedia.
`http://www.gridmedia.com.cn`.

Grumberg, O. and Long, D. E. (1991). Model checking and modular verification. In *CONCUR '91: Proceedings of the 2nd International Conference on Concurrency Theory*, pages 250--265, London, UK. Springer-Verlag.

Ip, C. N. and Dill, D. L. (1993). Better verification through symmetry. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 97--111, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Jha, S. (1996). *Symmetry and Induction in Model Checking.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

kazaa (2007). Kazaa. `http://www.kazaa.com`.

Lane, S. M. and Birkhoff, G. (1988). *Algebra: Third Edition.* AMS Chelsea Publishing.

Lee, W., Pardo, A., Jang, J.-Y., Hachtel, G., and Somenzi, F. (1996). Tearing based automatic abstraction for ctl model checking. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 76--81, Washington, DC, USA. IEEE Computer Society.

McMillan, K. L. (1992). *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University.

Miller, A., Donaldson, A., and Calder, M. (2006). Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8.

Pace, G., Prisacariu, C., and Schneider, G. (2007). Model Checking Contracts –a case study. In *5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of Lecture Notes in Computer Science, pages 82--97, Tokyo, Japan. Springer.

Plaxton, C. G., Rajaraman, R., and Richa, A. W. (1997). Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311--320, New York, NY, USA. ACM.

Pnueli, A. (1977). The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57.

PPlive (2007). PPLive. `http://www.pplive.com`.

Sistla, A. P. and Godefroid, P. (2004). Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702--734.

Sistla, A. P., Gyuris, V., and Emerson, E. A. (2000). Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133--166.

Somenzi, F. (1998). Cudd: Cu decision diagram package release 2.2.0.

Sopcast (2007). Sopcast.
`http://www.sopcast.com`.

Sörensson, N. and Eén, N. (2005). Minisat vl.13 - a SAT solver with conflict-clause minimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, St. Andrews, location, UK. Springer-Verlag.

Tang, Y., Sun, L., Zhang, M., Yang, S., and Zhong, Y. (2006). A novel distributed and practical incentive mechanism for peer to peer live video streaming. In *ICME 2006, IEEE International Conference on Multimedia & Expo, Toronto, Canada.*

Tewari, R., Dahlin, M., Vin, H. M., and Kay, J. S. (1998). Beyond hierarchies: Design considerations for distributed caching on the internet. Technical report, in Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS).

Zhang, M., Luo, J.-G., Zhao, L., and Yang, S.-Q. (2005a). A peer-to-peer network for live media streaming using a push-pull approach. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 287--290, New York, NY, USA. ACM Press.

Zhang, X., Liu, J., Li, B., and Yum, Y. S. P. (2005b). Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 3:2102--2111 vol. 3.

Zhao, L., Luo, J.-G., Zhang, M., Fu, W.-J., Luo, J., Zhang, Y.-F., and Yang, S.-Q. (2005). Gridmedia: A practical peer-to-peer based live video streaming system. In *Multimedia Signal Processing, 2005 IEEE 7th Workshop*, pages 1–4.